USENIX Association

Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)

April 28–30, 2025
Philadelphia, PA, USA

© 2025 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-46-5

Conference Organizers

Program Co-Chairs

Theophilus A. Benson, *Carnegie Mellon University*
Radhika Niranjan Mysore, *VMware Research Group*

Program Committee

Soheil Abbasloo, *Microsoft Research*
Sangeetha Abdu Jyothi, *University of California, Irvine, and VMware Research*
Anubhavnidhi "Archie" Abhashkumar, *ByteDance Inc.*
Fawad Ahmad, *Rochester Institute of Technology*
Zahaib Akhtar, *Amazon Prime Video*
Ramnatthan Alagappan, *University of Illinois at Urbana–Champaign and VMware Research*
Emmanuel Amaro, *University of California, Berkeley*
Ganesh Ananthanarayanan, *Microsoft*
Gianni Antichi, *Politecnico di Milano and Queen Mary University of London*
Roshan Ayyalasomayajula, *University at Buffalo*
Wei Bai, *Microsoft Research*
Tom Barbette, *Université catholique de Louvain*
Noman Bashir, *Massachusetts Institute of Technology*
Ryan Beckett, *Microsoft*
Ran Ben Basat, *University College London*
Yaniv Ben-Itzhak, *VMware Research*
Ankit Bhardwaj, *MIT CSAIL*
Francesco Bronzino, *École normale supérieure de Lyon*
Fabián Bustamante, *Northwestern University*
Matthew Caesar, *University of Illinois at Urbana–Champaign*
Balakrishnan Chandrasekaran, *Vrije Universiteit Amsterdam*
Ang Chen, *University of Michigan*
Kai Chen, *The Hong Kong University of Science and Technology*
Kang Chen, *Tsinghua University*
Ruichuan Chen, *Nokia Bell Labs*
Xiaoqi Chen, *Broadcom*
Xusheng Chen, *Huawei Cloud*
Peng Cheng, *Microsoft Research*
Yue Cheng, *University of Virginia*
Marco Chiesa, *KTH Royal Institute of Technology*
Mosharaf Chowdhury, *University of Michigan*
Paolo Costa, *Microsoft Research*
Mallesham Dasari, *Northeastern University*
Quentin De Coninck, *Université de Mons*
Fahad Dogar, *Tufts University*
Mingkai Dong, *Shanghai Jiao Tong University*
Dong Du, *Shanghai Jiao Tong University*
Ram Durairajan, *University of Oregon*
Lars Eggert, *NetApp*
Shir Landau Feibish, *The Open University of Israel*
Anja Feldmann, *Max Planck Institute for Informatics*
Yashar Ganjali, *University of Toronto and Huawei Canada*
Jiaqi Gao, *Alibaba*
Kaihui Gao, *Zhongguancun Laboratory*
Aaron Gember-Jacobson, *Colgate University*
Soudeh Ghorbani, *Meta and Johns Hopkins University*
Brighten Godfrey, *University of Illinois at Urbana–Champaign and Broadcom*
Ayush Goel, *Meta*
Sergey Gorinsky, *IMDEA Networks Institute*
Prateesh Goyal, *Microsoft Research*
Indranil Gupta, *University of Illinois at Urbana–Champaign*
Hamed Haddadi, *Imperial College London and Brave Software*
Andreas Haeberlen, *University of Pennsylvania and Roblox*
Yotam Harchol, *NVIDIA*
Shaddi Hasan, *Virginia Tech*
Kurtis Heimerl, *University of Washington*
Oliver Hohlfeld, *University of Kassel*
Liu Hongqiang, *Alibaba*
Kevin Hsieh, *Microsoft Research*
Wenjun Hu, *Yale University*
Yu Hua, *Huazhong University of Science and Technology*
Ryan Huang, *University of Michigan*
Anand Iyer, *Georgia Institute of Technology*
Rishabh Iyer, *University of California, Berkeley*
Kyle Jamieson, *Princeton University*
Keon Jang, *Max Planck Institute for Software Systems and Rubrik*
Karthick Jayaraman, *Microsoft*
Junchen Jiang, *The University of Chicago*
Xin Jin, *Peking University*
Suraj Jog, *Microsoft Research*
Raj Joshi, *National University of Singapore*
Siva Kesava Reddy Kakarla, *Microsoft Research*
Mohan Kalkunte, *Broadcom*
Srikanth Kandula, *Microsoft*
Eric Keller, *University of Colorado Boulder*
Changhoon Kim, *Google*
Daehyeok Kim, *The University of Texas at Austin*
Gyuyeong Kim, *Sungshin Women's University*
Hyojoon Kim, *University of Virginia*
Dejan Kostic, *KTH Royal Institute of Technology*
Arvind Krishnamurthy, *University of Washington and Google*
Bhuvana Krishnaswamy, *University of Wisconsin—Madison*
Praveen Kumar, *Google*
Sam Kumar, *University of Washington and University of California, Los Angeles*
Fan Lai, *University of Illinois at Urbana–Champaign and Google*
Zeqi Lai, *Tsinghua University*
Andrea Lattuada, *VMware Research*
Wenke Lee, *Georgia Institute of Technology*
Huaicheng Li, *Virginia Tech*
Yuanjie Li, *Tsinghua University*
Chieh-Jan Mike Liang, *Microsoft Research*
Kate Ching-Ju Lin, *National Yang-Ming Chiao Tung University*
Alan Zaoxing Liu, *University of Maryland, College Park*
Guyue Liu, *Peking University*
Si Liu, *ETH Zürich*
Vincent Liu, *University of Pennsylvania*
Ioana Livadariu, *Simula Research Laboratory*
Chang Lou, *University of Virginia*
Songwu Lu, *University of California, Los Angeles*
Harsha V. Madhyastha, *University of Southern California*

Ajay Mahimkar, *AT&T*
Morley Mao, *University of Michigan and Google*
Zili Meng, *The Hong Kong University of Science and Technology*
Congcong Miao, *Tencent*
Rui Miao, *Meta*
Vishal Misra, *Columbia University*
Radhika Mittal, *University of Illinois at Urbana–Champaign*
Michael Mitzenmacher, *Harvard University*
Nitinder Mohan, *Technische Universität München*
Iqbal Mohomed, *Samsung AI Centre Toronto*
Masoud Moshref, *NVIDIA*
Kanthi Nagaraj, *Google*
Srinivas Narayana, *Rutgers University*
Arvind Narayanan, *AT&T*
Deepak Narayanan, *NVIDIA*
T. S. Eugene Ng, *Rice University*
Dave Oran, *Network Systems Research & Design*
Tian Pan, *Alibaba*
Aurojit Panda, *New York University*
Seo Jin Park, *University of Southern California*
Przemysław Pawełczak, *Delft University of Technology*
Chunyi Peng, *Purdue University*
Ben Pfaff, *Feldera*
Peter Pietzuch, *Imperial College London*
Boris Pismenny, *EPFL and NVIDIA*
Soujanya Ponnapalli, *University of California, Berkeley*
Feng Qian, *University of Minnesota Twin Cities*
Hang Qiu, *University of California, Riverside*
K. K. Ramakrishnan, *University of California, Riverside*
Fernando Ramos, *University of Lisbon*
Sanjay Rao, *Purdue University*
Robert Ricci, *University of Utah*
Anthony Rowe, *Carnegie Mellon University*
Nirupam Roy, *University of Maryland, College Park*
Raja Sambasivan, *Tufts University*
Nishanth Sastry, *University of Surrey*
Mariano Scazzariello, *KTH Royal Institute of Technology*
Michael Schapira, *The Hebrew University of Jerusalem*
Stefan Schmid, *Technische Universität Berlin*
Malte Schwarzkopf, *Brown University*
Colin Scott, *Google*
Muhammad Shahbaz, *Purdue University*
Yizhou Shan, *Huawei Cloud*
Longfei Shangguan, *University of Pittsburgh*
Vishal Shrivastav, *Purdue University*
Rachee Singh, *Cornell University*
Ramesh K. Sitaraman, *University of Massachusetts Amherst*
Gagan Somashekar, *Microsoft*
John Sonchack, *Princeton University*
Ravi Soundararajan, *VMware*
Peter Steenkiste, *Carnegie Mellon University*
Srikanth Sundaresan, *Facebook*
Srikanth Sundaresan, *Meta*
Alain Tchana, *Grenoble INP*
Renata Teixeira, *Netflix*
Alexandru Uta, *DFINITY*
Amin Vahdat, *Google*
Shay Vargaftik, *VMware Research*
Mythili Vutukuru, *Indian Institute of Technology*
Chenxi Wang, *Chinese Academy of Sciences*
Haoyu Wang, *Huazhong University of Science and Technology*
Jia Wang, *AT&T*
Jingxian Wang, *National University of Singapore*
Xiaoliang Wang, *Nanjing University*
Zeke Wang, *Zhejiang University*
Hassan M. G. Wassel, *Google*
Hakim Weatherspoon, *Cornell University and Exostellar, Inc.*
Michael Wei, *VMware*
Xingda Wei, *Shanghai Jiao Tong University*
Yiting Xia, *Max Planck Institute for Informatics*
Qiao Xiang, *Xiamen University*
Yaxiong Xie, *University at Buffalo*
Jie Xiong, *University of Massachusetts Amherst*
Erci Xu, *Alibaba*
Hong Xu, *The Chinese University of Hong Kong*
Tianyin Xu, *University of Illinois at Urbana–Champaign*
Francis Y. Yan, *Microsoft Research*
Xiaowei Yang, *Duke University*
Kasim Sinan Yildirim, *University of Trento*
Jihong Yu, *Beijing Institute of Technology*
Minchen Yu, *Chinese University of Hong Kong*
Yifan Yuan, *Intel Labs*
Yasir Zaki, *New York University Abu Dhabi*
Ennan Zhai, *Alibaba*
Hong Zhang, *University of Waterloo*
Junxue Zhang, *The Hong Kong University of Science and Technology*
Peng Zhang, *Xi'an Jiaotong University*
Ying Zhang, *Meta*
Yiran Zhang, *Beijing University of Posts and Telecommunications*
Zhi-Li Zhang, *University of Minnesota Twin Cities*
Renjie Zhao, *Johns Hopkins University*
Shizhen Zhao, *Shanghai Jiao Tong University*
Zhizhen Zhong, *Massachusetts Institute of Technology*
Diyu Zhou, *EPFL and Peking University*
Fang Zhou, *The Ohio State University*
Hang Zhu, *ByteDance Inc.*
Danyang Zhuo, *Duke University*
Marco Zimmerling, *Technische Universität Darmstadt*
Pengfei Zuo, *Huawei Cloud*

Poster Session Co-Chairs
Balakrishnan Chandrasekaran, *Vrije Universiteit Amsterdam*
Shir Landau Feibish, *The Open University of Israel*

Test of Time Awards Committee
Mahesh Balakrishnan, *Meta*
Jon Howell, *VMware Research*
James Mickens, *Harvard University*
Amar Phanishayee, *Meta*
George Porter, *University of California, San Diego*
Renata Teixeira, *Netflix*

Pre-Review Task Force
Vamsi Addanki, *Technische Universität Berlin*
Xinyi Chen, *University of Pennsylvania*
Changhan Ge, *The University of Texas at Austin*
Rashna Kumar, *Northwestern University*
Jennifer Lam, *Princeton University*
Jiaxin Lin, *The University of Texas at Austin*
Shinan Liu, *The University of Chicago*
Pooria Namyar, *University of Southern California*
Filip Rezabek, *Technische Universität München*
Alireza Sanaee, *Queen Mary University of London*
Ammar Tahir, *University of Illinois at Urbana–Champaign*
Bhavana V S, *Rutgers University*
Shu-Ting Wang, *University of California, San Diego*
Ertza Warraich, *Purdue University*
Emilia Weyulu, *Max Planck Institute for Informatics*
Chen-Yu Yen, *New York University*
Liangcheng Yu, *University of Pennsylvania*
Anlan Zhang, *University of Southern California*
Qingzhao Zhang, *University of Michigan*
Chenxingyu Zhao, *University of Washington*
Changgang Zheng, *University of Oxford*

Review Task Force
Gianni Antichi, *Politecnico di Milano and Queen Mary University of London*
Fabián Bustamante, *Northwestern University*
Matthew Caesar, *University of Illinois at Urbana–Champaign*
Marco Chiesa, *KTH Royal Institute of Technology*
Ram Durairajan, *University of Oregon*
Shir Landau Feibish, *The Open University of Israel*
Anja Feldmann, *Max Planck Institute for Informatics*
Yashar Ganjali, *University of Toronto and Huawei Canada*
Indranil Gupta, *University of Illinois at Urbana–Champaign*
Hamed Haddadi, *Imperial College London and Brave Software*
Wenjun Hu, *Yale University*
Srikanth Kandula, *Microsoft*
Arvind Krishnamurthy, *University of Washington and Google*
Vishal Misra, *Columbia University*
Ben Pfaff, *Feldera*
Michael Schapira, *The Hebrew University of Jerusalem*
Ramesh K. Sitaraman, *University of Massachusetts Amherst*
Peter Steenkiste, *Carnegie Mellon University*
Renata Teixeira, *Netflix*
Mythili Vutukuru, *Indian Institute of Technology*
Jia Wang, *AT&T*
Xiaowei Yang, *Duke University*
Yasir Zaki, *New York University Abu Dhabi*
Ying Zhang, *Meta*

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

Message from the NSDI '25 Program Co-Chairs

Welcome to the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25).

We are pleased to bring to you an exciting array of papers representing the best work in the networked systems research community. We hope that you will come away from NSDI invigorated, inspired and reconnected with the cross-cutting work across the broad community.

In keeping with past years, NSDI '25 received a record number of submissions, a total of 666, 11% more than last year on a growing spectrum of research topics. There were 265 papers in Spring and 401 papers in the Fall cycle. Of these, only papers that addressed both network and systems challenges, with network contributions spanning multiple layers of the OSI stack were considered to be in scope (https://www.usenix.org/conference/nsdi25/call-for-papers). Out of the 565 papers deemed in scope, 83 were accepted into the program. The final program has a 12.5% acceptance rate, where 14.7% of the papers in scope were accepted. It is our hope to see the acceptance rate grow as the community adjusts to the scope changes in the future years.

To ensure that the submissions receive 3-6 reviews with adequate expertise, we assembled a Program Committee of record size, consisting of 198 members from 118 institutions across 20 countries. Of these, 87 committee members were chosen from 177 self-nominations; these included many first-time committee members. Our goal with PC composition was to balance many key factors, especially, expertise and seniority across NSDI topics. With this carefully chosen large PC, we were able to keep the reviewing load to at most 18 for the entire year; each member reviewed at most 6 papers in Spring and 12 papers in the Fall cycle.

Papers with at least one weak accept advanced from R1 to R2. In Spring, 87 of the 200 papers in R1 made it to R2 and in Fall, 201 of the 365 R1 papers made it to R2. More papers advanced in the Fall cycle because many papers lacked the necessary reviews to decide them. Overall, reviewers submitted a total of 2805 reviews, 868 in Spring and 1937 in Fall. Many reviewers helped out with last minute review requests, especially in the Fall cycle, to fill review gaps and keep us on time.

This year we introduced two additional committees to handle NSDI's growing scale.

To maintain NSDI's high standard of reviewing and to guide first time PC members, we recruited a Review Task Force of 24 experienced committee members who were tasked with reviewing paper reviews in lieu of a smaller reviewing load.

To ensure that submissions met the criteria in the CFP, such as anonymization, we formed the Pre-Review Task Force composed of 21 senior PhD students. These students were recruited from 81 nominations we received to an open call, based on their authorship experience in top-tier systems conferences. Both chairs reviewed every paper that the Pre-Review Task Force flagged to eliminate false violations. The Pre-Review Task force gets the credit for admitting very few CFP violations into the review pool. We hope that future program chairs continue to benefit from students' experience while providing them with valuable insight into PC operation. We also hope future chairs enhance the experience of students beyond what we did in NSDI '25 and provide them with exposure to members of the PC.

The SC has had the idea of limiting NSDI's scope for a few years now. We undertook the responsibility of defining what this scope should be and enforcing it in NSDI '25. Together with the SC, we determined that the desired scope of NSDI is defined by contributions that address both network and system challenges. Since the SC required this definition to be tested out before impacting the CFP, it was decided that the chairs would be responsible for flagging submissions that were out of scope to the SC by anonymizing them, and these would be cross-checked by the SC. This tight coupling between the SC and the chairs worked well in Spring, but fell apart in Fall when the submissions grew by 1.5x. The direct consequence of the chairs and SC assuming the burden of this tight coupling was the untimely delivery of scope rejections. We are happy to see our feedback to the SC reflected in a clear scope definition in NSDI '26 CFP and a more streamlined mechanism to deliver scope rejections.

Reflecting on our experiences, we have two key pieces of advice for future chairs as they consider scope implementations in the face of NSDI's growing submissions. First, to carefully consider how they split responsibilities with the Program Committee as they consider new ideas to implement. Second, to scale this partnership by communicating decisions and guidance to the community in a timely manner, ideally well before submission deadlines.

Our gratitude goes out to Usenix, the Steering Committee, past chairs, the Program Committee, the Review and Pre-Review Task Force, and the Award committee members for making NSDI '25 possible. Special thanks to Renata Teixeira for her multiple roles as a counsel, the co-chair for chair-conflict papers, Review Task Force and Awards Committee member; and to Jon Howell, Jay Lorch, Gianni Antichi and Sujata Banerjee for their counsel. Together, they contributed immensely towards making chairing a pleasure.

Theophilus A. Benson, *Carnegie Mellon University*
Radhika Niranjan Mysore, *VMware Research Group*
NSDI '25 Program Co-Chairs

22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)

April 28–30, 2025
Philadelphia, PA

Monday, April 28

Data Centers Queuing and Routing

PRED: Performance-oriented Random Early Detection for Consistently Stable Performance in Datacenters 1
Xinle Du, *Huawei Technologies;* Tong Li, *Renmin University of China;* Guangmeng Zhou, Zhuotao Liu, Hanlin Huang, and Xiangyu Gao, *Tsinghua University;* Mowei Wang and Kun Tan, *Huawei Technologies;* Ke Xu, *Tsinghua University*

Rajomon: Decentralized and Coordinated Overload Control for Latency-Sensitive Microservices 21
Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, and Justin Qiu, *University of Pennsylvania;* Henri Maxime Demoulin, *DBOS, Inc;* Konstantinos Kallas, *University of California, Los Angeles;* Benjamin C. Lee, *University of Pennsylvania*

Learnings from Deploying Network QoS Alignment to Application Priorities for Storage Services 37
Matthew Buckley and Parsa Pazhooheshy, *Google and University of Toronto;* Z. Morley Mao, Nandita Dukkipati, Hamid Hajabdolali Bazzaz, Priyaranjan Jha, Yingjie Bi, and Steve Middlekauff, *Google;* Yashar Ganjali, *University of Toronto*

DiSC: Backpressure Mitigation In Multi-tier Applications With Distributed Shared Connection 55
Brice Ekane and Djob Mvondo, *Univ. Rennes, Inria, CNRS, IRISA, France;* Renaud Lachaize, *Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France;* Yérom-David Bromberg, *Univ. Rennes, Inria, CNRS, IRISA, France;* Alain Tchana, *Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France;* Daniel Hagimont, *IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3 Toulouse, France*

Data Plane Programmability 1

Enabling Silent Telemetry Data Transmission with InvisiFlow 71
Yinda Zhang, *University of Pennsylvania;* Liangcheng Yu, *University of Pennsylvania and Microsoft Research;* Gianni Antichi, *Politecnico di Milano and Queen Mary University of London;* Ran Ben Basat, *University College London;* Vincent Liu, *University of Pennsylvania*

Unlocking ECMP Programmability for Precise Traffic Control 87
Yadong Liu, *Tencent;* Yunming Xiao, *University of Michigan;* Xuan Zhang, Weizhen Dang, Huihui Liu, Xiang Li, and Zekun He, *Tencent;* Jilong Wang, *Tsinghua University;* Aleksandar Kuzmanovic, *Northwestern University;* Ang Chen, *University of Michigan;* Congcong Miao, *Tencent*

Enabling Portable and High-Performance SmartNIC Programs with Alkali 107
Jiaxin Lin, *UT Austin;* Zhiyuan Guo, *UCSD;* Mihir Shah, *NVIDIA;* Tao Ji, *Microsoft;* Yiying Zhang, *UCSD;* Daehyeok Kim and Aditya Akella, *UT Austin*

Scaling IP Lookup to Large Databases using the CRAM Lens 127
Robert Chang and Pradeep Dogga, *University of California, Los Angeles;* Andy Fingerhut, *Cisco Systems;* Victor Rios and George Varghese, *University of California, Los Angeles*

Data Center Resource Scheduling

Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing 147
Zhenyuan Ruan, *MIT CSAIL;* Shihang Li, *Brown University;* Kaiyan Fan, *MIT CSAIL;* Seo Jin Park, *USC;* Marcos K. Aguilera, *VMware Research by Broadcom;* Adam Belay, *MIT CSAIL;* Malte Schwarzkopf, *Brown University*

Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs 167
Quanxi Li, Hong Huang, Ying Liu, and Yanwen Xia, *Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Jie Zhang, *Peking University;* Mosong Zhou, *Huawei Cloud;* Xiaobing Feng and Huimin Cui, *Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Quan Chen, *Shanghai Jiao Tong University;* Yizhou Shan, *Huawei Cloud;* Chenxi Wang, *Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences*

Making Serverless Pay-For-Use a Reality with Leopard **189**
Tingjia Cao, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Tyler Caraza-Harter, *University of Wisconsin-Madison*

GRANNY: Granular Management of Compute-Intensive Applications in the Cloud **205**
Carlos Segarra, Simon Shillaker, Guo Li, and Eleftheria Mappoura, *Imperial College London;* Rodrigo Bruno, *INESC-ID, Instituto Superior Técnico, University of Lisbon;* Lluís Vilanova and Peter Pietzuch, *Imperial College London*

Verification 1

On Temporal Verification of Stateful P4 Programs **219**
Delong Zhang, Chong Ye, and Fei He, *School of Software, BNRist, Tsinghua University, Beijing 100084, China; Key Laboratory for Information System Security, MoE, China*

NDD: A Decision Diagram for Network Verification **237**
Zechun Li, Peng Zhang, and Yichi Zhang, *Xi'an Jiaotong University;* Hongkun Yang, *Google*

Smart Casual Verification of the Confidential Consortium Framework **259**
Heidi Howard, Markus A. Kuppe, Edward Ashton, and Amaury Chamayou, *Azure Research, Microsoft;* Natacha Crooks, *Azure Research, Microsoft, and UC Berkeley*

VEP: A Two-stage Verification Toolchain for Full eBPF Programmability **277**
Xiwei Wu, Yueyang Feng, Tianyi Huang, Xiaoyang Lu, Shengkai Lin, Lihan Xie, Shizhen Zhao, and Qinxiang Cao, *Shanghai Jiao Tong University*

Failure and Diagnosis

MeshTest: End-to-End Testing for Service Mesh Traffic Management **301**
Naiqian Zheng, Tianshuo Qiao, Xuanzhe Liu, and Xin Jin, *Peking University*

Preventing Network Bottlenecks: Accelerating Datacenter Services with Hotspot-Aware Placement for Compute and Storage **317**
Hamid Hajabdolali Bazzaz, Yingjie Bi, and Weiwu Pang, *Google;* Minlan Yu, *Harvard University;* Ramesh Govindan, *University of Southern California;* Neal Cardwell, Nandita Dukkipati, Meng-Jung Tsai, Chris DeForeest, and Yuxue Jin, *Google;* Charles Carver, *Columbia University;* Jan Kopański, Liqun Cheng, and Amin Vahdat, *Google*

Enhancing Network Failure Mitigation with Performance-Aware Ranking **335**
Pooria Namyar and Arvin Ghavidel, *University of Southern California;* Daniel Crankshaw, Daniel S. Berger, Kevin Hsieh, and Srikanth Kandula, *Microsoft;* Ramesh Govindan, *University of Southern California;* Behnaz Arzani, *Microsoft*

One-Size-Fits-None: Understanding and Enhancing Slow-Fault Tolerance in Modern Distributed Systems **359**
Ruiming Lu, *University of Michigan and Shanghai Jiao Tong University;* Yunchi Lu and Yuxuan Jiang, *University of Michigan;* Guangtao Xue, *Shanghai Jiao Tong University;* Peng Huang, *University of Michigan*

All Things Transport

Pyrrha: Congestion-Root-Based Flow Control to Eliminate Head-of-Line Blocking in Datacenter **379**
Kexin Liu, Zhaochen Zhang, Chang Liu, and Yizhi Wang, *Nanjing University;* Vamsi Addanki and Stefan Schmid, *TU Berlin;* Qingyue Wang, Wei Chen, Xiaoliang Wang, and Jiaqi Zheng, *Nanjing University;* Wenhao Sun, Tao Wu, Ke Meng, Fei Chen, Weiguang Wang, and Bingyang Liu, *Huawei, China;* Wanchun Dou, Guihai Chen, and Chen Tian, *Nanjing University*

eTran: Extensible Kernel Transport with eBPF **407**
Zhongjie Chen, *Tsinghua University;* Qingkai Meng, *Nanjing University;* ChonLam Lao, *Harvard University;* Yifan Liu and Fengyuan Ren, *Tsinghua University;* Minlan Yu, *Harvard University;* Yang Zhou, *UC Berkeley and UC Davis*

White-Boxing RDMA with Packet-Granular Software Control **427**
Chenxingyu Zhao and Jaehong Min, *University of Washington;* Ming Liu, *University of Wisconsin-Madison;* Arvind Krishnamurthy, *University of Washington*

SIRD: A Sender-Informed, Receiver-Driven Datacenter Transport Protocol **451**
Konstantinos Prasopoulos, *EPFL;* Ryan Kosta, *UCSD;* Edouard Bugnion, *EPFL;* Marios Kogias, *Imperial College London*

LLM Training and Resilience

Accelerating Design Space Exploration for LLM Training Systems with Multi-experiment Parallel Simulation . . . 473
Fei Gui, *Tsinghua University, BNRist, and Tsinghua Shenzhen International Graduate School;* Kaihui Gao and Li Chen, *Zhongguancun Laboratory;* Dan Li, *Tsinghua University;* Vincent Liu, *University of Pennsylvania;* Ran Zhang and Hongbing Yang, *Zhongguancun Laboratory;* Dian Xiong, *Tsinghua University*

Optimizing RLHF Training for Large Language Models with Stage Fusion 489
Yinmin Zhong, Zili Zhang, Bingyang Wu, and Shengyu Liu, *School of Computer Science, Peking University;* Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, and Yibo Zhu, *StepFun;* Xin Jin, *School of Computer Science, Peking University*

Minder: Faulty Machine Detection for Large-scale Distributed Model Training 505
Yangtao Deng, *Tsinghua University;* Xiang Shi and Zhuo Jiang, *ByteDance;* Xingjian Zhang, *Tsinghua University;* Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, and Gaohong Liu, *ByteDance;* Fuliang Li, *Northeastern University;* Shuguang Wang, Haibin Lin, and Jianxi Ye, *ByteDance;* Minlan Yu, *Harvard University*

***Holmes*: Localizing Irregularities in LLM Training with Mega-scale GPU Clusters 523**
Zhiyi Yao and Pengbo Hu, *Fudan University and Tencent;* Congcong Miao and Xuya Jia, *Tencent;* Zuning Liang and Yuedong Xu, *Fudan University;* Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, Zekun He, Yachen Wang, and Xianneng Zou, *Tencent;* Junchen Jiang, *University of Chicago*

SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale Large Language Model Training with Scalability and Precision 541
Xizheng Wang, *Alibaba Cloud and Tsinghua University;* Qingxu Li, Yichi Xu, and Gang Lu, *Alibaba Cloud;* Dan Li, *Tsinghua University;* Li Chen, *Zhongguancun Laboratory;* Heyang Zhou, *Alibaba Cloud;* Linkang Zheng, *Alibaba Cloud and South China University of Technology;* Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu, *Alibaba Cloud*

ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development 559
Borui Wan, *The University of Hong Kong;* Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, and Xin Liu, *ByteDance Inc.;* Chuan Wu, *The University of Hong Kong*

Video and Cloud Gaming

Mowgli: Passively Learned Rate Control for Real-Time Video 579
Neil Agarwal and Rui Pan, *Princeton University;* Francis Y. Yan, *University of Illinois Urbana-Champaign;* Ravi Netravali, *Princeton University*

Dissecting and Streamlining the Interactive Loop of Mobile Cloud Gaming. 595
Yang Li, Jiaxing Qiu, Hongyi Wang, and Zhenhua Li, *Tsinghua University;* Feng Qian, *University of Southern California;* Jing Yang, *Tsinghua University;* Hao Lin, *Tsinghua University and UIUC;* Yunhao Liu, *Tsinghua University;* Bo Xiao and Xiaokang Qin, *Ant Group;* Tianyin Xu, *UIUC*

Region-based Content Enhancement for Efficient Video Analytics at the Edge 613
Weijun Wang, *Institute for AI Industry Research (AIR), Tsinghua University;* Liang Mi, Shaowei Cen, and Haipeng Dai, *State Key Laboratory for Novel Software Technology, Nanjing University;* Yuanchun Li, *Institute for AI Industry Research (AIR), Tsinghua University;* Xiaoming Fu, *University of Göttingen;* Yunxin Liu, *Institute for AI Industry Research (AIR), Tsinghua University*

Tooth: Toward Optimal Balance of Video QoE and Redundancy Cost by Fine-Grained FEC in Cloud Gaming Streaming 635
Congkai An, Huanhuan Zhang, Shibo Wang, Jingyang Kang, Anfu Zhou, Liang Liu and Huadong Ma, *Beijing University of Posts and Telecommunications;* Zili Meng, *Hong Kong University of Science and Technology;* Delei Ma, Yusheng Dong, and Xiaogang Lei, *Well-Link Times Inc.*

AsTree: An Audio Subscription Architecture Enabling Massive-Scale Multi-Party Conferencing 653
Tong Meng, Wenfeng Li, Chao Yuan, Changqing Yan, and Le Zhang, *ByteDance Inc.*

Tuesday, April 29

Infra For ML

AutoCCL: Automated Collective Communication Tuning for Accelerating Distributed and Parallel DNN Training 667
Guanbin Xu, Zhihao Le, Yinhe Chen, Zhiqi Lin, and Zewen Jin, *University of Science and Technology of China;* Youshan Miao, *Microsoft Research;* Cheng Li, *University of Science and Technology of China; Anhui Province Key Laboratory of Biomedical Imaging and Intelligent Processing; Institute of Artificial Intelligence, Hefei Comprehensive National Science Center*

OptiReduce: Resilient and Tail-Optimal AllReduce for Distributed Deep Learning in the Cloud 685
Ertza Warraich, *Purdue University;* Omer Shabtai and Khalid Manaa, *Nvidia;* Shay Vargaftik, *VMware Research;* Yonatan Piasetzky and Matty Kadosh, *Nvidia;* Lalith Suresh, *Feldera;* Muhammad Shahbaz, *University of Michigan*

Efficient Direct-Connect Topologies for Collective Communications 705
Liangyu Zhao, *University of Washington;* Siddharth Pal, *RTX BBN Technologies;* Tapan Chugh, *University of Washington;* Weiyang Wang, *MIT CSAIL;* Jason Fantl, Prithwish Basu, and Joud Khoury, *RTX BBN Technologies;* Arvind Krishnamurthy, *University of Washington*

SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads. 739
Alind Khare and Dhruv Garg, *Georgia Tech;* Sukrit Kalra, *UC Berkeley;* Snigdha Grandhi, *Adobe;* Ion Stoica, *UC Berkeley;* Alexey Tumanov, *Georgia Tech*

Fast Scalable Consensus

Pineapple: Unifying Multi-Paxos and Atomic Shared Registers 759
Tigran Bantikyan, *Northwestern;* Jonathan Zarnstorff, *unaffiliated;* Te-Yen Chou, *CMU;* Lewis Tseng, *UMass Lowell;* Roberto Palmieri, *Lehigh University*

Ladder: A Convergence-based Structured DAG Blockchain for High Throughput and Low Latency 779
Dengcheng Hu, Jianrong Wang, Xiulong Liu, and Hao Xu, *Tianjin University;* Xujing Wu, *Jd.Com, Inc;* Muhammad Shahzad, *North Carolina State University;* Guyue Liu, *Peking University;* Keqiu Li, *Tianjin University*

Vegeta: Enabling Parallel Smart Contract Execution in Leaderless Blockchains 795
Tianjing Xu and Yongqi Zhong, *Shanghai Jiao Tong University;* Yiming Zhang, *Shanghai Jiao Tong University and Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3;* Ruofan Xiong, *Xiamen University;* Jingjing Zhang, *Fudan University;* Guangtao Xue and Shengyun Liu, *Shanghai Jiao Tong University and Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3*

Shoal++: High Throughput DAG BFT Can Be Fast and Robust! 813
Balaji Arun and Zekun Li, *Aptos Labs;* Florian Suri-Payer, *Cornell University;* Sourav Das, *UIUC;* Alexander Spiegelman, *Aptos Labs*

Operational Experiences

Learning Production-Optimized Congestion Control Selection for Alibaba Cloud CDN 827
Xuan Zeng, *Alibaba Cloud;* Haoran Xu, *Sun Yat-sen University;* Chen Chen and Xumiao Zhang, *Alibaba Cloud;* Xiaoxi Zhang and Xu Chen, *Sun Yat-sen University;* Guihai Chen, *Nanjing University;* Yubing Qiu, Yiping Zhang, Chong Hao, and Ennan Zhai, *Alibaba Cloud*

GPU-Disaggregated Serving for Deep Learning Recommendation Models at Scale 847
Lingyun Yang, *Hong Kong University of Science and Technology;* Yongchen Wang and Yinghao Yu, *Alibaba Group;* Qizhen Weng, *Hong Kong University of Science and Technology;* Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, and Lin Qu, *Alibaba Group;* Wei Wang, *Hong Kong University of Science and Technology*

Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production. 865
Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, and Dennis Cai, *Alibaba Cloud*

PAPAYA Federated Analytics Stack: Engineering Privacy, Scalability and Practicality 883
Harish Srinivas, Graham Cormode, Mehrdad Honarkhah, Samuel Lurye, Jonathan Hehir, Lunwen He, George Hong, Ahmed Magdy, Dzmitry Huba, Kaikai Wang, Shen Guo, and Shoubhik Bhattacharya, *Meta*

Middleboxes

HA/TCP: A Reliable and Scalable Framework for TCP Network Functions 899
Haoyu Gu, Ali José Mashtizadeh, and Bernard Wong, *University of Waterloo*

High-level Programming for Application Networks 915
Xiangfeng Zhu, Yuyao Wang, Banruo Liu, Yongtong Wu, and Nikola Bojanic, *University of Washington;* Jingrong Chen, *Duke University;* Gilbert Louis Bernstein and Arvind Krishnamurthy, *University of Washington;* Sam Kumar, *University of Washington and UCLA;* Ratul Mahajan, *University of Washington;* Danyang Zhuo, *Duke University*

State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing 937
Qiongwen Xu, *Rutgers University;* Sebastiano Miano, *Politecnico di Milano;* Xiangyu Gao and Tao Wang, *New York University;* Adithya Murugadass and Songyuan Zhang, *Rutgers University;* Anirudh Sivaraman, *New York University;* Gianni Antichi, *Queen Mary University of London and Politecnico di Milano;* Srinivas Narayana, *Rutgers University*

MTP: Transport for In-Network Computing 959
Tao Ji, *UT Austin;* Rohan Vardekar and Balajee Vamanan, *University of Illinois Chicago;* Brent E. Stephens, *Google and University of Utah;* Aditya Akella, *UT Austin*

Rethinking Data Center Efficiency

ONCache: A Cache-Based Low-Overhead Container Overlay Network 979
Shengkai Lin, Shizhen Zhao, Peirui Cao, and Xinchi Han, *Shanghai Jiao Tong University;* Quan Tian, Wenfeng Liu, Qi Wu, and Donghai Han, *Broadcom;* Xinbing Wang, *Shanghai Jiao Tong University*

GREEN: Carbon-efficient Resource Scheduling for Machine Learning Clusters 999
Kaiqiang Xu and Decang Sun, *iSING Lab, Hong Kong University of Science and Technology;* Han Tian, *USTC;* Junxue Zhang and Kai Chen, *iSING Lab, Hong Kong University of Science and Technology*

The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling 1015
Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout, *UC San Diego*

Securing Public Cloud Networks with Efficient Role-based Micro-Segmentation 1033
Sathiya Kumaran Mani and Kevin Hsieh, *Microsoft;* Santiago Segarra, *Rice University;* Ranveer Chandra, *Microsoft;* Yajie Zhou, *University of Maryland;* Srikanth Kandula, *Microsoft*

RDMA

Mitigating Scalability Walls of RDMA-based Container Networks 1049
Wei Liu, *Tsinghua University and Alibaba Cloud;* Kun Qian, *Alibaba Cloud;* Zhenhua Li, *Tsinghua University;* Feng Qian, *University of Southern California;* Tianyin Xu, *UIUC;* Yunhao Liu, *Tsinghua University;* Yu Guan, Shuhong Zhu, Hongfei Xu, Lanlan Xi, Chao Qin, and Ennan Zhai, *Alibaba Cloud*

Eden: Developer-Friendly Application-Integrated Far Memory 1067
Anil Yelam, Stewart Grant, and Saarth Deshpande, *UC San Diego;* Nadav Amit, *Technion, Israel Institute of Technology;* Radhika Niranjan Mysore, *VMware Research Group;* Amy Ousterhout, *UC San Diego;* Marcos K. Aguilera, *VMware Research Group;* Alex C. Snoeren, *UC San Diego*

Achieving Wire-Latency Storage Systems by Exploiting Hardware ACKs 1085
Qing Wang, Jiwu Shu, Jing Wang, and Yuhao Zhang, *Tsinghua University*

ODRP: On-Demand Remote Paging with Programmable RDMA 1101
Zixuan Wang, Xingda Wei, Jinyu Gu, Hongrui Xie, Rong Chen, and Haibo Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*

Storage

Understanding and Profiling NVMe-over-TCP Using ntprof **1117**
Yuyuan Kang and Ming Liu, *University of Wisconsin-Madison*

Building an Elastic Block Storage over EBOFs Using Shadow Views **1137**
Sheng Jiang, *Carnegie Mellon University;* Ming Liu, *University of Wisconsin-Madison*

Pushing the Limits of In-Network Caching for Key-Value Stores **1155**
Gyuyeong Kim, *Sungshin Women's University*

Cellular and Wireless

CellReplay: Towards accurate record-and-replay for cellular networks **1169**
William Sentosa, *University of Illinois Urbana-Champaign;* Balakrishnan Chandrasekaran, *VU Amsterdam;* P. Brighten Godfrey, *University of Illinois Urbana-Champaign and Broadcom;* Haitham Hassanieh, *EPFL*

Large Network UWB Localization: Algorithms and Implementation **1187**
Nakul Garg and Irtaza Shahid, *University of Maryland, College Park;* Ramanujan K Sheshadri, *Nokia Bell Labs;* Karthikeyan Sundaresan, *Georgia Institute of Technology;* Nirupam Roy, *University of Maryland, College Park*

Towards Energy Efficient 5G vRAN Servers **1205**
Anuj Kalia, *Microsoft;* Nikita Lazarev, *MIT;* Leyang Xue, *University of Edinburgh;* Xenofon Foukas and Bozidar Radunovic, *Microsoft;* Francis Y. Yan, *Microsoft Research and UIUC*

Building Massive MIMO Baseband Processing on a Single-Node Supercomputer **1221**
Xincheng Xie, Wentao Hou, Zerui Guo, and Ming Liu, *University of Wisconsin-Madison*

Efficient Multi-WAN Transport for 5G with Otter **1243**
Mary Hogan, *Oberlin College;* Gerry Wan, *Google;* Yiming Qiu, *University of Michigan;* Sharad Agarwal and Ryan Beckett, *Microsoft;* Rachee Singh, *Cornell University;* Paramvir Bahl, *Microsoft*

Wednesday, April 30

Verification 2

Verifying maximum link loads in a changing world **1269**
Tibor Schneider, *ETH Zürich;* Stefano Vissicchio, *University College London;* Laurent Vanbever, *ETH Zürich*

A Layered Formal Methods Approach to Answering Queue-related Queries **1289**
Divya Raghunathan, Maria Apostolaki, and Aarti Gupta, *Princeton University*

Runtime Protocol Refinement Checking for Distributed Protocol Implementations **1305**
Ding Ding, Zhanghan Wang, Jinyang Li, and Aurojit Panda, *NYU*

CEGS: Configuration Example Generalizing Synthesizer **1327**
Jianmin Liu, *Tsinghua University;* Li Chen, *Zhongguancun Laboratory;* Dan Li, *Tsinghua University;* Yukai Miao, *Zhongguancun Laboratory*

Security

Suppressing BGP Zombies with Route Status Transparency **1349**
Yosef Edery Anahory, *The Hebrew University of Jerusalem;* Jie Kong, Nicholas Scaglione, and Justin Furuness, *University of Connecticut;* Hemi Leibowitz, *The College of Management Academic Studies;* Amir Herzberg and Bing Wang, *University of Connecticut;* Yossi Gilad, *The Hebrew University of Jerusalem*

ValidaTor: Domain Validation over Tor **1367**
Jens Frieß, *National Research Center for Applied Cybersecurity ATHENE and Technische Universität Darmstadt;* Haya Schulmann, *National Research Center for Applied Cybersecurity ATHENE and Goethe-Universität Frankfurt;* Michael Waidner, *National Research Center for Applied Cybersecurity ATHENE and Technische Universität Darmstadt*

From Address Blocks to Authorized Prefixes: Redesigning RPKI ROV with a Hierarchical Hashing Scheme for Fast and Memory-Efficient Validation 1381
Zedong Ni, *Computer Network Information Center, Chinese Academy of Sciences; and School of Cyber Science & Engineering, Southeast University;* Yinbo Xu, Hui Zou, and Yanbiao Li, *Computer Network Information Center, Chinese Academy of Sciences; and University of Chinese Academy of Sciences;* Guang Cheng, *School of Cyber Science & Engineering, Southeast University; and Purple Mountain Laboratories;* Gaogang Xie, *Computer Network Information Center, Chinese Academy of Sciences; and University of Chinese Academy of Sciences*

PreAcher: Secure and Practical Password Pre-Authentication by Content Delivery Networks. 1399
Shihan Lin, *Duke University;* Suting Chen, *Northwestern University;* Yunming Xiao, *University of Michigan;* Yanqi Gu, *University of California, Irvine;* Aleksandar Kuzmanovic, *Northwestern University;* Xiaowei Yang, *Duke University*

Data Plane Programmability 2

ClubHeap: A High-Speed and Scalable Priority Queue for Programmable Packet Scheduling 1421
Zhikang Chen, *Tsinghua University;* Haoyu Song, *Futurewei Technologies;* Zhiyu Zhang and Yang Xu, *Fudan University;* Bin Liu, *Tsinghua University*

Self-Clocked Round-Robin Packet Scheduling 1437
Erfan Sharafzadeh, *Johns Hopkins University and Hewlett Packard Labs;* Raymond Matson, *University of California Riverside;* Jean Tourrilhes and Puneet Sharma, *Hewlett Packard Labs;* Soudeh Ghorbani, *Johns Hopkins University and Meta*

***Everything Matters* in Programmable Packet Scheduling. 1467**
Albert Gran Alcoz, *ETH Zürich;* Balázs Vass, *BME-TMIT;* Pooria Namyar, *USC;* Behnaz Arzani, *Microsoft Research;* Gábor Rétvári, *BME-TMIT;* Laurent Vanbever, *ETH Zürich*

When P4 Meets Run-to-completion Architecture 1487
Hao Zheng, *State Key Laboratory for Novel Software Technology, Nanjing University, China;* Xin Yan, *Huawei, China;* Wenbo Li, Jiaqi Zheng, and Xiaoliang Wang, *State Key Laboratory for Novel Software Technology, Nanjing University, China;* Qingqing Zhao, Luyou He, Xiaofei Lai, Feng Gao, and Fuguang Huang, *Huawei, China;* Wanchun Dou, Guihai Chen, and Chen Tian, *State Key Laboratory for Novel Software Technology, Nanjing University, China*

ML for Networks

Mutant: Learning Congestion Control from Existing Protocols via Online Reinforcement Learning 1507
Lorenzo Pappone, *Computer Science Department, Saint Louis University;* Alessio Sacco, *DAUIN, Politecnico di Torino;* Flavio Esposito, *Computer Science Department, Saint Louis University*

CATO: End-to-End Optimization of ML-Based Traffic Analysis Pipelines 1523
Gerry Wan, *Stanford University;* Shinan Liu, *University of Chicago;* Francesco Bronzino, *ENS Lyon;* Nick Feamster, *University of Chicago;* Zakir Durumeric, *Stanford University*

Resolving Packets from Counters: Enabling Multi-scale Network Traffic Super Resolution via Composable Large Traffic Model 1541
Xizheng Wang, *Tsinghua University and Zhongguancun Laboratory;* Libin Liu and Li Chen, *Zhongguancun Laboratory;* Dan Li, *Tsinghua University;* Yukai Miao and Yu Bai, *Zhongguancun Laboratory*

BFTBrain: Adaptive BFT Consensus with Reinforcement Learning 1563
Chenyuan Wu and Haoyun Qin, *University of Pennsylvania;* Mohammad Javad Amiri, *Stony Brook University;* Boon Thau Loo, *University of Pennsylvania;* Dahlia Malkhi, *UC Santa Barbara;* Ryan Marcus, *University of Pennsylvania*

PRED: Performance-oriented Random Early Detection for Consistently Stable Performance in Datacenters

Xinle Du[†] Tong Li[‡,*] Guangmeng Zhou[§] Zhuotao Liu[§] Hanlin Huang[§]
Xiangyu Gao[§] Mowei Wang[†] Kun Tan[†] Ke Xu[§,*]
[†]*Huawei Technologies* [‡]*Renmin University of China* [§]*Tsinghua University*

Abstract

For decades, Random Early Detection (RED) has been integrated into datacenter switches as a fundamental Active Queue Management (AQM). Accurate configuration of RED parameters is crucial to achieving high throughput and low latency. However, due to the highly dynamic nature of workloads in datacenter networks, maintaining consistently high performance with statically configured RED thresholds poses a challenge. Prior art applies reinforcement learning to predict proper thresholds, but their real-world deployment has been hindered by poor tail performance caused by instability. In this paper, we propose PRED, a novel system that enables automatic and stable RED parameter adjustment in response to traffic dynamics. PRED uses two loosely coupled systems, Flow Concurrent Stabilizer (FCS) and Queue Length Adjuster (QLA), to overcome the challenges of dynamically setting RED parameters to adapt to the ever-changing traffic pattern. We perform extensive evaluations on our physical testbed and large-scale simulations. The results demonstrate that PRED can keep up with the real-time network dynamics generated by realistic workloads. For instance, compared with the static-threshold-based methods, PRED keeps 66% lower switch queue length and obtains up to 80% lower Flow Completion Time (FCT). Compared with the state-of-the-art learning-based method, PRED reduces the tail FCT by 34%.

1 Introduction

Datacenters host a variety of services with distinct networking preferences. For example, storage [1] and data mining [2] require high throughput, while web search [3] and machine learning [4] services require low latency. To meet these application requirements, a lot of innovative congestion control algorithms (e.g., DCTCP [3], DCQCN [5], TIMELY [6], HPCC [7], Swift [8], BFC [9]) have been proposed to reduce the delay caused by in-network queuing.

These new congestion controls demonstrate tremendous improvements; however, deployment is a challenging issue. For instance, it takes Google a year to release a new kernel to support new network functions [8, 10]. Many congestion controls need to be deployed on hardware Network Interface Cards (NICs) (e.g., HPCC [7]) or require new switches to support them (e.g., BFC [9]), which cannot be deployed in heterogeneous datacenters made up of legacy devices [11]. In some particular scenarios, such as multi-tenant datacenter networks, it is also difficult to modify the network protocol stack at the end host [12, 13].

To compensate for the host-based new congestion control protocols, the community also explores more accurate in-network congestion signals [14–17]. For instance, Explicit Congestion Notification (ECN) [18] is widely deployed in datacenter networks as the new congestion signal besides packet losses. Random Early Detection (RED) [19], the algorithm that controls how to discard packets or how to mark packets for ECN, has also been integrated into switches as a basic Active Queue Management (AQM) function. Many production datacenter transport protocols rely on ECN and RED [20–22], such as DCTCP and DCQCN.

Broadly speaking, the prior work on addressing RED/ECN's shortcomings can be classified as either new AQMs (e.g., TCD [14]), case-based RED/ECN (e.g., ECNsharp [15], BCC [16]), or learning-based RED/ECN (e.g., ACC [17]). The new AQM relies on a new switch design, thus requiring a long product release cycle [17]. The case-based arts (e.g., ECNsharp, BCC) realize that RED/ECN with a fixed threshold is ineffective in datacenters and therefore propose to consider multiple cases during threshold adjustment. However, they only recognize a limited number of cases and still use fixed thresholds for each case (see details in § 6.3.2). Prediction model-based ACC [17] leverages the deep reinforcement learning (DRL) [23] to adjust the RED settings dynamically in reaction to network congestion. However, it might make the adjustment of the RED unstable, resulting in sharp changes in queue length and poor tail performance (see details in § 6.3.3).

Thus from a philosophical standpoint, it is worth asking: Why can't static RED adapt to dynamic traffic? Can the prob-

*Tong Li and Ke Xu are the corresponding authors.

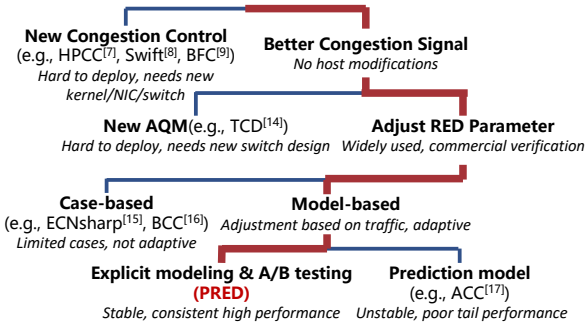


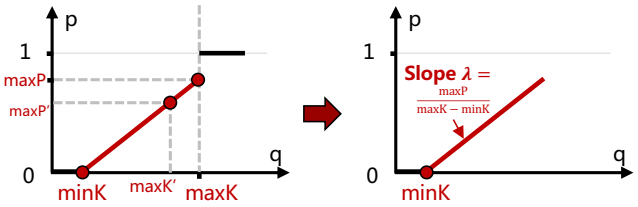
Figure 1: Design space in Datacenters.

lem be solved simply by dynamically adjusting the threshold of the already widely used RED without modifying the algorithm itself? Is there a stable way to adjust the RED threshold? In this paper, we seek to answer these questions with a new scheme PRED (Performance-oriented RED).

Figure 1 surveys the design space for congestion control and places PRED in context by following the thick red lines through the design tree. At the highest level, PRED is an AQM implemented on the switches to provide a better congestion signal. Next, instead of designing a new AQM that needs an entirely new switch design, PRED dynamically adjusts the parameters of the widely commercially proven RED in datacenters. As shown in the lowest branch of the design tree, PRED falls into the category of explicit modeling and A/B testing approach that is more adaptive and stable.

Through experiments, we have discovered that the performance of RED is mainly influenced by the concurrency and distribution of traffic. Prediction model-based ACC [17] initially employed a traffic-aware approach to proactively adjust the RED threshold; However, it failed to provide stable adjustment due to two reasons: **the impact of rapid changes in concurrent flow numbers** and **inaccurate predictions**. Different concurrency levels require different settings, but because there is a mismatch between the delay of DRL and the delay of estimating concurrency, ACC cannot make adjustments based on concurrency levels. Moreover, due to inherent error rates in machine learning predictions, the design principle of ACC fails to achieve consistently stable performance.

PRED employs two loosely coupled systems, namely the Flow Concurrent Stabilizer (FCS) and Queue Length Adjuster (QLA), to address the difficulties associated with unstable adjustments in dynamically adjusting RED parameters. In order to keep the queue length stable under different concurrency levels, FCS uses modeling to adjust RED parameters by counting the number of concurrent flows on a switch port. In order to achieve stable real-time adjustment of the queue length to accommodate varying traffic, QLA employs a step-by-step approach, utilizing testing and verification methods, to gradually modify the RED parameters. With FCS and QLA, PRED can achieve consistently high performance under dynamic traffic in a more stable way without modifying the RED logic.



(a) Two-point form (b) Point-slope form

Figure 2: Two Forms of RED parameter configuration.

We implement a prototype of PRED using the barefoot Tofino switch [24] as the programmable data plane and evaluate PRED extensively using various use cases. Our testbed experiments show that PRED can keep queues stable under various flow-concurrent conditions. Based on real network workloads, we demonstrate that PRED can properly adjust the RED parameters to a more workload-friendly state to keep up with the traffic dynamics.

We further perform larger-scale simulations to evaluate PRED. Our quantitative results confirm that PRED advances state-of-the-art in various aspects. For example, compared with the algorithms with static thresholds, PRED can achieve up to 80% lower Flow Completion Time (FCT) for short flows. From the microscopic view of switch queues, PRED effectively mitigates queue buildups by keeping the switch queue length 66% lower than that of the static threshold algorithms (from 25 to 15 packets). We further demonstrate that PRED reduces the 99th FCT by 34% compared to ACC, the state-of-the-art DRL-based RED configuration approach, whose uncertainty results in suboptimal parameter selections in extreme cases.

The rest of the paper is organized as follows. We introduce the background in § 2 and the motivation of the PRED in § 3. § 4 illustrates the design of our solution. In § 5, we address the implementation of PRED on P4 [25]. In § 6, we evaluate PRED in NS-3 [26] and a small-scale testbed. § 7 surveys the related work. Finally, § 8 concludes this paper.

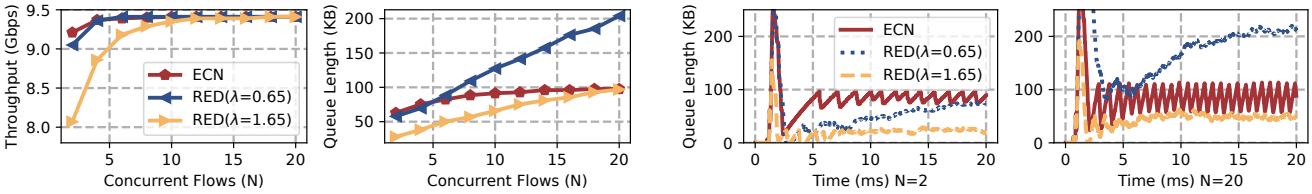
2 Background

2.1 The Form of RED Setting: Two-point Turns to Point-slope

RED [19] is the most widely applied AQM scheme and is commonly supported by commodity switches. [1] The state-of-the-art congestion control mechanisms have widely adopted RED in datacenter networks. For example, DCTCP [3] and DCQCN [5] adopt RED with the standard ECN on switches and use the marked-packet-aware rate control on end hosts. First of all, to facilitate the design of PRED in this paper, we revisit the RED parameter configuration by explaining why we replace the two-point form with the point-slope form.

Two-Point Form. RED works on switches and makes decisions about marking packets based on the output queue length

[1] We note that if not otherwise specified, RED adopts packet marking ECN instead of packet discarding in this paper.



(a) Testbed Result (b) Simulation Breakdown

Figure 3: RED performance with different flow concurrency levels (N).

(denoted by q). The decision-making is driven by three parameters, including the maximum marking probability (maxP), and two queue thresholds (minK and maxK). When the queue length is lower than minK, no action is taken. When the queue length exceeds maxK, every arrived packet is marked. When the queue length is between minK and maxK, RED calculates a probability p that the packet should be marked. In production networks, linear interpolation is the common practice to calculate p as $p = \text{maxP} \cdot \frac{q-\text{minK}}{\text{maxK}-\text{minK}}$. As plotted in Figure 2(a), the resulting curve of p is in the *two-point* form: point (minK, 0) and point (maxK, maxP).

Point-Slope Form. In this paper, we argue that applying the equivalent *point-slope* form makes the design of the RED parameter configuration more interpretable (see §2.2). As shown in Figure 2(b), we define λ ($\lambda \in (0, +\infty)$) as the slope of the line. λ is computed as $\lambda = \frac{\text{maxP}}{\text{maxK}-\text{minK}}$, and thus we have $p = \lambda \cdot (q - \text{minK})$. For example, given $\text{minK} = 0.005$ MB, $\text{maxK} = 0.2$ MB, $\text{maxP} = 0.1$, we have $\lambda \approx 0.5$.

2.2 The Essence of RED Setting: Controlling the Steady-state Queue Length

RED is a tool for congestion control convergence, so adjusting the RED threshold is essentially adjusting the steady-state queue length as it converges. To explain why we replace the two-point form with the point-slope form and the essence of the RED setup, we first take DCTCP as an example of the RED parameter breakdown based on a fluid model [3,5]. Considering N long-lived flows traversing a single bottleneck link with capacity C and propagation delay d (RTT without queueing), the relationship between the steady-state queue length (denoted by $\dot{q}$) and the RED setting is derived as follows[2] (see Appendix A for detailed derivations):

$$(\dot{q} - \text{minK})^2(\dot{q} + Cd) = \frac{2N}{(\frac{\text{maxP}}{\text{maxK}-\text{minK}})^2} = \frac{2N}{\lambda^2} \qquad (1)$$

λ can monotonically influence $\dot{q}$: The first thing we infer from Equation (1) is that λ can monotonically influence $\dot{q}$. This reveals that the point-slope form expresses a more clear physical meaning than the two-point one. If we adjust minK, maxK and maxP, different settings may lead to the same result, for example, (minK,maxK,maxP) and (minK,maxK',maxP') in Figure 2(a). It is easier for the operator to adjust the parameters if there is only one variable and the linear adjustment will result in a linear result. Therefore, the point-slope is better than the two-point because the monotone change in the λ in the two-point form results in a linear change in the steady-state $\dot{q}$. In the latter, we use (minK,λ) to replace (minK,maxK,maxP).

Key factors affecting the $\dot{q}$: From Equation (1), we can see that the steady-state $\dot{q}$ is only related to three key factors. **(a) The network-specific factor**, including the bottleneck link C and propagation delay d. Once the network topology is built, this factor will be fixed. **(b) The flow concurrency level** N. The larger N is, the larger $\dot{q}$ is. If the network experiences drastically changing N, the steady-state $\dot{q}$ will change accordingly. [3] **(c) The setting of RED parameters** (the only one that the network operator typically controls), i.e., λ and minK. By setting the RED parameters, we can properly adjust the steady-state queue length.

3 Motivation

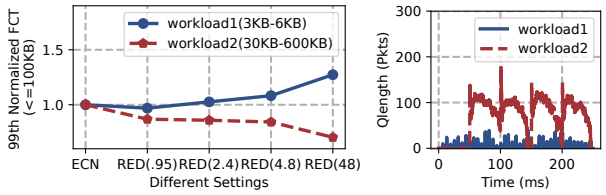
3.1 RED Requires Traffic Awareness

We define the *flow concurrency level* on a switch as the number of flows whose packets are buffered in the switch's output queue at a certain time, define the *flow distribution* as the distribution of flow sizes and define *steady-state queue length* (denoted by $\dot{q}$) as the average queue length of the switch output queue at the stable state. Based on the measurement studies on datacenter network workloads with varying flow concurrency levels and flow distributions, we observe two interesting observations in RED parameter configurations.

Observation 1 (Flow Concurrency): Steady-state queue length grows with the number of flows. From Equation (1) in § 2, we conclude that the larger N is, the larger $\dot{q}$ is. In this section, we also verify this observation through experiments. In these experiments, we run many-to-one (incast) tests with different flow concurrency levels through both testbed evaluation and NS-3 simulation (see setup details in § 6). *ECN* acts as the baseline, which represents the legacy way of marking packets using a single threshold [3], i.e., minK = maxK. *RED* ($\lambda = 1.65$) represents a more aggressive RED parameter setting with higher ECN marking probabilities, and *RED*

[2] Similar conclusions are reached by other ECN-based congestion controls, as demonstrated in studies [27]. Additional sources, including DCQCN and TCP [27, 28], also uphold consistent conclusions. For brevity, we skip the detailed discussion. The assumption of N long-lived flows is for simplicity in understanding the congestion model, but it is not a strict requirement.

[3] The RED capacity of queue adjustment is limited. When N becomes too large, Equation (1) becomes ineffective. The detailed discussion is in § 4.3.



(a) FCT (b) Queue length

Figure 4: RED performance with different workloads.

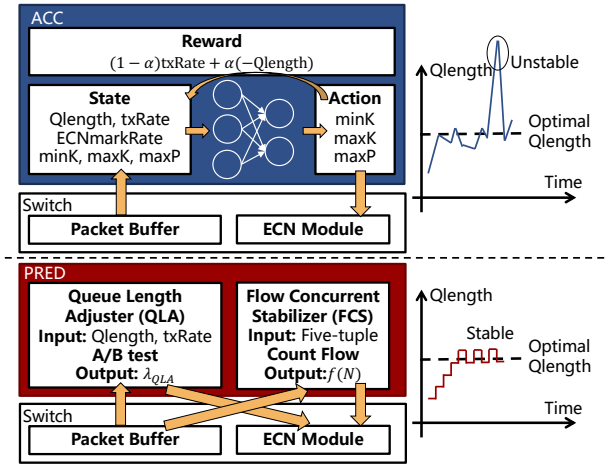


Figure 5: The design comparison of ACC and PRED.

($\lambda = 0.65$) represents a more conservative RED.

Figure 3(a) shows the testbed results of total throughput and queue length on the switch. Through experiments, we again verify that the queue length of both RED ($\lambda = 0.65, 1.65$) increases with the increase of N. This conclusion holds even in the case of multiple bottlenecks, where several hops before the last hop are also congested (see § 6). Datacenter network traffic can be very bursty, but the static RED algorithm increases $\dot{q}$ as the flow concurrency level increases. The tail FCT of short flows will increase with the increase of concurrency, which seriously affects the application performance. A large number of concurrent flows can also suffer from the TCP incast throughput collapse because the queue length is not bounded at the bottleneck switch. A high queue length may trigger back pressure mechanisms (e.g., ECN in DCTCP and priority-based flow control (PFC) in DCQCN) that restrain pushing more packets into the network.

We can also find that different concurrency levels require different λ. *RED* ($\lambda = 1.65$) suffers from under-bandwidth utilization when the flow concurrency level is low, and *RED* ($\lambda = 0.65$) suffers from high queuing delay (queue length) when the flow concurrency level is high. We further zoom into the run-time queue length via simulations for the case of $N = 2$ and $N = 20$. As shown in Figure 3(b), a larger λ (e.g., $\lambda = 1.65$) achieves better performance (i.e., high bandwidth and low latency) when the concurrency level is large (e.g., $N = 20$), and vice versa. Although *ECN* achieves acceptable bandwidth utilization and overall low latency, its amplitude of the steady-state queue length increases with the flow concurrency level. This indicates that applying the single threshold is insufficient to achieve consistent low latency. **To sum up, static RED is not suitable for dynamic flow concurrency.**

Observation 2 (Flow Distribution): Small flows require a large steady-state queue length, while large flows require a small steady-state queue length. In this experiment, we examine RED performance given different flow distributions. We created two different workloads to represent pure small flows and large flows: workload 1 has a flow distribution of size from 3 KB to 6 KB, and workload 2 has a flow distribution of size from 30 KB to 600 KB. In the many-to-one scenario, the number of concurrent senders is fixed at 18 (see setup details in § 6). Figure 4(a) plots the normalized 99th-percentile FCT of flows. The x-axis denotes different settings of λ. It is clear that under workload 1, FCT increases with λ, while under workload 2, FCT decreases with λ. This shows that the proper RED parameter configuration highly depends on the flow distributions in the workload.

For flow distribution such as workload 1, where small flows are dominant, the bottleneck queue is only occupied for a short time. The changes in queue length are spikes one by one in Figure 4(b). Link utilization is not full all the time. Therefore, in this case, increasing the ECN marking probability (increasing λ) will further reduce the link utilization, resulting in a larger FCT of the small flows. To achieve low FCT, the steady-state queue length is expected to be large enough (decrease λ) to avoid triggering packet marking at the bottleneck switch. On the other hand, for flow distribution such as workload 2, where larger flows are dominant, the high link utilization and high bottleneck queue occupancy will last for a long period. This queue occupancy is unnecessary and significantly impacts the latency of small flows. Thus, the steady-state queue length is expected to be small enough to achieve low FCT performance. **In conclusion, static RED is also unsuitable for dynamic flow distribution.**

3.2 Our Goals: Traffic Awareness and Stability

Since the influence of flow concurrency and flow distribution makes the static RED unsuitable for dynamic flows, a natural question is: Can we make the RED parameters configuration traffic-aware? ACC [17] and PRED share a common high-level approach of dynamically adjusting RED parameters. In the following sections, we will highlight the limitations of ACC and explain how PRED overcomes these limitations by achieving stable queue length adjustments.

Limitations of ACC. ACC was the first to use deep reinforcement learning (DRL) to adjust RED parameters to make RED traffic-aware. As shown in Figure 5, ACC takes the following four features as the input state: The current queuing length, the output data rate for each link, the output rate of ECN marked packets for each link, and the current ECN setting. The output action is the next ECN setting. Then, high throughput and short queues are used as reward functions to continuously train the neural network to learn historical data, so as to make

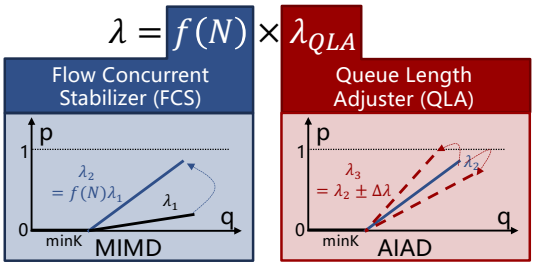


Figure 6: The design rationale of PRED.

predictions for future scenarios. DRL is a valuable approach for adapting to dynamic environments. However, as discussed in § 6.3.3, ACC tends to induce high instability in the queue, resulting in prolonged tail FCT and substantial degradation of the overall user experience.

There are two reasons for the instability of ACC adjustment: (a) **The impact of rapid changes in concurrent flow numbers:** ACC struggles to promptly adjust to varying concurrency levels because of the swift fluctuations in N, resulting in instability. Given that the delay in DRL falls within the millisecond range, while the calculation of concurrency level N operates at the microsecond scale, achieving real-time matching of N is infeasible. Even if ACC can match N in real-time, it also has the following problem. (b) **Inaccurate predictions:** ACC operates on a prediction model derived from observed and learned historical data to make decisions. However, incorporating additional network conditions does not ensure accurate predictions matching 100% real-world scenarios. Predictions inherently carry an error rate, and when an inaccurate prediction is made, the network's performance becomes unstable, ultimately leading to a degradation in tail FCT. Next, we will explain how PRED can be stabilized through the following two problems.

How to bound queue according to flow concurrency? Our answer is to count N directly in the switch and adjust the RED λ according to the flow concurrency. In Equation (1), we analyzed the relationship between $\dot{q}$ and N and λ. The left-hand term $(\dot{q} - \text{minK})^2(\dot{q} + Cd)$ of Equation (1) contains the steady-state queue $\dot{q}$, and if we want $\dot{q}$ to remain constant, then the right-hand term $\frac{2N}{\lambda^2}$ of Equation (1) needs to remain constant. N changes dynamically, but if the switch can calculate the current value of N and then increase/decrease the corresponding λ, then the steady-state queue length $\dot{q}$ will not change with N. We design the Flow Concurrent Stabilizer (FCS) to count the N and change the λ to the concurrency level. As shown in Figure 5, the input to the FCS is a five-tuple of each packet. It outputs the multiplier factor of the change in the RED parameter by calculating the number of flows over a period of time. In Figure 6, FCS works in the form of Multiplicative-Increase Multiplicative-Decrease (MIMD) because it can calculate the corresponding λ directly from N.

How can queue length adjustment be traffic-aware in a stable manner? Our answer to this question is that we test and then verify in small steps. The traffic is uneven and time-varying in datacenters. In order to achieve stable high performance, some workloads demand a smaller steady-state queue length to operate at the optimized point [29], while others demand a larger steady-state queue length to avoid inefficient link utilization. It is difficult for the switch to know the needed steady-state queue length. But the switch knows the queue length and throughput. If the switch can always maintain a high throughput and a low queue length, this queue length must be close to the needed steady-state queue length. We design the Queue Length Adjuster (QLA) to achieve traffic awareness at run time. In Figure 5, QLA takes throughput and queue length as input and outputs a corresponding linear factor λ_{QLA} through the A/B test. First, it sets a utility function in terms of goodput and queue length to judge the performance of the current RED setting. And then, as shown in Figure 6, it applies a direct small modification to λ_{QLA} to adapt λ to the traffic. To reduce the impact of noise, it is tested twice and adjusted only when the results are consistent. Note that any measurement behavior will affect the network performance itself, so the detection step of FCS should be small. Using any of the more aggressive detection algorithms here leads to instability in the network, such as Additive-Increase Multiplicative-Decrease (AIMD), binary search, and prediction-based machine learning algorithms. The QLA works in the form of Additive-Increase Additive-Decrease (AIAD) because it needs stable but fast convergence. AIAD is a heuristic process that relies on A/B testing. Unlike other methods, it demonstrates a clear convergence direction with incremental testing steps. However, it tends to operate at a slower pace compared to other algorithms, which may prioritize speed over stability.

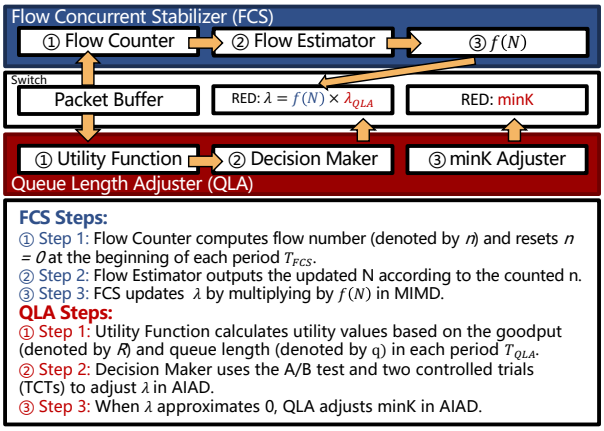
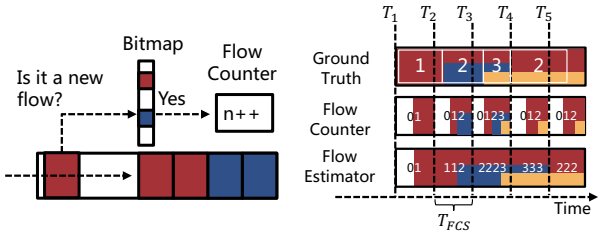


Figure 7: The design of PRED.

4 Design

In this paper, we propose PRED to answer the question of how to set RED settings dynamically. Figure 7 illustrates the design of PRED, primarily focused on adjusting λ to maintain an optimized $\dot{q}$. The FCS determines N using the flow counter and flow estimator. Subsequently, it applies $f(N)$, a monotonically increasing function of N, to dynamically



(a) Flow Counter (b) Flow Estimator

Figure 8: Examples of how the modules work in FCS.

adjust λ according to the concurrency level. The QLA directly modifies λ_{QLA} to achieve traffic-awareness at runtime. In the following sections, we will delve into each module, covering topics such as N estimation, utility function design, noise reduction through two controlled trials, and dynamic adjustments of minK.

4.1 Flow Concurrent Stabilizer

As shown in Figure 7, the FCS comprises three modules: Flow Counter, Flow Estimator and $f(N)$. (1) Within each T_{FCS} period, the Flow Counter computes the number of flows (denoted by n) traversing a port and resets $n = 0$ at the period's onset. (2) Simultaneously, the Flow Estimator calculates the updated value of N based on the accumulated n. (3) Subsequently, the FCS module adjusts λ by multiplying it with $f(N)$. In the following, we explain how the Flow Counter and Flow Estimator work and discuss the selections of $f(N)$.

Flow Counter (to count new flows). A flow can be identified by the hash of the five-tuple: <source_ip, destination_ip, source_port, destination_port, protocol>. We define the start of the flow as the first time a packet with a particular five-tuple appears at a switch and count the number of flows that send a packet within each timeout-sized interval T_{FCS}. This new-arrival count way helps to avoid biases in cases when the start of flows or the end of flows are missed [30]. The logic is straightforward, as in Figure 8(a), the Flow Counter checks if it is a new flow arrival by looking up the bitmap maintained for each port. If yes, the flow counter increases n by 1. For every period of T_{FCS}, bitmap and n are reset and recalculated.

Flow Estimator (to estimates N). Flow Estimator estimates N by taking into account both the current-period estimate n and the last-period estimate n_{last} output by the Flow Counter, which is $N = \max\{n_{last}, n\}$. Since Flow Counter always starts at 0, we can't treat n as the current N. Figure 8(b) shows an example. 'Ground Truth' represents the actual value of flow concurrency at the present moment. At T_2, the Flow Counter outputs $n = 0$, while the last-period estimate $n_{last} = 1$. If FCS uses n instead of N, FCS will get a bad flow estimate of 0. In this paper, we estimate N at T_2 as $N = \max\{n, n_{last}\} = 1$.

Selection of $f(N)$. In order to bound the steady-state queue length regardless of flow concurrency, according to Equation (1), $f(N)$ should meet $f(N) \geq \sqrt{N}$. Through experiments, we find that using $f(N) = \sqrt{N}$ cannot completely bound the length of the steady-state queue because Equation (1) assumes N synchronous traffic, which is difficult to meet when N becomes large. We test with the choice of $f(N)$ through experiments and choose $f(N) = N$. We have further given the reason why N is feasible in § 6.

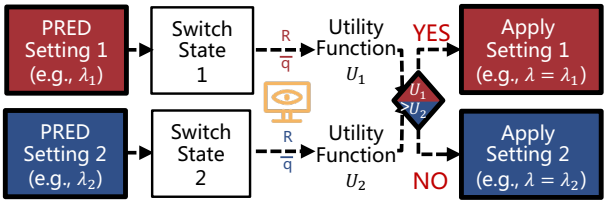


Figure 9: The closed-loop decision-making of QLA.

4.2 Queue Length Adjuster

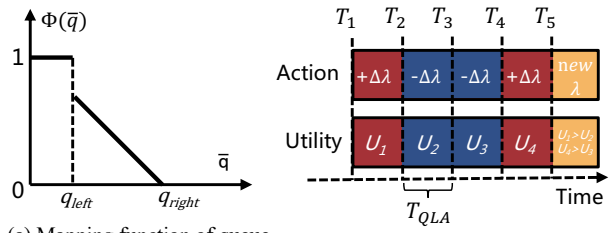
As shown in Figure 7, the QLA comprises three modules: Utility Function, Decision Maker and minK Adjuster. (1) The Utility Function calculates utility values based on the goodput (denoted by R) and the average queue length (denoted by $\bar{q}$) within each period T_{QLA}. (2) The Decision Maker employs the A/B testing and two controlled trials (TCTs) to adjust PRED in AIAD. (3) If deemed necessary (e.g., when λ approaches 0), it further adjusts minK using a similar approach.

The QLA module applies a closed-loop decision-making structure to set λ according to the demand of the traffic pattern. The efficiency of this performance-oriented learning approach is also validated in prior works (e.g., PCC [31–33]). As illustrated in Figure 9, the control action of QLA is its choice of PRED setting. The decision-making is based on the A/B testing of two actions, e.g., $\lambda = \lambda_1$ and $\lambda = \lambda_2$.These performance metrics are combined with the numerical utility values, say U_1 and U_2, respectively, via the utility function. The decision-making of λ is then based on comparing U_1 and U_2. However, the dynamic nature of the network can produce a lot of noise, which leads to inaccurate results in the A/B testing. To reduce the impact of noise, TCTs adjusts only when the results are consistent.

Utility Function (to decide trade-offs). From the network operator's point of view, we usually select the critical network performance metrics of latency and throughput as the reward function [17]. We define the utility function as a trade-off between latency and throughput (i.e., the trade-off between high utilization and low queue):

$$U(\lambda_t) = \beta \times \frac{R_t}{C} + (1-\beta) \times \Phi(\bar{q}_t)$$

R represents the average throughput of one egress queue, i.e., the amount of data delivered to the link during the time interval T_{QLA}. We normalize the R by the link bandwidth C to represent the link utilization. The latency is represented by the average queue length $\bar{q}$ to indicate the impact of queuing delay. β is a weight factor. We select the average value instead of the instantaneous queue length because the instant queue length varies in a large range, which can make utility unstable. $\Phi()$ is a mapping function as shown in Figure 10(a). Generally, the lower the queue length, the better. Note that when the



(a) Mapping function of queue length

(b) Two controlled trials

Figure 10: Design details of QLA.

queue goes short (e.g., $\bar{q} \leq q_{left}$), the marginal benefit of further reducing the queue length goes to zero. This avoids the situation where the utility function becomes unstable when the queue is too short.

Decision Maker (to reduce noise impact). A simple A/B test will be disturbed by network noise. To decide which direction and amount to change its value, QLA uses TCTs to reduce the noise impact. Assume QLA is currently at setting λ_{QLA}. As shown in Figure 10(b), QLA takes four consecutive T_{QLA} and divides them into two pairs. For each pair, QLA attempts a slightly higher $\lambda_{QLA}+\Delta\lambda$ and slightly lower $\lambda_{QLA}-\Delta\lambda$, each for one T_{QLA}. After the four consecutive trials, QLA needs to judge the next adjustment based on the four utility values ($U_i, i = 1,2,3,4$) obtained four times. If the higher λ_{QLA} consistently has higher utility ($U_1 > U_2$ *and* $U_4 > U_3$), then FCS adjusts its λ_{QLA} to $new\lambda_{QLA} = \lambda_{QLA}+\Delta\lambda$; and if the lower λ_{QLA} consistently has higher utility then QLA picks $new\lambda_{QLA} = \lambda_{QLA}-\Delta\lambda$; But, if the results are inconclusive, e.g., $U_1 > U_2$ *but* $U_4 < U_3$, QLA stays at its current λ_{QLA}.

Adjustment of minK. Besides λ, the other important value in the RED parameter is minK. The initial value of minK can be set very small so that λ can make the steady-state queue adjusted over a wide range ($[\text{minK}, +\infty)$) theoretically. However, in the design, we cannot use $\lambda = 0$ to get a large steady-state queue $\dot{q}$ because the FCS module fails. So when $\lambda < \lambda_{\min}$, QLA no longer adjusts λ and adjusts minK instead. Let minK change ΔminK each time and λ always stays λ_{min} for FCS. For example, in § 3.1, workload 1's burst flow does not occupy the buffer. In this case, minK should be bigger, and the steady-state queue $\dot{q}$ is further increased to avoid the burst being marked with ECN.

4.3 Discussion

Some notable design details of PRED are essential for reasoning about the differences from legacy RED. We briefly discuss these considerations below.

Range of PRED adjustment capabilities. The RED parameter adjustment has limitations, particularly when N surpasses a certain value. Through experiments, we test the adjustment range of PRED and observe that when the number of concurrent long flow N exceeds 32, PRED is unable to maintain queue stability (see Appendix D.6). Fortunately, the occurrence of concurrent long flows is rare [3], indicating that the range of PRED is generally adequate in practice. Moreover, to address high-concurrency short-flow bursts, FCS can rapidly notify the end nodes by adjusting λ to ensure fast convergence of the queue.

About the maxK. In PRED, we utilize the point-slope form (minK,λ) instead of the the two-point form (minK,maxK,maxP). This means that in the point-slope form, the maxK is implicitly incorporated within the parameter λ. However, we acknowledge that maxK is still necessary for final protection in practical implementation because the point-slope is still implemented on the two-point basis. To accommodate extreme scenarios, we set a larger fixed value for maxK. In our measurements, maxK is set to 500 KB.

Selection of T_{FCS}. The T_{FCS} should be carefully set as its accuracy significantly affects the performance. First of all, T_{FCS} must be greater than RTT because the estimate will be small because at least one RTT is required for all flows to pass through the switch. T_{FCS} should not be too big because when the flow number decreases, the update of N will be delayed. As shown in Figure 4.1(b), N remains 3 from time T4 to time T5, while the ground truth is only 2. Considering the queuing delay during congestion, we find that setting $T_{FCS} = 1.25RTT$ is applicable. We have also verified the parameter sensitivity analysis in Appendix D.5.

Selection of T_{QLA}. Note that T_{FCS} is independent of T_{QLA} because the two modules are separate modules. The setting of T_{QLA} is a tradeoff. The smaller T_{QLA}, the faster the convergence, but the more unstable. Vice versa. In addition, it is also limited by the location where the QLA module is deployed. If the QLA module is deployed on the control plane, the interaction delay between the control plane and the data plane should be considered. In our implementation, reading the register in the control plane takes 5 to 12 ms, and updating the RED table takes 80 to 110 ms. Therefore T_{QLA} is at least 200 ms due to hardware limitations. We discuss the details in Appendix C.2. In this paper, it is recommended $T_{QLA} = 5RTT$ in simulation (on the data plane) and $T_{QLA} = 400$ ms in the testbed (on the control plane due to hardware limitation). We have also verified the parameter sensitivity analysis in Appendix D.5.

Selection of other parameters. PRED also has some parameter settings that are trade-offs. In QLA, β is the weight that represents the bandwidth-delay tradeoff. The network operator can easily set the parameters based on the requirement of running applications. For example, our measurements are based on $\beta = 0.4$. The initial minK needs a relatively small value in order to ensure that the adjustment range of the λ can be larger. The initial minK is 10 packets. q_{left} is the parameter that QLA uses to ensure that the value of the utility function remains stable when the queue is small. The larger the q_{left}, the more stable the utility value, but the longer the queue length. q_{left} is set to 15 packets in this paper. λ_{min} is the dividing point between the adjusting λ and the adjusting minK in QLA, we use $\lambda_{min} = 0.05$, For the trial step parameter setting, the bigger the trial step, the faster the adjustment speed, but

the worse the stability, we use $\Delta\lambda = 0.025$, and $\Delta\text{minK} = 5$ packets in this paper. PRED tunes the RED parameters because they are sensitive to traffic dynamics. The parameters introduced by PRED are traffic-insensitive. However, fine-tuning these parameters remains essential, and enhancing the parameter selection process is a key focus of our future work.

5 Implementation

Ideally, PRED's FCS and QLA would be implemented in switch ASICs. We implement a prototype of PRED on Barefoot Tofino 1 programmable switch, including 12 MAU stages, 120 Mbit SRAM and 6.2 Mbit TCAM per pipeline. The development effort on the switch includes about 350 lines of *P*4 code [34] for the data plane and 300 lines of Python for the control plane. The FCS module needs to quickly recognize N, so it is deployed on the data plane. Theoretically, the QLA can be deployed on either the data plane or the control plane. Deployment on the control plane has low resource overhead but high latency, and vice versa. We chose to deploy the QLA to the control plane due to limited MAU stages. We have implemented PRED where both QLA and FCS are deployed on the data plane, but 17 stages are needed (see details in Appendix C). Tofino 1 [24] has only 12 stages, so we deploy the QLA module on the control plane. However, the new Tofino 2 and Tofino 3 switches support 20 stages [24], so the QLA and FCS modules can be deployed in the data plane at the same time. All implementation details are deferred to Appendix C.

6 Evaluation

In this section, we conduct both testbed experiments and NS-3 simulations[4] to answer the following questions:

- How does PRED perform in practice and scale to large datacenters?
- Why does PRED achieve high performance?
- What are the advantages of PRED compared to Deep Reinforce Learning methods?
- How do we decide the key parameters in PRED itself?

6.1 Methodology

Transport Protocol. We are using DCTCP [3] as the default congestion control at the end host. We also have experiments to verify the capabilities of PRED with DCQCN (details in Appendix D.1). Since the results are similar, we only use DCTCP later. In our testbed experiments, we used DCTCP from Linux kernel 4.15.0 [35]. The NS-3 simulation [26] implementation of DCTCP refers to the official implementation [36]. The parameters are set as suggested in [3].

Schemes Compared. We compare PRED against the following 9 schemes:

(a) RED [testbed, simulation]: RED's implementation is based on instantaneous ECN marking rather than weighted average queues, as in the DCTCP [3] and DCQCN [5] papers. For the testbed, we implement it on the Barefoot Tofino switch. For simulation, we start from the NS-3's RED implementation and add instantaneous ECN marking.

(b) ECN [testbed, simulation]: Here we refer to the single-valued RED algorithm as ECN, i.e., instantaneous ECN marking based on a single threshold $minK = maxK = K$.

(c) ECNsharp [simulation]: ECNsharp [15] marks packets based on both instantaneous and persistent congestion.

(d) CoDel [simulation]: CoDel [37] tracks minimal queueing over an interval to mark packets based on persistent queueing.

(e) ACC [simulation]: ACC [17] is a practical approach that allows automatic adjustment of ECN parameters at switches by utilizing deep reinforcement learning (DRL). We refer to the original design in paper [17] and re-implement ACC in the NS-3 simulation.

(f) Only FCS [testbed, simulation]: The particular PRED which contains only the Flow Concurrent Stabilizer.

(g) Only QLA [testbed, simulation]: The particular PRED which contains only the Queue Length Adjuster.

(h) TCN [simulation]: TCN uses instantaneous sojourn time to adapt to packet schedulers. We implement TCN based on the software prototype provided by TCN paper [38].

(i) HPCC [simulation]: HPCC [7] is a new congestion control scheme, which needs to modify the NIC and the switch that supports INT. HPCC can adjust the rate MIMD through the INT information.

Metrics. We use the Flow Completion Time (FCT) as the primary metric. Besides the overall average FCT, we also break down FCT results across small flows (< 100 KB) and large flows (> 1 MB). All are normalized to the results achieved by PRED. We also show a queue length timing diagram for a fine-grained comparison of the various algorithms.

6.2 Testbed Experiments

In this section, we analyze the performance of PRED. Our assessment aims to confirm the stability of traffic concurrency (FCS), assess the effectiveness of dynamically adjusting queues in response to traffic dynamics (QLA), and determine the importance of integrating both modules.

Flow Concurrent Stabilizer. We first tested PRED's stability of traffic concurrency, as well as different design options. We use the testbed in § 3.1 with 3 servers connected to a Barefoot Tofino switch with PRED 's implementation. There are 2 senders and 1 receiver. We used iperf3 [39] for sending traffic and measured the throughput and queue length by varying the concurrent flows sent by each sender. Using only two senders is to verify that concurrency problems still exist in the case of multiple bottlenecks (one at the sender and one at the switch). We use RED as a comparison. We pick initial λ whose effect is approximately equal when N=2. λ in FCS(N) and PRED is 0.1 ($2 \times 0.1 = 0.2$), λ in FCS(N^2) is 0.05 ($2^2 \times 0.05 = 0.2$), and λ in FCS($\sqrt{N}$) is 0.2 ($\sqrt{2} \times 0.2 = 0.28$).

Figure 11 shows the performance of PRED and only FCS with different design choices. We can see from Figure 11(a)

[4] The implementation in NS3 is identical to that of the programmable switch, yet the languages differ; NS3 is C++, and the testbed is P4.

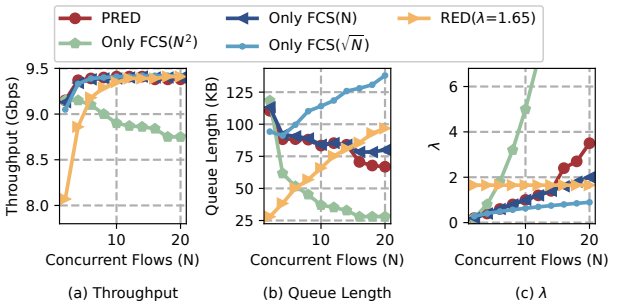


Figure 11: [Testbed] PRED performance with different flow concurrency levels (N).

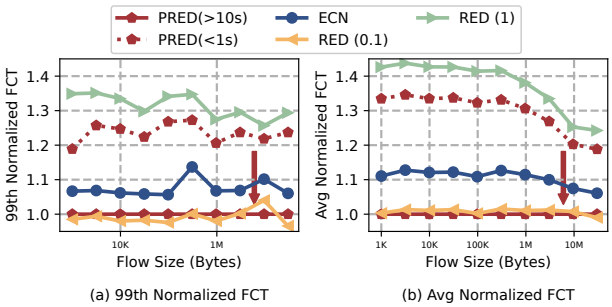


Figure 12: [Testbed] FCT statistics ($T_{QLA} = 0.4$ s).

that all algorithms except FCS(N^2) and RED can maintain a relatively high throughput of about 9.4 Gbps. FCS(N^2) sacrificed about 0.4 Gbps throughput due to low queue length. From Figure 11(b), the queue length of PRED and FCS(N) remained stable with the increase of N. The queue length of FCS($\sqrt{N}$) was still growing, but the queue length of FCS(N^2) was decreasing with the increase of N. The reason for the noise when $N = 2$ is that there is not enough concurrency. Figure 11(c) shows the λ as N changes.

The above analysis shows that it is most appropriate for PRED to select the N for the FCS module. In § 4.1, we analyze that $f(N) = \sqrt{N}$ can eliminate the influence of N on queue length. However, the modeling assumes concurrent traffic, so when N<5, the FCS module can still maintain a stable queue, but when N>5, the concurrent traffic assumption fails so that the queue will grow. We also tested $f(N) = N$ and $f(N) = N^2$, N^2 turns out to be too aggressive, while N is more feasible.

Finally, we compared PRED and only FCS (N) separately. As can be seen from Figure 11(b), when N>15, PRED will make the queue length lower without losing throughput. Compared with only FCS, PRED can further adjust the queue length by adjusting λ according to the network condition.

Queue Length Adjuster. We next tested PRED's ability to tune parameters dynamically. We use 7 servers connected to a Barefoot Tofino switch. There are 6 senders and 1 receiver. Each sender uses an open-source traffic generator [40, 41] to generate the benchmark traffic to the receiver. The network load is 60%. K in ECN is 70 packets. We generate traffic based on WebSearch [3] realistic workloads.

Figure 12 shows FCT on different flow sizes. From the results, we can see that the 99th FCT and average FCT of RED (λ =0.1) is the smallest, and the FCT of RED (λ =1) is the largest. With the same workload, PRED does not initially have a suitable λ, so the FCT of PRED (<1s) is close to RED (1). When run for about 10 s, PRED converge to the appropriate λ, so the FCT of PRED (>10s) converges to RED (0.1). Note that the implementation of PRED on Tofino 1 places QLA on the control plane, so $T_{QLA} = 0.4$ s. As discussed in § 5, the bigger T_{QLA} is, the slower convergence rate is, but the deterministic direction of adjustment is stable. If PRED can be implemented on Tofino 2 or a better switch, T_{QLA} can become smaller and PRED can converge faster.

Based on the above analysis, we know that PRED can bound steady-state queue length with dynamic flow concurrency and can make queue length adjustment traffic-aware in a stable manner to maximize the network effect.

6.3 Simulation Experiments

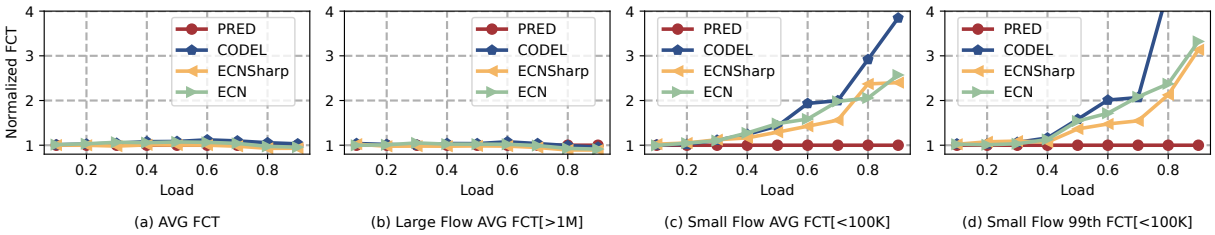
In this section, we will use simulation to analyze the performance of PRED. At the same time, we will fine-grain compare the differences between different algorithms and answer why PRED can have a good performance. First, the performance of PRED in real datacenters is tested by large-scale NS-3 simulation. Then we analyze why FCS and QLA should cooperate with each other through the sequence diagram of queue length and analyze why the performance of different algorithms is not optimal. Then we compared the differences between DRL-enable ACC and PRED. Finally, we analyze the PRED related parameter settings and packet scheduler in detail.

6.3.1 Large Scale Simulations

To complement our testbed experiments, we evaluate PRED on a larger-scale spine-leaf topology with realistic workloads.

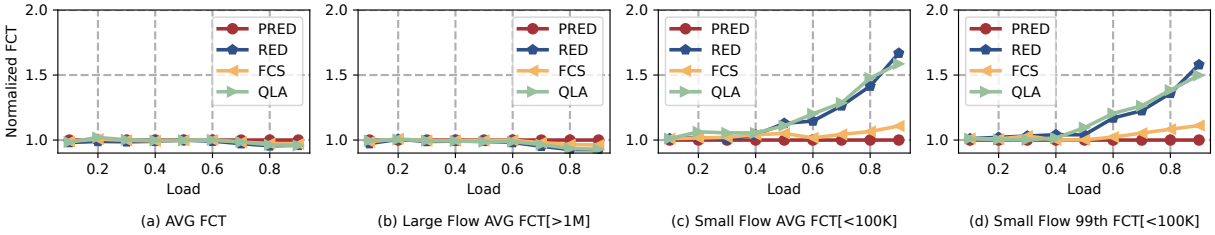
Setup: We simulate a 128-host leaf-spine topology with 8 spine and 8 leaf switches. Each leaf is connected to 16 servers via 10 Gbps links. The spine and leaf switches are also connected via 10 Gbps links. The latency of the link is 10 μs. We use ECMP for load balancing. We generate traffic based on two realistic workloads in production: WebSearch [3] and DataMining [42]. Each sender sends messages in a Poisson flow, and the target loads for the fixed receiver range from 10% to 90%. K in ECN is 70 packets. The instantaneous marking threshold for ECNSharp is 80 μs, the persistent target threshold is 10 μs and the persistent interval is 150 μs. For CoDel, we set the interval to be 150 μs and the target to be 10 μs. The results are shown in Figures 13 and 14. Due to the space limitation, We listed the DataMining results in Appendix D.2. All results have been normalized to FCT achieved by PRED.

Results: As shown in Figure 13, PRED's small flow 99th FCT (1.4 ms) has a reduction ranging from 68% (ECNSharp, 4.34 ms) to 80% (CODEL, 6.68 ms) compared to ECN algorithms at 90% load. For Large flows, there was a negligible increase in the FCT of PRED (96.8 ms), ranging from 4.6% (CODEL, 92.5 ms) to 12.5% (ECNSharp, 86 ms) compared to ECN algorithms at 90% load. This proves that PRED can outperform other ECN algorithms in real-workload scenar-



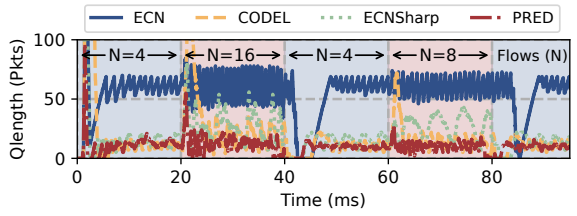
(a) AVG FCT (b) Large Flow AVG FCT[>1M] (c) Small Flow AVG FCT[<100K] (d) Small Flow 99th FCT[<100K]

Figure 13: [Simulation] FCT statistics with different ECN algorithm.

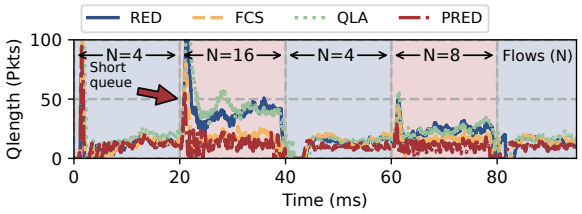


(a) AVG FCT (b) Large Flow AVG FCT[>1M] (c) Small Flow AVG FCT[<100K] (d) Small Flow 99th FCT[<100K]

Figure 14: [Simulation] FCT statistics with different RED algorithm.



(a) Compare with ECN algorithm



(b) Compare with FCS and QLA

Figure 15: [Simulation] Queue Length.

ios. In Figure 14, PRED and FCS perform better than the static threshold RED. QLA alone does not improve performance. FCS alone is not optimal at high concurrency, and QLA should be combined to further reduce the FCT of small flows.

6.3.2 PRED Microscopic View

We want to know why PRED is better, so we conduct a comparative analysis through time sequence diagrams.

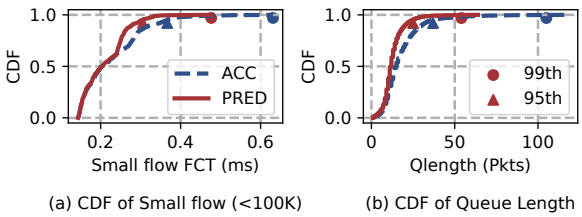
Setup: We use a simple 16-to-1 topology with 10 Gbps links, 16 servers are senders and 1 receiver. Other settings are similar to § 6.3.1. To provide a clearer illustration of how different schemes handle their queues, we sample the queue length of the bottleneck link every 10 μs. The number of concurrent flows is constantly changing.

Results: Figure 15 shows the queue changes of different algorithms at the bottleneck port. PRED effectively eliminates queue buildups by keeping the switch queue length 66% lower than that of the static threshold algorithms (from 25 packets to 15 packets). As can be seen in Figure 15(a), PRED gradually adjusts the queue length over time, making the queue lower than other algorithms. And since it is essentially a RED algorithm, probabilistic marking makes the queue more stable, and the vibration amplitude will be smaller. ECNsharp uses two ECN thresholds, but fixed adjustment makes the queue unstable as shown in Figure 15(a). In addition, PRED can converge faster when a burst is encountered, resulting in less queue fluctuation.

Figure 15(b) compares FCS and QLA. From Figure 15(b), FCS can converge rapidly but will keep the queue length stable at about 23 packets, while PRED will further reduce the queue length. Because QLA cannot adjust λ MIMD according to N, the convergence process is turbulent. Therefore, we can conclude that FCS and QLA need to cooperate so that PRED can achieve the best performance.

6.3.3 Compared with ACC

ACC uses DRL to automatically adjust the RED parameter. DRL requires a lot of data and training to use. We compare PRED and ACC with a simple many-to-one incast scenario. The RED part of ACC uses NS3 simulation and the Learning part of ACC uses NS3-GYM [43]. We use only one agent to learn the traffic conditions in the congested port, reducing the amount of data to learn. We use Double DQN [23] to implement ACC. The **State** is 6 normalized values $(\bar{q}, R, R^m, \text{minK}, \text{maxK}, \text{maxP})$. They are the queue length $(\bar{q})$, the port speed (R), the speed of marking ECN (R^m), and the current RED setting $(\text{minK}, \text{maxK}, \text{maxP})$. **Action** space dimension is $4 \times 3 \times 21$, i.e., $\text{minK}(2,4,8,16) \times \text{maxK}(20,50,100) \times \text{maxP}(\{1\%, j \times 5\%\}, \forall j \in [1,20])$. The **Reward** function is consistent with PRED's utility function. We train ACC for 200,000 epochs using the same trace for both training and testing. It's important to note that in our current testing, ACC encounters the same trace as during training. Even if we aim for deep reinforcement learning to over-fit the results, the stability of the outcomes cannot be guaranteed.



(a) CDF of Small flow (<100K) (b) CDF of Queue Length

Figure 16: [Simulation] CDF of FCT and Queue length.

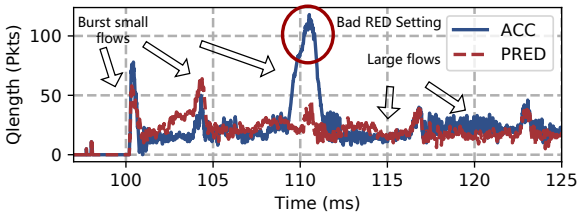


Figure 17: [Simulation] Qlength of ACC and PRED.

Setup: We use a simple 18-to-1 topology with 10 Gbps links, 18 servers are senders, and 1 receiver. Other settings are similar to § 6.3.1. We send flows from 18 senders to the receiver. The size of both large and small flows are generated based on two artificial workloads mentioned in § 3.1 (workload 1 (3-6 KB) for small flows and workload 2 (30-600 KB) for large flows). Both kinds of traffic are sent synchronously, and we measure the FCT of small flows (< 100 KB).

Results: Figure 16 shows the FCT and queue length of ACC and PRED in this particular scenario. As can be seen from Figure 16(a), FCT is similar at 50th, but PRED reduced 99th FCT by 34% compared to ACC (from 0.63 ms to 0.47 ms). From Figure 16(b), the tail of the queue length in ACC is also larger. The 95th percentile in the ACC is 37 packets, and the PRED is 25 packets; The 99th queue length ACC is 105 packets, while the PRED is 54 packets.

According to the above analysis, PRED was better than ACC in tail FCT and tail queue length. To further understand why PRED is better than ACC, we selected a time slice of the ACC runtime, shown in Figure 17. From the figure, the queue length convergence of PRED and ACC is similar because they share the same objective function. However, at around 110 ms, the ACC queue length significantly increases. The reason is that even after ACC is deployed, it still needs to have a random exploration probability from learning new network condition changes. Some bad RED Settings will be selected (around 110 ms in Figure 17), resulting in large queue lengths and high tail latency. In conclusion, the use of machine learning methods needs to be carefully set up in the exploration scheme, The unreadable and untunable model will affect the tail latency. A white-box PRED can ensure that the queue converges to the appropriate position.

6.3.4 100 Gbps with 1000s Concurrent Flows

We also evaluate PRED on 100 Gbps networks and with thousands of concurrent flows.

Setup: The topology is the same as the one used in § 6.3.1 with a 128-host leaf-spine topology and 100 Gbps links. The

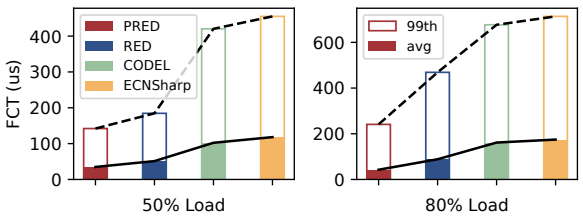


Figure 18: [Simulation] Small Flow FCT statistics with DataMining workload.

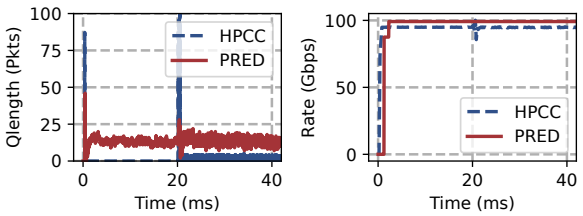


Figure 19: [Simulation] Comparison of PRED and HPCC.

latency of the link is 1 μs. On 128 hosts, randomly select 1000 senders, with an average of 7 senders started per host, and choose a fixed receiver. Each sender sends messages in a Poisson flow, and the target loads for the fixed receiver are 50% and 80% respectively. The DataMining [42] results are shown in Figure 18. Due to the space limitation, We listed the WebSearch results in Appendix D.3.

Results: As shown in Figure 18, PRED's small flow 99th FCT (240 μs) has a reduction ranging from 48% (RED, 468 μs) to 66% (CODEL, 713 μs) at 80% load. PRED's small flow average FCT (42 μs) has a reduction ranging from 52% (RED, 88 μs) to 75% (CODEL, 174 μs) at 80% load. This proves that PRED can outperform other ECN algorithms in 100 Gbps with 1000s concurrent flows.

6.3.5 Comparison of PRED and HPCC

We also compare PRED with HPCC [7] to show how PRED compares to the newer congestion control schemes. HPCC's convergence target is 95% bandwidth, so it can always keep the short queue length and is very friendly for short flows. HPCC needs to modify NICs and switches but PRED only needs to modify switches incrementally. Here we want to see how far PRED is from HPCC.

Setup: We use a simple 20-to-1 topology with 100 Gbps links, 20 servers are senders and 1 receiver. The latency of the link is 1 μs. The parameters are set as suggested in [7]. The NS-3 simulation implementation of HPCC refers to the implementation in [7]. Starting at 0 ms, start two long flows. At 20 ms, 6 long flows are started concurrently.

Results: Figure 19 shows the queue length and throughput of PRED and HPCC. As shown in Figure 19, HPCC has consistently maintained a low queue length of almost zero and 93% throughput. The PRED queue length gradually converges from 20 to 15, and it is always full throughput. At 20 ms burst, the PRED queue fluctuates less, mainly because HPCC does not have a slow start process, so the queue changes greatly.

6.3.6 Packet Scheduler and Parameter Design Choice

We also compare PRED with TCN [38] and ECNsharp [15] to show how PRED works with arbitrary packet scheduler and analyze parameter design choice. Due to space limitations, we present the results in Appendix D.4.

7 Related Work

The most relevant work for PRED is ACC [17], which has been discussed in detail in this paper. Next, we will introduce some other related work.

Dynamically Adjust the RED Threshold. Since the RED algorithm was initially proposed, many proposals studied how to adjust the RED parameters dynamically [44–48]. Feng's ARED [44, 45] adjusts the marking/dropping probability, maxP, in RED to keep the average queue size stable between minK and maxK, in the form of MIMD. Floyd's ARED [46] adjusts maxP in AIMD way to make the queue length fluctuate around a certain value. STAQM [48] adjusts AQM parameters by modeling and parameterizing the RED algorithm and PI algorithm [49] to estimate network state according to network measurement and estimation. These art demonstrate the benefits of dynamic RED parameters. However, they only assure convergence of queue length, while the exact converged queue length must be manually set. As a result, their applicability in datacenter networks is limited.

Recently, there also exist some new algorithms [14–16] for ECN proposed in datacenters. BCC [16] dynamically adjusts the RED algorithm based on the global shared buffer usage. Port-level RED is used when utilization is low, and the shared buffer RED is used when the utilization is high. TCD [14] finds that in lossless networks, the queue has an undetermined state between congestion and non-congestion. It is critical to identify the undetermined state and notify the end host. These art found that ECN with a fixed threshold would be ineffective in datacenters and therefore set more than one state during threshold adjustment. However, they still use fixed thresholds that cannot automatically adjust based on traffic dynamics.

Buffer Sizing and AQM in Internet. The most well-known rule of buffer sizing showed that the minimum buffer size should be $C \times RTT/\sqrt{N}$ when there are a large number of N long-lived TCP Reno flows [50]. Existing new congestion controls such as BBR [29] also require new theories to set buffer sizes [51,52]. Bruce et al. [30] also discussed how to estimate the number of N on a router. The above studies inspired PRED's motivation to dynamically adjust RED Settings by counting traffic numbers in datacenters.

Most ECN-based datacenter congestion control algorithms [3, 53, 54] set the two thresholds of RED to the same value, $\text{minK} = \text{maxK} = K$. The ideal ECN marking threshold K is given as $K = \lambda \times C \times RTT$ [3, 38, 41, 54, 55]. Different congestion controls have different λ, TCP is 1 [54], DCTCP is 0.17 [55]. Through experiments in §3.2, we found that the relationship between single-value RED and N is that the amplitude of the queue increases with the increase of N. Therefore, PRED does not regulate based on single values.

MQ-ECN [41] first pointed out the drawbacks of existing ECN/RED implementations in the packet scheduling context. To adapt to the varying queue capacity caused by packet schedulers, TCN [38] proposed to use instantaneous sojourn time to mark packets. ECNsharp [15] inherits the merit of TCN but further tracks the persistent congestion state to reduce long-term queue buildups. PRED can make the steady-state queue consistent for different priorities/queues so that it can be well adapted to different packet schedulers.

Congestion Control for Datacenters. Most of the network congestion problems are solved by proposing new congestion control solutions. HULL [56] sacrifice throughput in exchange for low latency. It marks the ECN by calculating link utilization rather than queue length. PRED can use a scheme similar to the HULL, but the difficulty is that it requires the endpoint to use the packet pacing function to ensure throughput utilization. pFabric [57], BFC [9], and HPCC [7] rely on precise in-network state information of switches and update transmission rate for each flow. pFabric achieves near-optimal FCT by using (infinite) priority queues on switches. The BFC enables the switch queue to reach one-hop BDP through the backpressure rate control of per-flow and per-hop. HPCC adjusts the speed MIMD through the INT information of the switch. PCN [58], Homa [59] rely on the receiver to send credit packets to determine the sending rate of each flow. TIMELY [6] and Swift [8] are RTT-based schemes to adjust the flow rate at end host. These approaches achieve remarkable performance. However, they require changes to the network stack that are not easy to implement on older hardware devices.

8 Conclusion

This paper revisits the RED parameter configuration and proposes the design of PRED by capturing the key features of datacenter traffic dynamics in terms of flow concurrency and flow distributions. Adaptively and effectively, PRED achieves automatic adjustment of RED parameters stably. Through evaluations, we demonstrate that PRED is more stable than ACC [17], the state-of-the-art learning-based approach. Without any modifications at the end host, and with a more stable RED parameter configuration that adapts well to traffic dynamics, as the future work, we believe PRED has good potential to be deployed in modern production datacenters.

Acknowledgements

We are thankful to the anonymous NSDI reviewers for their constructive feedback. This work is supported in part by the China National Funds for Distinguished Young Scientists (No.62425201), the NSFC Projects (No.62472247, No.62202473, No.61932016, No.62132011, and No.62221003), the CCF-Huawei Populus Grove Fund, and the fund for building world-class universities (disciplines) of Renmin University of China.

References

[1] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE MSST*, pages 1–10, 2010.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM*, pages 63–74. ACM, 2010.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[5] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

[6] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[7] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In Jianping Wu and Wendy Hall, editors, *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 44–58. ACM, 2019.

[8] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*, pages 514–528. ACM, 2020.

[9] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *USENIX NSDI*, pages 779–805. USENIX Association, 2022.

[10] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve D. Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 399–413. ACM, 2019.

[11] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, and Soudeh Ghorbani. A large-scale deployment of DCTCP. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 239–252, Santa Clara, CA, April 2024. USENIX Association.

[12] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 230–243. ACM, 2016.

[13] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Gu, Wes Felter, John Carter, and Aditya Akella. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 244–257. ACM, 2016.

[14] Yiran Zhang, Yifan Liu, Qingkai Meng, and Fengyuan Ren. Congestion detection in lossless networks. In *ACM SIGCOMM*, pages 370–383. ACM, 2021.

[15] Junxue Zhang, Wei Bai, and Kai Chen. Enabling ecn for datacenter networks with rtt variations. In *ACM CoNext*, pages 233–245. ACM, 2019.

[16] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. *IEEE/ACM Transactions on Networking*, 29(2):489–502, 2020.

[17] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. Acc: Automatic ecn tuning for high-speed datacenter networks. In *ACM SIGCOMM*, pages 384–397. ACM, 2021.

[18] K Ramakrishnan, Sally Floyd, and D Black. Rfc3168: The addition of explicit congestion notification (ecn) to ip, 2001.

[19] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, 1(4):397–413, 1993.

[20] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[21] Glenn Judd. Attaining the promise and avoiding the pitfalls of tcp in the datacenter. In *USENIX NSDI*, pages 145–157. USENIX Association, 2015.

[22] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *ACM SIGCOMM*, pages 66–85. ACM, 2022.

[23] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 30, 2016.

[24] Tofino product family brochure. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/tofino-product-family-brochure.pdf. 2022.

[25] The P4 Language Consortium. $p4_{16}$ language specification version 1.0.0. 2016.

[26] Network simulator 3. (2019). https://www.nsnam.org/. 2019.

[27] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *ACM CoNext*, pages 313–327. ACM, 2016.

[28] Sally Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[29] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *ACM Queue*, 14(5):20–53, 2016.

[30] Bruce Spang and Nick McKeown. On estimating the number of flows. In *Stanford Workshop on Buffer Sizing*, 2019.

[31] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *USENIX NSDI*, pages 395–408. USENIX Association, 2015.

[32] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *USENIX NSDI*, pages 343–356. USENIX Association, 2018.

[33] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. In *ACM SIGCOMM*, pages 615–631. ACM, 2020.

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

[35] Dctcp in linux kernel. https://www.kernel.org/doc/html/latest/networking/dctcp.html. 2022.

[36] Dctcp in ns3 simulation. https://www.nsnam.org/docs/release/3.36/models/html/tcp.html?highlight=dctcp. 2022.

[37] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[38] Wei Bai, Kai Chen, Li Chen, Changhoon Kim, and Haitao Wu. Enabling ecn over generic packet scheduling. In *ACM CoNext*, pages 191–204. ACM, 2016.

[39] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/. 2022.

[40] Traffic generator. https://github.com/HKUST-SING/TrafficGenerator. 2022.

[41] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ecn in multi-service multi-queue data centers. In *USENIX NSDI*, pages 537–549. USENIX Association, 2016.

[42] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *ACM SIGCOMM*, pages 51–62. ACM, 2009.

[43] Piotr Gawłowicz and Anatolij Zubow. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *ACM MSWiM*. ACM, 2019.

[44] Wuchang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. Techniques for eliminating packet loss in congested tcp/ip networks. Technical report, Citeseer, 1997.

[45] W-C Feng, Dilip D Kandlur, Debanjan Saha, and Kang G Shin. A self-configuring red gateway. In *IEEE INFOCOM*, volume 3, pages 1320–1328, 1999.

[46] Sally Floyd, Ramakrishna Gummadi, Scott Shenker, et al. Adaptive red: An algorithm for increasing the robustness of red's active queue management, 2001.

[47] Srisankar Kunniyur and Rayadurgam Srikant. Analysis and design of an adaptive virtual queue (avq) algorithm for active queue management. *ACM SIGCOMM Computer Communication Review*, 31(4):123–134, 2001.

[48] Honggang Zhang, Don Towsley, CV Hollot, and Vishal Misra. A self-tuning structure for adaptation in tcp/aqm networks. In *ACM SIGMETRICS*, pages 302–303. ACM, 2003.

[49] Chris V Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. On designing improved controllers for aqm routers supporting tcp flows. In *IEEE INFOCOM*, volume 3, pages 1726–1734, 2001.

[50] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *ACM SIGCOMM Computer Communication Review*, 34(4):281–292, 2004.

[51] Bruce Spang. Updating the theory of buffer sizing. *ACM SIGMETRICS Performance Evaluation Review*, 49(3):55–56, 2022.

[52] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. TACK: improving wireless transport performance by taming acknowledgments. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 15–30. ACM, 2020.

[53] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[54] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ecn for data center networks. In *ACM CoNext*, pages 25–36. ACM, 2012.

[55] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):73–84, 2011.

[56] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, pages 253–266. USENIX Association, 2012.

[57] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[58] Wenxue Cheng, Kun Qian, Wanchun Jiang, Tong Zhang, and Fengyuan Ren. Re-architecting congestion management in lossless ethernet. In *USENIX NSDI*, pages 19–36. USENIX Association, 2020.

[59] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*, pages 221–235. ACM, 2018.

Variable	Description
w	Window size
N	Flow concurrency level
α	Estimated congestion degree of DCTCP
g	DCTCP's parameter
t	Time
q	Bottleneck queue length
RTT	Round-trip time (RTT)
C	Bottleneck link capacity
d	Propagation delay

Table 1: Variables of fluid model.

A Problem Formulation

We take DCTCP as an example of the RED parameter breakdown based on a fluid model, as in paper [3, 5]. Table 1 summarizes the main notations in this paper. Note that we can also analyze other sources (e.g., DCQCN and TCP) in a similar way [27, 28], since the conclusions remain similar, we omit them here for conciseness.

In a fluid model, we consider N long-lived flows traversing a single bottleneck link with capacity C. The following nonlinear, delay-differential equations describe the dynamics of the window size $w(t)$, the DCTCP's estimated congestion degree $\alpha(t)$, and the queue size $q(t)$ at the switch:

$$\frac{dw}{dt} = \frac{1}{RTT(t)} - \frac{w(t)\alpha(t)}{2RTT(t)} p(t - RTT^*) \tag{2}$$

$$\frac{d\alpha}{dt} = \frac{g}{RTT(t)} (p(t - RTT^*) - \alpha(t)) \tag{3}$$

$$\frac{dq}{dt} = N\frac{w(t)}{RTT(t)} - C \tag{4}$$

Here $p(t)$ indicates the RED (ECN) marking process at the switch and is given by:

$$p(t) = \begin{cases} 1, & q(t) > \mathsf{maxK} \\ \frac{q(t)-\mathsf{minK}}{\mathsf{maxK}-\mathsf{minK}}\mathsf{maxP}, & \mathsf{minK} < q(t) <= \mathsf{maxK} \\ 0, & q(t) <= \mathsf{minK} \end{cases} \tag{5}$$

and $RTT(t) = d + q(t)/C$ is the RTT, where d is the propagation delay (assumed to be equal for all flows), and $q(t)/C$ is computed as the queueing delay.

Equations (2) and (3) describe the congestion window adjusting scheme of DCTCP at the source, and Equation (4) describes the queuing process at the switch. The source and the switch are coupled through the packet marking process $p(t)$. This feedback delay is approximately a fixed value $RTT^* = d + q_{avg}/C$.

Equation (2) models the window evolution and consists of the standard additive increase term, $1/RTT(t)$, and a multiplicative decrease term, $-w(t)\alpha(t)/2RTT(t)$. The latter term models the source's reduction of window size by a factor $\alpha(t)/2$ when packets are marked. Equation (3) is a continuous approximation of DCTCP's estimated congestion degree. Equation (4) models the queue evolution: $N\frac{w(t)}{RTT(t)}$ is the network input rate and C is the service rate.

By setting the LHS (left hand side) of Equations (2)-(4) to zero, we see that any fixed points of the DCTCP (if they exist) must satisfy:

$$1 - \frac{\dot{W}\dot{\alpha}}{2}\dot{p} = 0, \dot{p} - \dot{\alpha} = 0, N\frac{\dot{W}}{d + \frac{\dot{q}}{C}} - C = 0 \tag{6}$$

At any of the fixed points, we assume the value of p is $\dot{p}$, which is shared by all flows, and we can get:

$$\dot{p} = \frac{\dot{q} - \mathsf{minK}}{\mathsf{maxK} - \mathsf{minK}}\mathsf{maxP} = \lambda(\dot{q} - \mathsf{minK}) \tag{7}$$

Combining Equations (6) and (7), we eliminate the variable $\dot{p}$, $\dot{\alpha}$ and $\dot{W}$. After simplification, we see that the value of $\dot{q}$ is determined by:

$$(\dot{q} - \mathsf{minK})^2(\dot{q} + Cd) = \frac{2N}{(\frac{\mathsf{maxP}}{\mathsf{maxK}-\mathsf{minK}})^2} = \frac{2N}{\lambda^2} \tag{8}$$

From Equation (8), we can see that the steady-state $\dot{q}$ is only related to three factors. The first factor is related to bottleneck link C and feedback delay d. Once the network topology is built, this factor will be fixed. The second factor is the flow concurrency level N, and the larger N is, the larger $\dot{q}$ is. Due to the dramatic change of the flow concurrency N, the steady-state $\dot{q}$ will fluctuate sharply, which will affect the stability of the network. The last but most important factor is the settings of the RED parameter, i.e., λ and minK. We thus infer that the essence of configuring RED parameters is to adjust the steady-state queue length.

B Additional Discussion

Computing and storage overhead. The computing and storage requirements are minimal. FCS is designed to be pipelined on the data plane, allowing all steps to execute in parallel, regardless of the size of the topology or the number and types of present flows. QLA involves a simple comparison of several values, imposing small computational overhead. In terms of storage requirements, each port requires only a few registers to cache the traffic count N and a corresponding bitmap. For example, each port uses 128 bits, N uses 8 bits, and the remaining 120 bits are used as bitmap.

Highly dynamic traffic pattern. If the network is highly dynamic, PRED will tend to be stable, rather than a faster convergence or faster traffic prediction. In a highly dynamic network, the QLA will filter out the noise, and PRED will adjust the parameters only when there is a positive reward in the statistics within T_{QLA} time. § D.5 shows that PRED

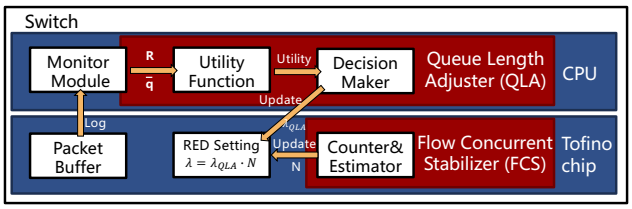


Figure 20: Hardware architecture of PRED on Tofino.

requires at least 200 ms to fully adapt to a purely short-flow load from a purely high-flow load.

Overestimating N or underestimating N. If the true N exceeds the capacity of the estimator, then N will be underestimated; If a lot of bursts come, then N will be overestimated. In this case, the FCS flow estimate will be biased, but the noisy N will still be better than the static threshold. As can be seen in § 6.3.2, PRED can quickly respond to changes in N, resulting in faster congestion adjustment and feedback even if inaccurate. In addition, QLA can also compensate for long-term estimation inaccuracies, so that PRED still keeps the switch statistically high performance.

C Additional Implementation

Figure 20 shows the hardware architecture of PRED on the Tofino switch.

C.1 Data Plane Modification

Counter & Estimator Module. PRED needs to know how many flows are passing through the port of the switch at a time interval. As described in § 4.1, switches only need a simple bitmap and counters to implement this module. However, each stage of the switch can operate on only one register, not an array of bitmap registers. Thus, the switch cannot bulk reset the bitmap registers periodically on the data plane. The control plane can reset the registers of the data plane, but it cannot be used because of the high API delay (>1 ms).

To solve this problem, PRED uses a time interval sequence counter and a write-after hashing operation to calculate the number of concurrent flows. Algorithm 1 shows the FCS data plane processing logic. When a packet $\mathcal{P}$ arrives, the switch first checks whether the current time *now* has been more than an interval T_{FCS} since the last update T_{start} (line 1). If the time is over this interval, some registers need to be updated and reset (lines 2-5). If not, the packet $\mathcal{P}$ needs to be checked to see whether it belongs to a new flow (lines 7-11).

We first analyze the reset process after the timeout (lines 2-5). T_{start}, n_{last}, n_{now}, $Interval_{seq}$ and $Bitmap$ are all Registers. T_{start} is the beginning of the interval (line 2) and needs to be set to the current time *now* after the timeout occurs. n_{last} is the flow number collected in the last interval (line 3), which should be set to n_{now} after the timeout. n_{now} is the flow number

Algorithm 1 Data Plane Processing Logic.

1: **if** $now - T_{start} > T_{FCS}$ **then**
2: $\quad$ UpdateReg$_{T_{start}}(now)$ ▷ Next time interval
3: $\quad$ UpdateReg$_{n_{last}}(n_{now})$ ▷ Record the last interval flow number
4: $\quad$ UpdateReg$_{n_{now}}(0)$ ▷ Reset the current interval flow number
5: $\quad$ UpdateReg$_{Interval_{seq}}(+1)$ ▷ Update the interval sequence
6: **else**
7: $\quad H_{new} \leftarrow$ Hash$(Packet5Tuple, Interval_{seq})$ ▷ Derive flow hash from packet 5-tuple and interval sequence
8: $\quad$ S$_{\text{id}} \leftarrow H_{new}[IndexSize : 0]$ ▷ Derive storage index
9: $\quad H_{old} \leftarrow$ Read&UpdateReg$_{Bitmap}$(S$_{\text{id}}, H_{new}$) ▷ Get the old hash value from the bitmap and set the new value
10: $\quad$ **if** $H_{new} != H_{old}$ **then** ▷ New flow is detected
11: $\quad\quad$ UpdateReg$_{n_{now}}(+1)$
12: $F_{Num} \leftarrow MAX(n_{last}, n_{now})$
13: $Prob \leftarrow ReadTable_{PRED}(q, F_{Num})$
14: **if** $Prob > Random()$ **then**
15: $\quad MarkECN(\mathcal{P})$

collected in the current interval (line 4), which needs to be set to 0 after the timeout. $Interval_{seq}$ is the serial number of the time interval (line 5), which is used to distinguish different time intervals. $Bitmap$ is a hash table of length $2^{IndexSize}$.

Next, when there's no timeout (lines 7-11), we analyze the most critical process of counting flow numbers. The packet $\mathcal{P}$'s flow hash H_{new} is computed by hashing over the packet's 5-tuple and interval sequence number $Interval_{seq}$ (line 7). Then the last $IndexSize$ bits of the hash value is used as the storage index S_{id} (line 8). The most crucial step is to read the stored flow hash H_{old} in the S_{id} position in the hash table $bitmap$ and overwrite the H_{new} in the corresponding position (line 9). Then PRED checks whether the values of H_{new} and H_{old} are equal and if they are not, then the new flow is detected (lines 10-11). The reason for using interval sequence number $Interval_{seq}$ for packet hashing is that we want flows across different time intervals to be recounted at different time intervals. To this end, the same flow must have different hash values at different time intervals. With the above steps, PRED does not need to bulk reset the value in the bitmap register periodically on the data plane because the new hash value is overwritten each time. Two conflicts may affect the flow count, i.e., hash conflict (different hash parameters but same hash value) and position conflict (different hash value but same S_{id}). The hash conflict causes conflicting flows to be viewed as one flow, and the position conflict causes conflicting flows to alternately overwrite the same register, bringing unusually high flow counts. However, the chances of the two conflicts are low, and thus they have little effect on the performance.

RED Setting Module. Because the P4 switch has no direct division to use for the RED Setting Module, here PRED programs the RED algorithm into the PRED table (line 13). The

table's key is the queue length and the flow number, and the table's value is the marking probability. After obtaining the marking probability, PRED compares it with the random value. If the marking probability is greater than the random value, ECN labeling will be tagged (lines 14-15).

Packet Buffer Module. The packet buffer module provides some queue information to the control plane, including $\bar{q}$ and R. R can be obtained directly from switch statistics. The instantaneous queue length cannot match the performance of the switch. Therefore, PRED samples the queue length multiple times, sum it and saves it in the register. The control plane can read the register to get the sum value and the sampling times to infer the average queue length.

C.2 Control Plane Modification

The QLA module needs at least 6 additional stages in the Tofino switch if it is to be deployed in the data plane. We have implemented PRED where both QLA and FCS are deployed on the data plane, but a total of 17 stages are needed. In § 4.2, we introduced that the QLA module needs to save four utility function values to judge the adjustment of λ_{QLA}. These operations require at least 3 stages, two for reading/writing the register and one for comparison operation. Since the RED table requires additional Key entries to identify the λ_{QLA}, the additional stages need at least 3 stages. For pure FCS modules, 11 stages have been used, so the existing Tofino switches do not support deploying both modules to the data plane at the same time. However, the new Tofino 2 and Tofino 3 switches support 20 stages [24], so the QLA and FCS modules can be deployed in the data plane at the same time.

In this paper, we deploy the QLA module on the control plane. The Monitor Module subscribes raw data from the Register for features analysis, including the total bytes sent and egress queue depth. In detail, at each time interval T_{QLA}, the Monitor Module achieves the subscribed data from forwarding chips's registers. Then, the utility function calculates the current utility value and passes it to the Decision Maker. The control module makes a judgment every 4 intervals T_{QLA} and sets the new RED setting to the data plane.

Discussion. There is a problem with implementing the QLA module in the control plane. Reading the register in the control plane takes 5 to 12 ms, and updating the Red table takes 80 to 110 ms. Therefore T_{QLA} is at least 200 ms due to hardware limitations. We discuss the setting of T_{QLA} in § D.5. The larger T_{QLA} is, the slower convergence will be. Therefore, if the implementation is upon Tofino 1, PRED has a slower convergence rate. However, we believe that if implemented upon Tofino 2 or Tofino 3, this problem would not occur.

D Additional Simulation Results

D.1 PRED with DCQCN

Setup: We use a simple 20-to-1 topology with 40 Gbps links, 20 servers are senders, and 1 receiver. The latency of the link is 2 μs. The parameters are set as suggested in [5, 27]. The NS-3 simulation implementation of DCQCN refers to the implementation in [27].
Results: Figure 21 shows the PRED performance with DCQCN. Figure 21(a) shows the average queue length and 99th queue length as N increases. The queue length for both RED (0.2) and RED (2) increases with N, and PRED avoids this problem. Figure 21(b) shows the time sequence diagram when N=2 and N=20. PRED at low concurrency (N=2), queue length changes close to RED (0.2) to ensure throughput. At high concurrency (N=20), PRED is close to RED (2) to ensure low latency. To sum up, PRED can also cooperate well with DCQCN.

D.2 Large Scale Simulation in DataMing workload

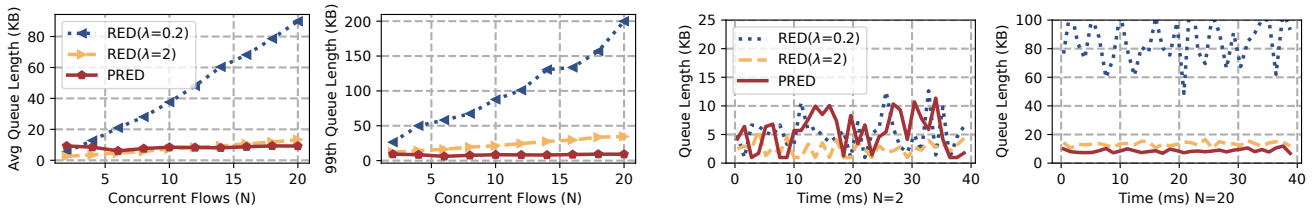
Figure 22 is the result under the DataMining workload. For details, refer to § 6.3.1. As shown in Figure 22, PRED's small flow 99th FCT (0.76 ms) has a reduction ranging from 58% (ECNSharp, 1.83 ms) to 86% (CODEL, 5.28 ms) compared to ECN algorithms at 90% load. For Large flows, there was a negligible increase in the FCT of PRED (44 ms) ranging from 6.3% (ECN, 41.36 ms) to 8% (ECNsharp, 40 ms) compared to ECN algorithms at 90% load. This proves that PRED can outperform other ECN algorithms under the DataMining workload.

D.3 100 Gbps Large Scale Simulation in WebSearch workload

Figure 23 is the result under the WebSearch workload. For details, refer to § 6.3.4. As shown in Figure 23, PRED's small flow 99th FCT (142 μs) has a reduction ranging from 27% (RED, 171 μs) to 59% (CODEL, 350 μs) at 80% load. PRED's small flow average FCT (29 μs) has a reduction ranging from 52% (RED, 61 μs) to 75% (CODEL, 117 μs) at 80% load. This proves that PRED can outperform other ECN algorithms in 100 Gbps with 1000s concurrent flows.

D.4 Packet Scheduler

Setup: We configure the switch with Deficit Weighted Round Robin (DWRR) with 3 queues/services. The weights among the 3 queues/services are 2: 1: 1. We first start a long-lived TCP flow from sender 1 and classify this flow into queue/service 1, then from sender 2 and classify it into queue/service 2,



(a) Different flow concurrency levels (b) Breakdown

Figure 21: [Simulation] PRED performance with DCQCN.

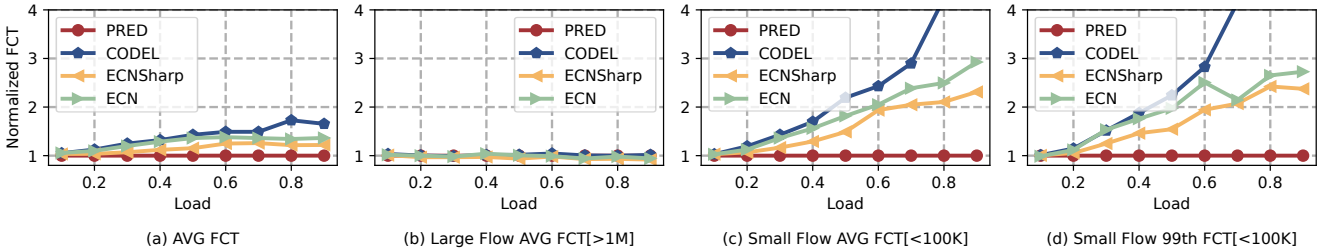


Figure 22: [Simulation] FCT statistics with different ECN algorithm. All are normalized to the results achieved by PRED (DataMining Workload).

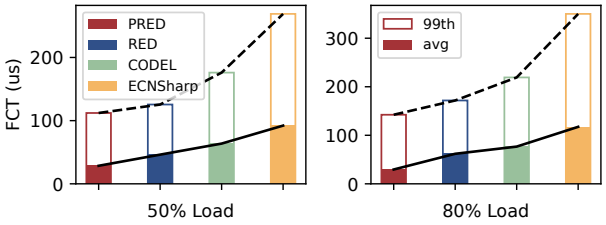
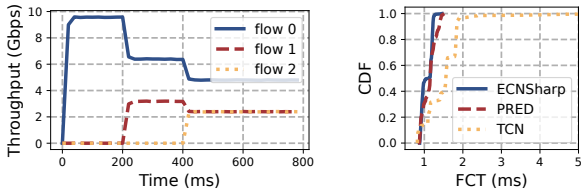


Figure 23: [Simulation] Small Flow FCT statistics with WebSearch workload.



(a) Flow throughput (b) FCT of small flows

Figure 24: [Simulations] PRED with packet schedulers.

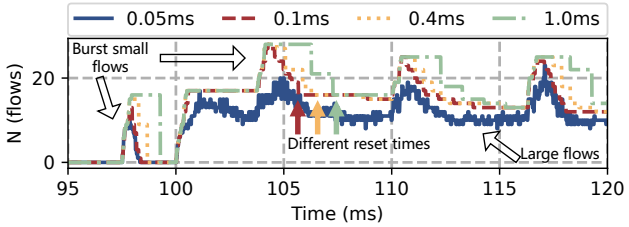


Figure 25: [Simulation] N Estimator of Different T_{FCS}. (RTT is 0.08 ms)

and finally from sender 3 into queue/service 3. We also randomly start small flows (the size is from 3 KB to 60 KB) from the rest of the senders to probe the queue occupancies. We set the TCN marking threshold to 150 μs to avoid throughput loss. The other settings are identical to §6.3.1.

Results: Figure 24(a) shows the throughput of flows achieved by PRED. We observe at the beginning that only queue 1 is active, flow 1 achieves around 9.6 Gbps. After flow 2 starts, queue 2 becomes active, and flow 1 achieves 6.42 Gbps while flow 2 achieves 3.18 Gbps. Finally, when three queues all become active, flow 1, flow 2 and flow 3 achieve around 4.82 Gbps, 2.40 Gbps and 2.40 Gbps goodput, respectively, which strictly preserves the packet scheduling policy.

We also measure the FCT of small flows among all queues, and the results are shown in Figure 24(b). PRED is similar to ECNsharp and superior to TCN in small flow. In summary, PRED can strictly preserve the packet scheduling policy with multiple queues/classes.

D.5 Parameter Design Choice

We focus here on the two most important time parameters T_{FCS} and T_{QLA} in PRED, and we will discuss the other parameters in future work. We analyzed how to set these in § 4, and we verified these by experiment in this section.

Setup: The topology, parameter settings, and traffic patterns are all similar to the previous subsection.

Results: Figure 25 shows the estimation of flow number under different T_{FCS}. The burst state and the stationary state of the

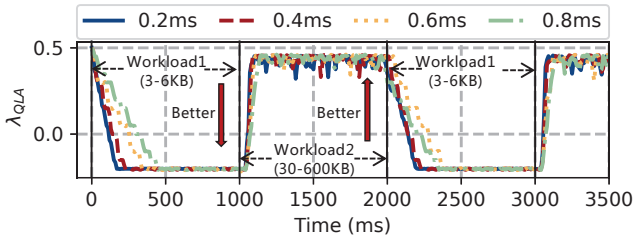


Figure 26: [Simulation] λ_{QLA} of Different T_{QLA}.

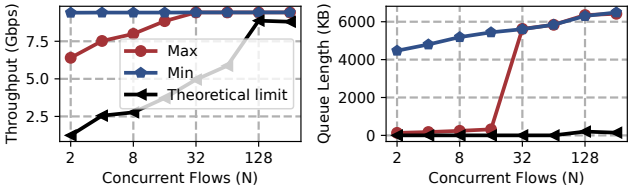


Figure 27: [Testbed] Range of PRED adjustment capabilities.

large flows both have 18 flows. When burst flows and large flows are concurrent, there should be 36 flows. From figure 25, when $T_{FCS} < RTT$, PRED cannot accurately estimate the correct N. Because all the traffic does not pass through the port completely. When $T_{FCS} > RTT$, PRED can estimate the correct N, but with the increase of T_{FCS}, the reset time of N is also increasing. So T_{FCS} should be slightly larger than RTT, and our rule of thumb is 1.25RTT.

Figure 26 shows the convergence of λ_{QLA} for different T_{QLA}. The workload 1 (3-6 KB) ranges from 0 to 1,000 ms and from 2,000 to 3,000 ms. Workload 2 (30-600 KB) is 1,000-2,000 ms. In workload 1, all flows are small, so the smaller the λ, the smaller the FCT. Therefore, it can be seen that PRED can rapidly reduce λ when 0-500 ms, while it can rapidly increase λ when 1,000 ms. And the smaller T_{QLA} is, the faster it converges, but the more unstable. Therefore, T_{QLA} is recommended to be 5RTT.

D.6 Range of PRED Adjustment Capabilities

We tested the range of PRED adjustment capabilities using Testbed. As shown in Figure 27, the throughput and queue length change with the number of concurrent N. The blue line indicates the parameter setting with the minimum λ and the lowest ECN probability, and we can see that its queue length is large and its throughput is high.

The red line indicates the parameter setting with the maximum λ and the maximum ECN marking probability. It can be seen that when $N < 32$, there is a loss in throughput and the queue length is small. When $N > 32$, the parameter Settings in the adjustment range cannot limit the stability of the queue. This means that PRED limits the number of concurrent requests to no more than 32 within the range of optional parameters. However, in practice, there are not that many concurrent long flows in the datacenter [3], and the range of PRED is sufficient.

The black line represents per-packet ECN, that is, every packet that enters the switch is marked with an ECN. It can be seen that the adjustment range of the black line is very large. However, the adjustment granularity is too large, and the change of very small parameters will lead to drastic changes in the queue and throughput, so the adjustment range of PRED is only between the red line and the blue line.

Rajomon: Decentralized and Coordinated Overload Control for Latency-Sensitive Microservices

Jiali Xing[1], Akis Giannoukos[1], Paul Loh[1], Shuyue Wang[1], Justin Qiu[1], Henri Maxime Demoulin[2], Konstantinos Kallas[3], and Benjamin C. Lee[1]

[1]University of Pennsylvania, USA. [2]DBOS, Inc, USA. [3]University of California, Los Angeles, USA.

Abstract

Microservices are increasingly central for cloud applications due to their flexibility and support for rapid integration and deployment. However, applications often experience overload or sudden traffic surges that exceed service capacity, resulting in increased latency or service failures. Moreover, microservices are decentralized, interdependent, and multiplexed, exacerbating risks from overload.

We present RAJOMON, a market-based overload control system for large microservice graphs. RAJOMON controls overload through distributed rate-limiting and load shedding. Clients attach tokens to requests and services charge a price for each API, dropping requests with insufficient tokens. Tokens and prices propagate through the entire call graph, piggybacking on requests and responses. Thus, RAJOMON is the first decentralized, end-to-end overload control system.

We implement and evaluate RAJOMON on a setup of up to 140 cores and on a variety of applications from academia and industry. Experiments indicate RAJOMON protects microservice goodput and tail latency from substantial demand spikes, even in the case of mixed request types and deeper service graphs. For high-load scenarios, RAJOMON reduces tail latency by 78% and increases goodput by 45% when compared against state-of-the-art overload control for microservices.

1 Introduction

Microservices enable extensible and flexible software architectures, advantages that have led to their adoption in many large, production systems [6, 7]. Effectively managing overload, where demand exceeds the system's capacity for computation, is essential for performance and reliability. Demand spikes might arise from expected events, such as sales or holidays, or unpredictable variations in client behavior. Supply inefficiencies might arise from hardware issues, such as power throttling, or software issues such as system misconfiguration or garbage collection. Overload can severely degrade user experience and system reliability [3, 52]. It is the principal culprit for cascading failures within service graphs [2].

Microservice applications, compared to monolithic systems, have several characteristics that make overload control more challenging. First, microservices are deployed automatically and managed in a decentralized manner [19]. Second, microservices are highly interdependent, creating the risk that congestion at one node can propagate along the execution path, turning localized overload into cascading failures [13]. Finally, microservice nodes are often multiplexed, serving multiple request interfaces. Each interface may involve different downstream call graphs but share the same nodes, making it difficult to detect which interfaces are overloaded [24, 25].

An effective, responsive overload control framework should incorporate three elements: (1) decentralized detection and decisions to improve responsiveness; (2) coordination across the call graph to control load early in the execution path, ideally at client side; and (3) the ability to distinguish and manage overload at interface granularity.

Current systems handle request surges with load shedding and rate limiting (Table 1), but they do not provide the key elements for effective overload control. Centralized approaches struggle to respond to overload in a timely manner because they rely on extensive telemetry and computation [34]. Without coordination across the call graph, many requests are processed by some nodes and dropped by others, wasting computational resources and lengthening queues. [52]. Without interface granularity, overloaded and idle interfaces are treated equally, causing head-of-line blocking and dropping requests [3]. These limitations hinder the practical deployment of previously proposed frameworks.

We respond to these challenges and propose RAJOMON to achieve key desiderata with tokens and price tables. Clients are granted *tokens* at predefined rates, and these tokens are attached to each request and its sub-requests. Each microservice maintains a table of *prices*, which specify the number of tokens required for an interface. First, RAJOMON decentralizes overload control as clients rate limit themselves when their token holdings are insufficient while services reject requests with insufficient tokens. Second, RAJOMON coordinates control across the microservice graph without extra

Frameworks	Decentralization	Coordination	Client Rate Limiting	Interface Granularity	Time Scale
Dagor	✓	Pair-wise	✗	✓	1s
Breakwater	✗	✗	✓	✗	< 1ms
BreakwaterD	✓	✗	✓	✗	< 1ms
TopFull	✗	✓	✗	✓	10s
Rajomon	✓	✓	✓	✓	< 1s

Table 1: Related work of microservice overload control, with time scales represented in seconds (s) and milliseconds (ms).

network overheads. When a service experiences overload, it increases prices and attaches updated prices to responses, thereby informing upstream services and end-users about congestion, who then proactively limit new requests before they are issued to overloaded execution paths. Finally, prices are updated independently for each interface based on their downstream interfaces, ensuring that overloaded interfaces are properly throttled while idle ones remain unaffected.

RAJOMON is designed and implemented as a gRPC interceptor package that does not modify application code [5]; we will release the source code. We evaluate RAJOMON using service graphs, with varied depths and sizes, from academic benchmarks and industrial traces. Compared to state-of-the-art baselines [3, 34, 52], RAJOMON improves goodput (*i.e.*, throughput within latency requirements) by 117–266% and reduces latency by 33–46% under challenging system conditions. When traffic surges, RAJOMON goodput recovers within a second, four times faster than baselines. RAJOMON's differentiates between concurrent requests across different interfaces, increasing goodput by 45–245% and reducing latency by 78–94%.

2 Motivation

Overload occurs when computational demands exceed system capacity, causing requests to queue, violate service-level obligations, or crash the service. For instance, the release of Taylor Swift's 'Midnights' album drew overwhelming traffic and caused a global outage of Apple Music and Spotify [15]. Another example is Coinbase's Super Bowl ad, which crashed its own website [37]. Such outages cause financial losses and harm user experience; 61% of users immediately turn to competitors when they have difficulty accessing websites [17].

We motivate an overload control framework that targets microservice-based applications, detailing their characteristics that lead to unique challenges. We address these challenges by specifying desiderata for the control framework. We also describe limitations of existing solutions.

2.1 Microservices

An application structured using microservices breaks down functionality into separate components, each of which is implemented in its own service [13, 25]. Each service provides an interface (API) that supports specific requests and communicates with other services, through remote procedure calls (RPCs), to perform an end-to-end task. By tracking services invoked for the task, we can construct a call graph. Several properties of the microservice computation – decentralized operation, interdependent requests, multiplexed services – pose significant challenges for overload control.

Decentralized Operation. Microservices are loosely coupled, through RPCs, and each is usually implemented, maintained, and managed by a different organization or team [19]. This decentralized structure makes traditional, monolithic overload control methods impractical because there is no single entity that has ownership or visibility over all services. Furthermore, large microservice applications tend to include hundreds or even thousands of services [25]. At this scale, even distributed tracing incurs significant delays. For instance, Dapper takes up to 15 seconds to propagate trace data, while Jaeger also operates asynchronously, adding further delay [11, 36]. As a result, centralized control mechanisms are unable to respond quickly enough because they first need to observe the state of all services.

Interdependent Requests. Microservice-based applications have graph structures with complex dependencies. According to Alibaba, 30% of graphs are wide and exhibit a fan-out pattern in which a service calls many others [24]. Many other graphs are deep and exhibit long chains of services. An overloaded service can cause cascading quality-of-service violations throughout the graph [13]. When an upstream node exhibits increased queuing delay, request queues have already lengthened in downstream nodes [25].

Multiplexed Services. Services are often multiplexed and shared by multiple applications and types of requests, which invoke different application programming interfaces (APIs). Approximately 90% of call graphs contain multiplexed services that are included in other call graphs and interact with various upstream and downstream services [24, 26]. With such multiplexing, one type of request might overload a shared service, causing head-of-line blocking and affecting the processing of other non-overloaded interfaces [25]. On the other hand, overload control actions suitable for one interface could be too conservative and starve other interfaces [34]. Determining the optimal rate for concurrent APIs given the varying capacities of downstream servers is challenging.

2.2 Control Desiderata

We design and implement RAJOMON to address the unique challenges from microservices. Our framework supports three key properties for effective overload control.

Decentralized Decisions. Every client rate limits and every service sheds load independently. Decentralized decisions require exchanging and propagating load signals. Yet no service

blocks on communication or commands from other services, ensuring timely reaction to overload.

Coordinated Control. Services propagate load metadata, recursively and eventually, to upstream services and clients. Metadata flows through the microservice graph based on paths taken by varied request types. Thus, services are aware of downstream overload and proactively reduce the rate of upstream requests. Such collaborative load shedding reduces wasted computation throughout the graph.[1]

Interface Granularity. Clients and services monitor and control overload for each interface separately. Control at interface granularity is more efficient than prior approaches, which operate at client granularity [3], because overload often occurs at specific interfaces rather than broad services or certain clients. RAJOMON applies rate limits and active queue management differently for each request type. Its approach can be viewed as *virtual multi-queues* for head-of-line blocking in the microservice context.

2.3 Existing Approaches

There exist several established approaches for overload control, but none adequately address the challenges inherent to microservice-based applications, as shown in Table 1.

Dagor implements decentralized overload control [52]. Each server monitors its queuing delay to determine the admission level. Dagor attaches a priority score to each request and expresses admission level as a priority cut-off. Each server drops requests that fail to meet this cut-off. Services communicate admission levels to their senders so that load is shed one tier before hotspots within the graph.

Breakwater implements single-layer overload control to manage interactions between one server and its clients [3]. The front-end service adjusts total credits based on queuing delays and then issues credits to each client to explicitly control the number of requests they can send. The front-end increases server utilization by over-issuing credits and then dropping requests when the excess of credits overload the server.

TopFull is an adaptive overload control framework for microservice applications that operates in a top-down manner [34]. It dynamically controls API load by adjusting request rates at the front-end. It first clusters APIs according to their graph structure and shared nodes. For each cluster, reinforcement learning throttles requests at API granularity based on observed end-to-end goodput and latencies.

Empirical Comparison. For motivation, Figure 1 presents throughput, dropped requests, and total demand for the *Compose Post* request in a social network application. Initially, all

[1]Note that Dagor also implements collaborative load shedding, but does so between single sender-receiver pairs, which limits scalability to larger applications [52].

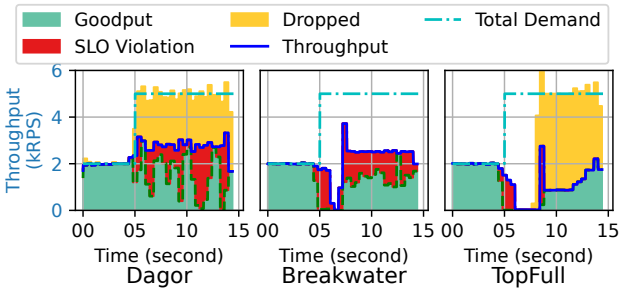


Figure 1: Existing OC approaches failed to handle the overload.

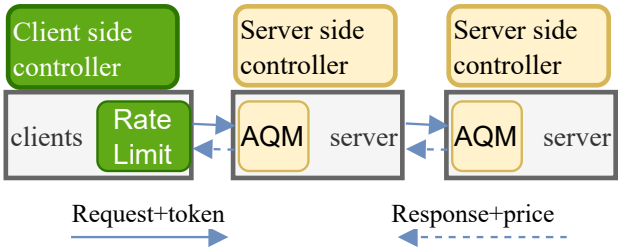


Figure 2: RAJOMON architecture.

frameworks are able to match goodput to total demand. But when demand spikes at 5 seconds, performance degrades.

Dagor lacks end-to-end coordination, shedding load only at nodes that precede hotpsots in the graph. This wastes resources and slows response time as many requests are processed by other services but are ultimately dropped. Breakwater, enabled only at the front-end, is unaware of real-time load conditions at downstream services such that overload causes service failure for 2 seconds. Moreover, Breakwater's decisions are misaligned with actual back-end capacities, leading to incorrect rate limits and causing half of requests to violate service-level obligations. TopFull's centralized reinforcement learning policy also experiences service failure for 4 seconds. Although it eventually recovers, communication and computation overheads delay system adaptation.

3 RAJOMON Design

Tokens function as a universal currency for computations requested by a sender (clients or upstream services) from a receiver (downstream services). Unlike conventional token bucket systems, RAJOMON's tokens accompany requests as they traverse the call graph. From the sender's perspective, tokens dictate each client's capacity to send requests. From the receiver's perspective, tokens signify each request's relative priority and enable collective agreement on which requests to shed first during overload (*i.e.*, those with fewer tokens).

RAJOMON employs tokens for two reasons. For clients, tokens permit differentiated rate limiting, which is important for microservices with multiple interfaces and diverse load dynamics at each interface. Clients should issue different

request types at different rates.

For services, tokens permit coordinated load shedding, which is important for efficiency. Applications often exhibit wide fan-outs in which one node in the call graph issues requests to and waits for multiple downstream nodes [24]. When load shedding is required, downstream nodes should coordinate and drop all sub-requests originating from the same request. Because all sub-requests must complete, the system need not compute for some when others are dropped. Tokens, attached to requests, provide the needed metadata about the original request and avoid additional communication for coordinated load shedding.

Prices represent the current demand and supply of computational resources at each service. Practically, prices set a threshold for the number of tokens required for a request's admission. Prices rise to curtail admissions during overload and fall when load eases. Prices propagate from downstream nodes toward clients, providing updated signals about system load to all services in the graph. A price is set for each interface. A table holds all prices for a service's interfaces.

Prices are necessary for sharing congestion information across nodes. Because each service interface calls different downstream services, its computational capacity depends on related downstream capacities. Determining whether a particular interface is overloaded requires callees that share their congestion levels with callers. Prices permit this coordination and decentralized overload control at interface granularity.

Controllers. Each node runs a local RAJOMON controller (Figure 2). The **Client-Side Controller** prevents clients from overwhelming the system by regulating the rate of outgoing requests. It generates tokens at a rate defined by its policy. Before a client issues a request, its controller ensures sufficient tokens are available for the corresponding interface. If not, it rate-limits and drops the request. Otherwise, it removes tokens from its pool and attaches them to the outgoing request. Periodically, the controller updates its price table based on the latest information received from the server.

The **Server-Side Controller** performs admission control, comparing tokens attached to incoming requests with the price of the requested interface. If tokens are insufficient, the request is dropped. Otherwise, the request is admitted and tokens are attached to any subsequent requests sent to downstream services. The controller periodically updates prices based on local queuing delays and feedback from downstream services to reflect the graph's current load conditions. Price updates are communicated back to upstream services and clients as part of the response metadata. Thus, controllers independently assess and manage load for each call path, enabling decentralized yet coordinated adaptive queue management.

3.1 Mechanisms for Overload Detection

The visibility of load conditions is limited because tasks require computation across many microservices loosely connected by RPCs. Measuring resource availability and service capacity is complicated by diverse graph topologies and request types. Simple indicators for overload in a single server, such as CPU utilization or queuing delay [3, 52], do not apply.

RAJOMON supports decentralized overload detection for loosely connected services. A service calculates its price for computation based on local queuing delays. Each of its interfaces sets its own price, allowing the system to track and manage congestion separately. Moreover, to capture end-to-end load conditions and quantify the total price a request must pay, prices accumulate from back-end services to clients.

When a service receives a price update from downstream callees, it updates the total price and shares with upstream callers. This backward propagation of prices informs clients and callers of the overload status for all downstream interfaces. Prices circulate asynchronously without requiring a centralized message broker.

3.2 Mechanisms for Overload Control

Each RAJOMON controller acts independently, making decisions about rate-limiting or adaptive queue management based on requests' attached tokens and interfaces' prices.

RAJOMON's control decisions account for call graph topology to minimize wasted computation. To reduce waste in parallel computation, which arises when one caller's requests fan out to multiple callees, RAJOMON uses token holdings to coordinate load shedding across downstream services. Overloaded services collaboratively shed sub-requests originating from the same caller's request, thereby avoiding unnecessary computation in parallel branches of the service graph.

To reduce waste in sequential computation, which arises when a client requests computation from a chain of microservices, RAJOMON controls load earlier in the call graph. Ideally, load is shed at clients or upstream services. By the time a request arrives at downstream services, its computation has already consumed significant service capacity and, for efficient goodput, should be completed if possible.

3.3 Policies for Client Control

Policies are devised to tune system responsiveness to overload while maintaining flexibility. These policies optimize the rate and granularity at which tokens and prices are updated.

Asynchronous Token Generation. The Client-Side Controller requires policies for token generation and spending, which should ensure stable throughput and smooth transitions between rate limiting and load shedding levels. RAJOMON uses a Poisson process to replenish tokens, which disperses token generation and client requests across time.

In contrast, generating tokens at some constant rate tends to produce bursty requests and oscillating goodput as clients simultaneously or synchronously replenish their token holdings. For example, when every client replenishes tokens every

10ms, they all send more requests at the beginning of the 10ms period. Randomizing the number of replenished tokens ameliorates this problem but does not fully resolve it.

Randomized Token Spending. RAJOMON selects a uniform random number of tokens for each outgoing request, yielding a range of tokens attached to requests such that the number of dropped requests increases gradually as the price increases. This policy provides a smoother reaction to overload and enhances system responsiveness.

Attaching a deterministic number of tokens to requests destabilizes throughput. A simple strategy spends all available tokens on the next request. But because clients may hold a similar number of tokens, the all-in strategy produces requests with similar numbers of tokens. Under this policy, token-based adaptive queue management is ineffective because the controller cannot distinguish priority across requests.

3.4 Policies for Server Control

Regularly updating and sharing prices are essential for timely overload control. Prices must reflect system dynamics and load conditions from nodes throughout the call graph.

Proportional Price Updates. The Server-Side Controller compares queuing delay against a threshold at regular intervals, determining whether price must be updated for a particular service interface. Threshold and interval are configurable parameters that typically range from 1ms to 20ms.

RAJOMON increases price by an increment that is proportional to queuing delay. The greater the queuing delay in excess of the threshold, the greater the price increase. Practically, when queuing delay exceeds the threshold by 1ms, the price increases by approximately 3 to 13 tokens. When queuing delay is less than half the threshold, the price decreases by 1 token. Otherwise, price is unchanged.

Traditionally, additive-increase / multiplicative-decrease (AIMD) algorithms adjust rates in token bucket mechanisms (*e.g.*, SEDA [41, 42]). AIMD divides admission rate by a constant factor (*e.g.*, two) during overload. But it throttles incoming requests by the same factor regardless of overload severity. AIMD cannot precisely control load.

Lazy Price Propagation. The Server-Side Controller sends prices to upstream nodes in the call graph with a lazy update policy. Lazy updates mitigate overheads associated with price updates. Because microservice functions and their corresponding RAJOMON controllers run on multiple threads, overheads could arise from ensuring thread safety for price updates. Because applications are implemented with tiers or layers of microservices, an overhead added to each layer can easily accumulate in deep call graphs and increase end-to-end latency. These overheads are modest when RAJOMON need not attach metadata to every response in the call graph. Upon sending a response, the controller attaches its price information with a configured probability, *e.g.*, 20%, updating the upstream services with the current local prices.

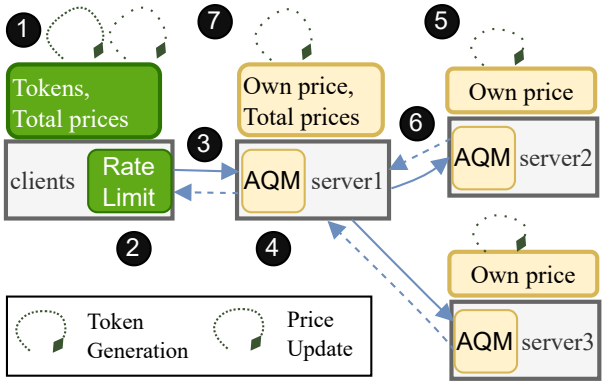


Figure 3: RAJOMON example.

Maximum Total Price. The Server-Side Controller determines the price of its computation by combining its local price with downstream price data. A service's price is the price of local computation plus the maximum of prices published by relevant downstream services. This policy emphasizes the most overloaded downstream service, allowing rapid responses to hotspots. Moreover, this policy is easy to implement. RAJOMON simply forwards a request's token holdings through the graph as it proceeds from client to back-end.

In contrast, an intuitive policy would resemble supply chains. A service's price is its local price plus the sum of prices for all relevant downstream services. If prices are additive, then token spending must also be additive. Whenever a request fans-out and triggers parallel sub-requests for downstream service, its token holdings must be split, allocated, and attached to those sub-requests. We must split tokens so adaptive queue management at back-end services are effective, but doing so adds complexity and latency.

3.5 Example

Figure 3 provides an example of RAJOMON's operation. A client requests computation from three services. Suppose server 1's `login` interface calls both server 2's `authentication` and server 3's `homepage` interfaces.

1 The **Client-Side Controller** generates tokens periodically as determined by its policy (*e.g.*, 10 tokens every 10ms). 2 The client issues a request to `login` if it holds enough tokens for the interface. Here, the client holds 10 tokens and `login`'s price is 5 tokens. 3 The client spends 5 tokens, attaching them to the outgoing `login` request.

4 The **Server-Side Controller** checks the request's tokens against the price, admits the request, and attaches 5 tokens to each sub-request to `authentication` and `homepage`. 5 The server periodically updates prices based on queuing delays, increasing prices for `homepage` and `authentication` to 7 and 8 tokens, respectively. 6 Servers 2 and 3 drop in-

coming requests, which hold 5 tokens, and responds to server 1 with updated prices. ❼ Server 1 adjusts its price table based on local queuing delay and the maximum price of downstream services (*i.e.*, 8 tokens). When server 1 receives and drops incoming requests with insufficient token holdings, it will respond to upstream clients with updated prices.

This example illustrates how RAJOMON coordinates load shedding and rate limiting for microservice graphs. For fanned-out microservices, suppose Servers 2 and 3 must drop half their requests. Since sub-requests inherit tokens from their originating request, the servers can identify and drop the half with fewer tokens attached. The servers would focus on completing the remaining sub-requests and better translate computation into goodput.

For a chain of microservices, when Server 1 receives increased prices from Servers 2 and 3, it immediately adjusts its price and drop more incoming requests. Doing so mitigates overload earlier in the graph and reduces load shedding at Servers 2 and 3. Moreover, Server 1 propagates its increased price to clients and prompt more aggressive rate limiting.

4 Implementation

The RAJOMON prototype is implemented as an open-source gRPC package [48] in 948 lines of Go code [5, 9]. The implementation includes three parts. First, for overload detection, RAJOMON implements a goroutine that periodically measures queuing delays for other running goroutines. Go uses goroutines instead of threads and our implementation is analogous to prior techniques that monitor thread queuing delays [3, 52]. Second, for price tables, RAJOMON uses synchronous maps with built-in support for concurrent use [16]. Third, RAJOMON implements gRPC interceptors for service receivers, service senders, and clients. Interceptors implement rate-limiting and adaptive queue management (§3).

Baselines and implementation. We compare RAJOMON to state-of-art microservice overload control frameworks—(i) Dagor, (ii) Breakwater, (iii) Breakwaterd, (iv) TopFull—which were introduced in Section 2.3. Breakwaterd is a distributed implementation of Breakwater that rate limits across all sender-receiver pairs in the service graph. This was suggested as a future work, but was neither implemented nor evaluated in the original study [3].

For fair comparison, we implement all baselines in Golang as gRPC libraries. The re-implementation follows the original design, but as a gRPC interceptor, we removed Breakwater's dependencies on certain hardware or operating system kernels [3, 12, 32]. We open source our implementation for Dagor, Breakwater(d), and TopFull [23, 46, 47].

Benchmarks. We evaluate RAJOMON using academic benchmarks and industry traces from Alibaba (Table 2). We re-implement academic benchmarks, Social Network and Hotel Reservation, in GoLang with gRPC and Redis based on code from [49]. SLOs for each interface are set to 5× its 95th-percentile latency under normal conditions.

We use microservice application traces from Alibaba [24] to generate synthetic benchmarks (traces are not deployable applications). We consume service call graphs and each service's computation times to produce corresponding service instances in Golang. Each service runs a busy-loop to emulate each request's computation. We focus on three highly demanded microservice endpoints—S_102000854, S_149998854, S_161142529—which we denote *S1*, *S2*, and *S3*. Endpoints are representative of call graphs of different sizes (10-20, 20-30, 30-40 nodes).

App	Interfaces	Depth	Width	#Nodes	SLO (ms)
Social Network	Compose Post	4	3	9	90
	User Timeline	4	2	7	20
	Home Timeline	4	1	5	20
Hotel Reservation	Search Hotel	4	3	9	60
	Reserve Hotel	3	2	5	25
Alibaba	S1	5	10	20	250
	S2	9	10	33	600
	S3	6	6	18	375

Table 2: Microservice benchmarks and SLOs.

Parameter Tuning. Overload control frameworks commonly involve four key parameters. First, queuing threshold specifies when overload control is triggered. Second, update frequency specifies how often control variables such as token holdings and prices adjust. Third, step width specifies the magnitude of the updates to control variables. Finally, client-side parameter include the token generation rate in RAJOMON and the expiration time in Breakwater. We allow Breakwaterd to have two sets of parameters—one for the front-end service and another for the other services—resulting in twice the number of parameters compared to Breakwater.

We use Bayesian optimization to tune overload control parameters for each application on its slowest request interface (*e.g.*, *Compose Post* for *Social Network* and *Search Hotel* for *Hotel Reservation*). The objective function for tuning is a weighted sum of goodput and tail latency, specifically defined as $goodput - 10 \times \max(0, latency - SLO)$, following similar methods from prior work [3, 4, 34]. In each iteration of the optimization procedure, we generate a traffic surge from 80% to 200% of the application's maximum sustainable load and then measure goodput and tail latency.

For TopFull, we adhere to their transfer learning process. We implemented a graph-based simulator to first train the RL model in the simulated environment before fine-tuning it on the real application. We use the same reinforcement learning model and parameters as described in the original paper.

5 Evaluation

We evaluate RAJOMON performance characteristics to answer the following questions.

1. How does RAJOMON affect the goodput and latency of microservice graphs under various loads? (§5.2)
2. How does RAJOMON impact the time required for goodput to recover from an overload condition? (§5.3-5.4)
3. What is the impact of RAJOMON on the goodput and latency of multiple, concurrent request interfaces? (§5.5-5.7)
4. How large is RAJOMON's parameter variation compared to other control schemes?(§5.8)
5. What is the latency overhead of RAJOMON compared to other control schemes? (§5.9)

5.1 Experimental Setup

Testbed. We perform most experiments on seven CloudLab m510 servers, equipped with 8-core Intel Xeon D-1548 CPUs, 64GB DDR4 RAM, 256GB NVMe SSDs, and 10GB NICs, unless otherwise stated. For *Social Network* and *Hotel Reservation* experiments involving multiple interfaces (§5.5, 5.6), we scaled up to seven c220g2 servers, each equipped with 20-core Intel Xeon E5-2660 v3 CPUs, 160GB RAM, and 10GB NICs. All servers run Ubuntu 18.04 LTS with kernel 4.15.0. Services are deployed with Kubernetes [21]. One dedicated node runs the control plane and clients while all other nodes run microservices. All nodes communicate through a single router. Each experiment for the *Social Network* or *Hotel Reservation* applications runs on eight nodes, while the Alibaba application runs on twenty nodes.

Load Generation and Measurement. We extend *ghz* for load generation [8]. On the client node, 1000 workers establish 1000 concurrent gRPC connections and issue requests asynchronously to the application. Requests follow a Poisson process with some rate specified by the experiment. We randomize input data for *Hotel Reservation* and *Social Network* requests because experiments focus on success rate rather than the content of responses.

Each experiment consists of a warmup (5s) and overload (10s) phase unless otherwise noted. For experiments with a single request type, we warm up with load at 80% of the application's maximimum sustainable throughput subject to its SLOs and then overload to some rate specified by the experiment. For experiments with multiple concurrent request types, we warm up with fast, computationally light requests and then overload with slow, computationally heavy requests.

We measure end-to-end request latency during overload and report 95th-percentiles for latency and goodput (*i.e.*, number of requests processed within the SLO). We do not report warm-up performance unless otherwise noted.

5.2 Performance for Single Interfaces

Figure 4 presents data for *Search Hotel* from the *Hotel Reservation* app and *Compose Post* from the *Social Network* app. Each experiment includes a warm-up phase—4k requests per second (RPS) for *Search Hotel* and 2k for *Compose Post*—followed by an overload phase with varied load spikes on x-axis. Data is the average of five experiments with error bars indicating variance in latency and goodput.

Results. For *Search Hotel*, RAJOMON ensures tail latency remains below 200ms and sustains goodput of approximately 3k RPS during overload. Goodput drops only marginally as traffic load increases. In contrast, baseline techniques cannot maintain performance as load increases. Once load exceeds 12k RPS, their tail latencies are five times that of RAJOMON's and their goodputs are less than half of RAJOMON's. TopFull maintains lower, steady goodput at 1k RPS but suffers from much higher tail latencies of 750ms.

For *Compose Post*, RAJOMON's tail latency increases modestly above the 80ms SLO, approaching 100ms, while goodput remains stable above 2k RPS. In contrast, baseline techniques exhibit a piecewise linear trend. Tail latencies spike to 800ms when load is 5k RPS and then stabilize. Goodput drops to 500 RPS and then stabilizes. Dagor is a bit more robust and sustains performance until load is 7k RPS.

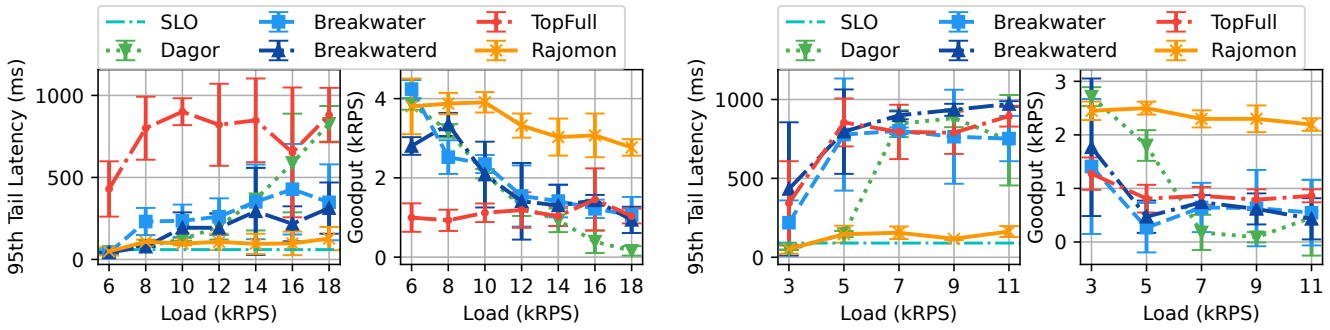
Discussion. RAJOMON's decentralized control delivers uniquely robust performance. It keeps tail latency low by quickly shedding load. Price updates regulate the rate of incoming requests from upstream clients and services. Moreover, RAJOMON's collaborative control keeps goodput high, even under heavy load conditions, as back-end services drop sub-requests that originate from the same upstream request.

In contrast, Breakwater is designed specifically for front-end services and lacks downstream visibility, which delays and diminishes any adaptation to overload in back-end services. Less effective adaptation means lower goodput and higher latency, especially under heavy load because requests can quickly accumulate in queues before overload is detected.

Dagor sheds load to counter request incast, but wastes server-side computation as client-side rate limiting would be more efficient. Shedding becomes more aggressive as load increases, which leads to lower goodput and higher latency.

Breakwaterd extends Breakwater to each sender-receiver pair in the graph but suffers for the same reasons as Dagor. Requests are rate-limited and dropped only in the node preceding an overloaded hotpsot because overload signals do not propagate through the graph. As load increases, more partially processed requests are ultimately dropped.

TopFull is limited by its coarse timescales. Within the 10-second overload phase, TopFull experiences an initial period of service unavailability followed by a slow recovery as it begins adaptation. Delayed detection and response means lower goodput and higher latency, which is consistent with its own experimental evaluation [34]. Although it is slow to



(a) Tail latency and Goodput during overload of *Search Hotel*

(b) Tail latency and Goodput during overload of *Compose Post*

Figure 4: Overall performance of RAJOMON and other baselines for *Search Hotel* and *Compose Post* requests.

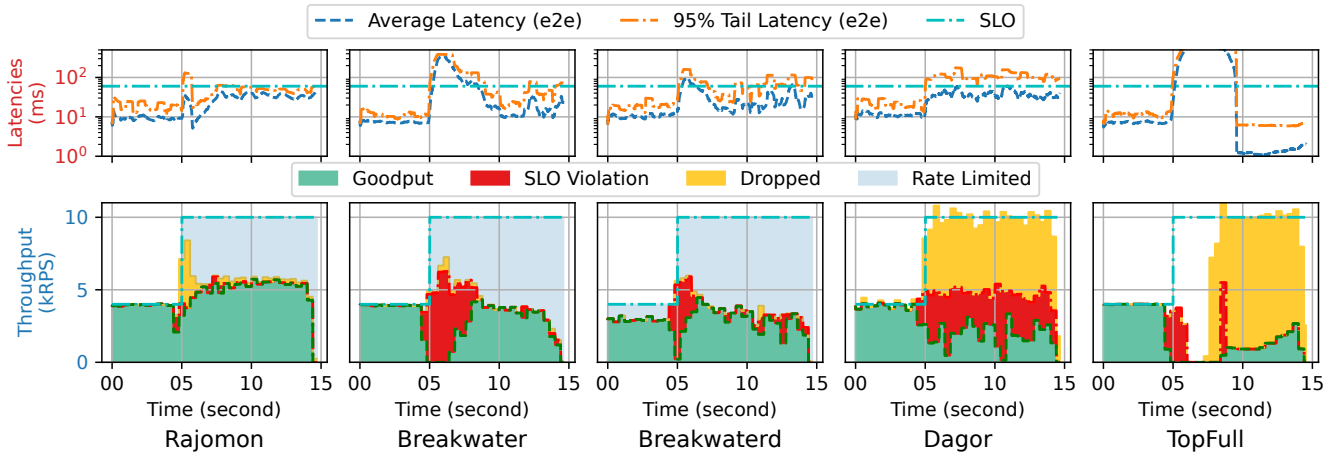


Figure 5: The throughput and tail latency under traffic surge of *Search Hotel* (single interface).

respond, TopFull's capable reinforcement learning model can eventually stabilize performance at medium and high load.

Across all baselines, performance losses are more pronounced for *Compose Post* because its back-end services require more intensive Redis writes. When not managed effectively, overload produces much greater increases in latency.

5.3 Control Dynamics for Single Interface

Figure 5 illustrates system dynamics. Client demand is 4k RPS during warm-up followed by 10k RPS during overload. The first row shows latency across time. The second row shows how requests are either rate-limited, dropped, or processed with a response time that may or may not satisfy SLOs.

Results. When overload begins at 5 seconds, RAJOMON drops many requests and then, over the next 2 seconds, steadily recovers goodput until it reaches 5k RPS. Most excess load is controlled through rate limiting thereafter. Latency initially spikes to 100ms but quickly returns to SLO target.

Breakwater(d)'s responses to overload do not occur until 2 to 4 seconds after its onset. During these periods, 50% to 100% of requests violate the SLO target latency. Eventually, latency returns to target but goodput stabilizes at lower levels, below the 4k RPS during warm-up.

Dagor maintains throughput of 5k RPS and aggressively drops the additional 5k RPS of overload. However, 95th percentile latency exceeds 100ms and most of its throughput violates the SLO. TopFull encounters 4 seconds of unavailability due to overload, followed by many dropped requests and slow goodput recovery.

Discussion. RAJOMON promptly detects server-side congestion and drops excess requests at 5 seconds. After this initial reaction, RAJOMON propagates updated prices upstream to clients, which rate limits subsequent requests. According to RAJOMON's lazy update policy, price updates are included in 20% of responses, which explains the 2-second transition from server-side load shedding to client-side rate limiting, which stabilizes latency and maintains goodput.

In contrast, Breakwater(d) experiences delays and inefficiencies due to a lack of coordination. The Breakwater front-end requires 2 seconds to detect queuing delays from back-end overload. Even after detection, back-end delays are higher

than those at the front-end. Since Breakwater issues credits based on front-end delays, this mismatch adds another 2 seconds to load adjustment even though Breakwater is tuned to update credits every 1ms.

Breakwaterd responds more quickly as it reduces load at intermediate tiers. When the back-end becomes overloaded, mid-tiers rate-limit and queue incoming requests, which eventually time out. However, timed-out requests do not signal upstream services or clients, leading to wasted computation on requests that are ultimately discarded. This inefficiency explains Breakwaterd's sub-optimal goodput.

Dagor's lack of rate limiting leads to many dropped requests, which slows down the system. Dagor drops requests only in the nodes preceding hotspots. For four-tier *Search Hotel*, most services continue processing 10k RPS even though half of these requests are dropped at the second-to-last tier. This wastes upstream computation and increases latency.

TopFull's centralized, front-end-only design inherits the limitations of both Breakwater and Dagor. It detects overload only after end-to-end goodput drops, preventing timely reactions to overload and causing extended service downtime. Moreover, load is shed only at the front-end, which wastes computation on requests that cannot satisfy SLOs and contribute to goodput. Reinforcement learning should eventually adapt to new load conditions but over a long timescale.

5.4 Recovery Time

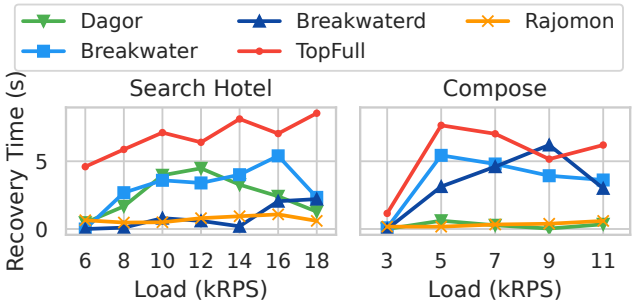


Figure 6: Recovery time of RAJOMON and baselines for *Search Hotel* and *Compose Post* request.

Figure 6 plots average recovery time under varying load conditions. We define *recovery time* as the period between the onset of overload and goodput stabilization at some level.

Results. RAJOMON consistently maintains sub-second recovery regardless of load conditions. Although baselines demonstrate sub-second recovery at low loads, they often struggle at higher loads. For Breakwater, recovery time rises to around 4 seconds as load increases. For Breakwaterd and Dagor, recovery times vary depending on the workload. Breakwaterd responds slowly for *Social Network* and Dagor for *Hotel Reservation*; recovery times range from 3 to 5 seconds depending on the severity of overload. Topfull is slow and often requires more than 5 seconds to recover.

Discussion. RAJOMON is the only framework that consistently recovers in less than a second. Its decentralized overload control is fast and responsive, regardless of load.

In contrast, baseline techniques present a trade-off between recovery time and stable goodput. Breakwaterd and Dagor's decentralized designs help them react to overload and reach equilibrium faster, but at the expense of goodput. They respond quickly, in part, because their stable goodput after the onset of overload is much lower than RAJOMON's. Similarly, as load increases, Breakwater reports lower goodput and shorter recovery times. TopFull suffers from longer recovery times due to its centralized architecture.

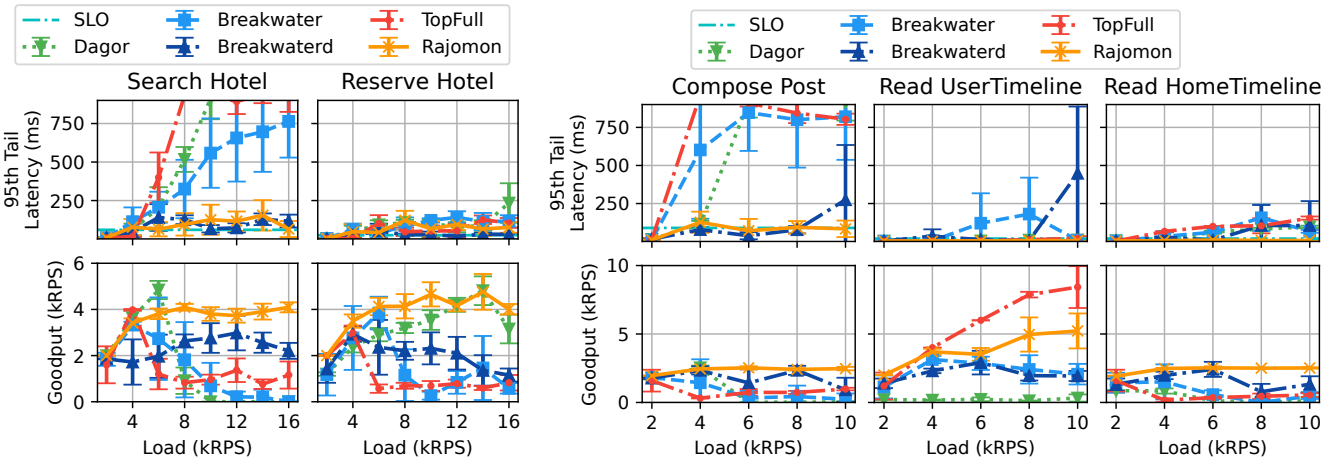
5.5 Performance for Multiple Interfaces

Figure 7 details RAJOMON's performance for concurrent requests to multiple interfaces. For the *Hotel Reservation* application, we generate *Reserve Hotel* requests at a certain rate during warm-up and add an equal amount of *Search Hotel* traffic during overload. For the *Social Network* application, we generate *Read User Timeline* requests during warm-up and add *Compose Post* and *Read Home Timeline* traffic during overload. Note x-axis reports load per interface and, for example, $x = 10$k RPS in the *Social Network* application means 30k RPS total load from three request types.

Results. In Figure 7a, RAJOMON maintains a latency of approximately 100ms and a goodput of 4k RPS for both *Search Hotel* and *Reserve Hotel* requests, even as the load increases. In contrast, Breakwater, Dagor, and TopFull maintain low tail latencies for *Reserve Hotel* but exhibit increased latencies for *Search Hotel* as the load increases. The goodput of most baselines for both interfaces drops from 4k to 1k as load approaches 14k RPS. Two exceptions are Breakwaterd for *Search Hotel* and Dagor for *Reserve Hotel*, both of which maintain goodput above 2k RPS while sustaining low tail latencies.

In Figure 7b, RAJOMON maintains latency within SLOs across all request types. It sustains *Compose Post* and *Read Home Timeline* goodput at 2.5k RPS, while *Read User Timeline* goodput increases to 5k RPS as load intensifies. Most baselines exhibit reduced goodput and increased latencies for *Compose Post* once the load surpasses 4k RPS. For *Read User Timeline* and *Read Home Timeline*, baseline goodput diminishes to less than half of RAJOMON's performance as the load nears 10k RPS. Notably, Topfull achieves higher *Read User Timeline* goodput beyond 6k RPS but at the expense of *Compose Post* and *Read Home Timeline* interfaces. Breakwaterd stands out by maintaining latency within SLOs for most interfaces; however, it experiences a decline in goodput.

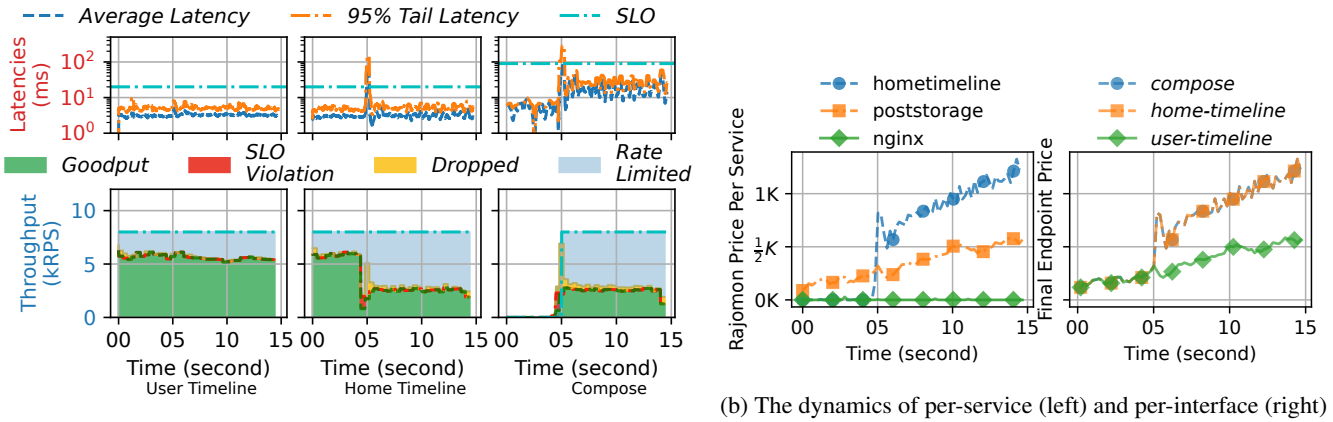
Discussion. RAJOMON demonstrates low latency and high goodput even under heavy concurrent loads from multiple request types. It achieves this by combining AQM with rate-



(a) The performance of OC on concurrent *Search Hotel* and *Reserve Hotel* requests.

(b) The performance of OC on concurrent *Compose Post*, *Read User Timeline*, and *Read Home Timeline* requests.

Figure 7: The performance of RAJOMON and baselines on concurrent requests (up to 30k RPS in total).



(a) The dynamics of the latency and throughput.

(b) The dynamics of per-service (left) and per-interface (right) prices.

Figure 8: The dynamics of the throughput, latency, and prices for concurrent *Social Network* requests.

limiting and assessing the maximum sustainable load for each interface separately. RAJOMON's advantage is particularly evident for *Read User Timeline*, which has the potential for higher goodput and lower latency because these requests do not share hotspots with *Compose Post* and *Read Home Timeline*. Only RAJOMON manages to realize this potential by avoiding head-of-line blocking. Per-interface pricing effectively virtualizes a multi-queue system and sets distinct admission rates for each interface.

Dagor distinguishes between interfaces, but its adaptive queue management lacks coordination across the service graph. The same inefficiencies observed in Figure 5 persist: The massive load shedding in mid-tier services increases the latencies of *Search Hotel* and *Compose Post* requests. This finding highlights that interface granularity must be combined with end-to-end coordination to be fully effective.

Breakwater are oblivious to interfaces and suffer from head-of-line blocking because, for instance, they cannot rate limit the slow requests (*Search Hotel* and *Compose Post*) independently. As a result, other interfaces also suffer from low goodput. If Breakwater could differentiate between interfaces, they could issue fewer credits for slow interfaces, ensuring performance for all interfaces. Breakwaterd achieves low tail latencies but at the expense of reduced goodput. This occurs because Breakwaterd reduces load in intermediate tiers without coordinating with frontend and end users, as previously discussed in Sections 5.2 and 5.3. Therefore, effective rate-limiting mechanisms should incorporate interface granularity and end-to-end coordination to optimize request rates.

TopFull cluster interfaces based on shared hotspots and implements admission control for each cluster. In our case, *Search Hotel* and *Reserve Hotel* form one cluster, *Compose*

Post and *Read Home Timeline* form another, and *Read User Timeline* forms a third. This approach can sometimes work. The control agent sets a 6k RPS rate for *Read User Timeline* while keeping latency within SLOs. But, for other clusters, the lack of decentralized control and client-side rate-limiting constrains performance. Although TopFull can set different rates for *Search Hotel* and *Reserve Hotel*, its slow reaction times prevent it from handling even 1k RPS effectively.

5.6 Control Dynamics for Multiple Interfaces

Figure 8 analyzes system dynamics for the *Social Network* application's multiple request types. Figure 8a follows the same interpretation as Figure 5, except for the presence of multiple concurrent requests. Figure 8b details service prices at interface granularity. Recall that the price of a service is the maximum of all the downstream prices paid by its sub-requests.

Results. Figure 8a indicates a negligible number of dropped requests and SLO violations. Once *Compose Post* requests arrive, RAJOMON immediately rate limits *Read Home Timeline*. Figure 8b illustrates how RAJOMON dynamically adjusts prices for different services to reflect their load. During warm-up, *post storage* service is the only hotspot and its price dictates prices for all request types. During overload, *post storage*'s prices keep increasing slowly so *Read User Timeline* interface remains inexpensive. In contrast, *home timeline*'s price increases substantially within half a second, indicating an emerging hotspot. The price increase propagates upstream such that *Compose Post* and *Read Home Timeline* both become much more expensive.

Discussion. RAJOMON's dynamic prices play a crucial role. First, prices reflect real-time load and delay across services. When *home timeline*'s delays are immediately reflected in *Compose Post* and *Read Home Timeline* prices, ensuring that overload control is precise and timely.

Furthermore, prices at interface granularity isolate *Read User Timeline* requests, which do not involve the overloaded *home timeline* service. These requests are neither rate limited nor dropped, ensuring consistent performance even when other parts of the system are under stress.

5.7 Performance for Alibaba Traces

We further evaluate three real-world traces from Alibaba (Section 4). We construct a service graph that is the union of *S1*, *S2*, and *S3*, which makes it much larger than *Hotel Reservation* and *Social Network*. We start *S1* and *S3* during warm-up and then add *S2* during overload.

Results. Figure 9 indicates RAJOMON maintains latencies below 200ms and goodput at 4k RPS for all interfaces and varied loads. Collectively, the baselines are less effective. As load approaches 8k-10k RPS, latency increases exponentially beyond 500ms and goodput falls by half.

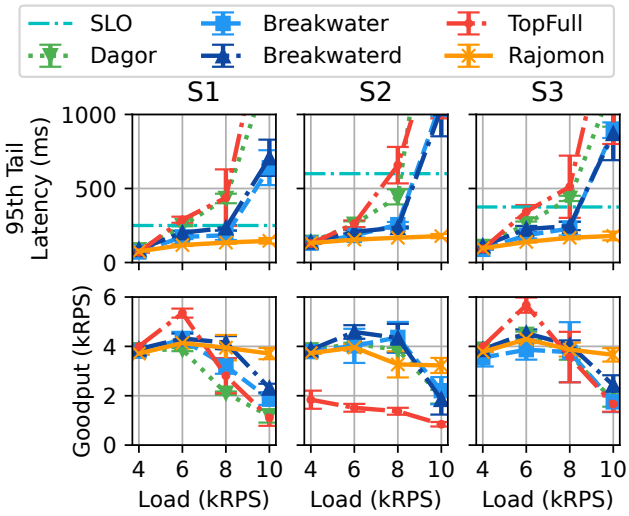
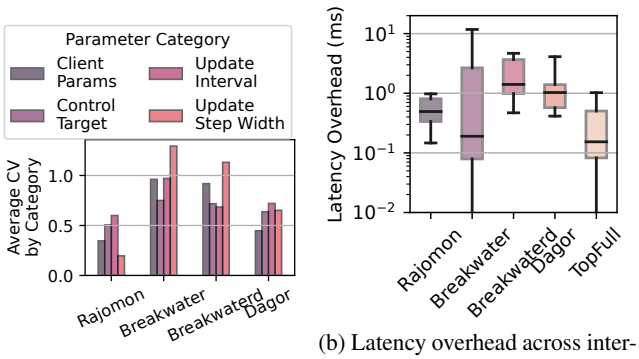


Figure 9: The performance impact of RAJOMON and baselines on large service graphs (up to 30k RPS in total).

Discussion. RAJOMON manages overload efficiently at scale, maintaining low latencies and high goodput even under extreme load. In contrast, baselines struggle under high load. Breakwater's rate limiting scales better and achieves consistently lower latencies than Dagor's adaptive queue management (*i.e.*, dropping requests) as detailed in §5.2. TopFull's RL agents set effective admission rates for *S1* and *S3*. But the reason for *S2*'s rate is unknown because TopFull is a black-box that does not detail the interactions between its RL agents. Yet its centralized approach for dropping requests suffers from higher latencies and lower goodput for all interfaces.

5.8 Parameter Variability



(a) CV of tuned parameters.

(b) Latency overhead across interfaces.

Figure 10: Comparison of parameter variability and latency overhead.

Overload control frameworks must be tuned for each application to be effective [3, 34]. A robust framework should

be less sensitive to tuning parameters. Such a framework would reduce tuning time and effort, while providing more consistent performance across varied applications and system conditions, given specific parameter values.

We quantify robustness with the Coefficient of Variation (CV) for tuned parameters.[2] CV measures the relative variability of parameters, permitting comparisons across control frameworks. We calculate CV of each framework's optimal parameters in Table 3 across the three applications in Table 2.[3]

Figure 10a indicates RAJOMON performs well with similar parameters for varied applications. It exhibits the lowest variability in tuned parameter values, indicating we need not extensively tune and optimize for each application.

Breakwater(d) exhibits higher variability, indicating the difficulty of re-tuning its parameters for each application and highlighting the risks from inconsistent performance. Breakwater's credit-based rate limiting requires precise control over the total number of credits issued and held by each client, making performance more sensitive to parameters. Dagor exhibits lower variability than Breakwater(d) because priority-based adaptive queue management does not cause abrupt changes in traffic and tends to be more robust [3].

5.9 Overhead Comparison

We assess overheads by measuring differences in median latency with and without control for a system with modest 2k RPS load. Figure 10b presents the distribution[4] of latency overheads across eight service interfaces in Table 2.

RAJOMON's overhead is low for two reasons. First, it maintains and propagates only necessary state such as tokens per request and price per interface. In contrast, Dagor updates large matrices at all microservices that track request priorities and determine admission levels based on a priority cut-off: (B,U). These matrices, which grow linearly with the number of interfaces and user groups, suffer from lock contention. Note that Dagor and Breakwaterd's accumulate across tiers in deep microservice graphs, harming end-to-end latency.

Second, RAJOMON performs local price calculations asynchronously and off the critical path. TopFull also incurs low overheads because it uses only one token bucket at the frontend and its reinforcement learning decisions are asynchronous. In contrast, Breakwater calculates a client's new credit holdings c_x^{new} before every response. This overhead lengthens queues and harms latency, especially under higher loads.

6 Related Work and Discussion

Network Congestion Control Extensive work has addressed congestion control at the network layer [1, 18, 20, 22, 28, 44, 51]. RAJOMON, however, accounts for microservice-specific factors like call graphs and interdependencies. Future work could integrate RAJOMON with these mechanisms to improve end-to-end performance.

Load Balancing and Autoscaling. RAJOMON complements existing resource management strategies. While autoscaling increases service capacity, it is inherently slow and typically requires 40–200 seconds [30, 33–35, 38–40, 50]. In contrast, RAJOMON provides rapid overload control on a subsecond timescale, mitigating performance issues before autoscaling adjustments take effect. Once autoscaling provisions additional resources, the prices fall accordingly to accommodate higher throughput. Moreover, the tokens and prices primitives provide insight into request characteristics and downstream loads. This information can enhance load balancing and job scheduling decisions. Finally, even with optimal scheduling and load balancing, systems can experience overload due to unexpected traffic bursts that exceed capacity [14, 29, 31, 43, 45]. In such scenarios, RAJOMON serves as a safeguard, maintaining performance when other strategies are insufficient.

Malicious Clients. RAJOMON assumes trusted clients, such as proprietary mobile apps or Internet-of-Things devices that account for most traffic [27]. However, untrusted clients may attempt to exploit the framework for more resources or disrupt services. To prevent this, RAJOMON would require a server-side validation module to ensure token spending aligns with token generation policy and history. Trust models and validation techniques are an avenue for future research.

Dynamic Service Graphs. We focused on applications with deterministic execution paths. In practice, microservices may have dynamic call paths. If call paths depend on request metadata, such as branching based on payload, these requests can be treated as different interfaces upon receipt. If call paths are random and unknown upon receipt, for example, due to cache misses, one extension for RAJOMON could use probability theory to update its price table based on the expected value of downstream costs. For instance, if the price is 10 for a cache hit and 50 for a miss, with a 10% miss rate, the local price should be 15, based on expectation. Future work could explore other pricing policies for dynamic paths and flexible microservice settings.

7 Conclusion

RAJOMON presents a novel, market-based overload control system for microservices, using tokens and price tables for distributed rate limiting and load shedding while coordinating control throughout service graphs. RAJOMON significantly outperforms state-of-the-art baselines, improving goodput by more than 45% and reducing latency by more than 78% under high load. Its fast and precise control, with sub-second recovery times, ensures system reliability and efficiency in complex microservice systems.

[2] CV is ratio of standard deviation and mean [10].
[3] TopFull's RL control is excluded.
[4] Maximum, 75th, 50th, 25th percentile, and minimum.

References

[1] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, aug 2013.

[2] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[3] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314. USENIX Association, November 2020.

[4] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 725–738, Boston, MA, April 2023. USENIX Association.

[5] Cloud Native Computing Foundation. gRPC, A high performance, open source universal RPC framework. https://grpc.io/, 2024.

[6] Adrian Cockcroft. Evolution of Microservices - Craft Conference. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, June 2024.

[7] Adrian Cockcroft. Microservices Workshop - Craft Conference. https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference, June 2024.

[8] Bojan D. ghz · gRPC benchmarking and load testing tool. https://ghz.sh/, 2023.

[9] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[10] Brian S Everitt and Anders Skrondal. *The Cambridge dictionary of statistics*. Cambridge University Press, 2010.

[11] Cloud Native Computing Foundation. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[12] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[14] Zhaoyu Gao, Anubhavnidhi Abhashkumar, Zhen Sun, Weirong Jiang, and Yi Wang. Crescent: Emulating heterogeneous production network at scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1045–1062, Santa Clara, CA, April 2024. USENIX Association.

[15] Stephanie Giang-Paunon. Taylor Swift's 'Midnights' album crashes Spotify, leaving fans shocked; nearly 8,000 outages reported. https://www.foxbusiness.com/entertainment/taylor-swifts-midnights-album-crashes-spotify-fans-shocked-8000-outages-reported, 2022.

[16] Go. sync package - sync - Go Packages. https://pkg.go.dev/sync#Map, 2024.

[17] Google. What Users Want Most from Mobile Sites Today. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/what-users-want-most-from-mobile-sites-today/, 2012.

[18] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.

[19] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 325–342, Boston, MA, July 2023. USENIX Association.

[20] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *SIGCOMM Comput. Commun. Rev.*, 42(4):127–138, aug 2012.

[21] Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io/, 2024.

[22] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. Flow scheduling with imprecise knowledge. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 95–111, Santa Clara, CA, April 2024. USENIX Association.

[23] Loh, Paul and Xing, Jiali. Breakwater gRPC Implementation. https://github.com/pennsail/breakwater-grpc. Accessed: 2024-09-16.

[24] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 412–426, New York, NY, USA, November 2021. Association for Computing Machinery.

[25] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An In-Depth Study of Microservice Call Graph and Runtime Performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, December 2022. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[26] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 355–369, New York, NY, USA, November 2022. Association for Computing Machinery.

[27] Daniele Molteni. Landscape of API Traffic. https://blog.cloudflare.com/landscape-of-api-traffic, January 2022.

[28] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities (Complete Version). *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, August 2018. arXiv: 1803.09615.

[29] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. Zero downtime release: Disruption-free load balancing of a multi-billion user website. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 529–541, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for Infrastructure-as-a-Service. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, June 2013. USENIX Association.

[31] Yipei Niu, Fangming Liu, and Zongpeng Li. Load balancing across microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 198–206, 2018.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[33] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '21, pages 154–167, New York, NY, USA, December 2021. Association for Computing Machinery.

[34] Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. Topfull: An adaptive top-down overload control for slo-oriented microservices. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 876–890, New York, NY, USA, 2024. Association for Computing Machinery.

[35] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[36] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver,

Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[37] Sebastian Sinclair. Coinbase site crashes from traffic as super bowl ads spark public interest. *Blockworks*, 2022.

[38] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, Zongquan Li, Ziyang Fang, Biao Lyu, Pengyu Zhang, Rong Wen, Li Yi, Zhigang Zong, and Shunmin Zhu. Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 860–875, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Akshitha Sriraman and Thomas F Wenisch. µTune: Auto-Tuned Threading for {OLDI} Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, 2018.

[40] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. Autothrottle: A practical Bi-Level approach to resource management for SLO-Targeted microservices. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 149–165, Santa Clara, CA, April 2024. USENIX Association.

[41] Matt Welsh and David Culler. Overload management as a fundamental service design primitive. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 63–69, New York, NY, USA, July 2002. Association for Computing Machinery.

[42] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, October 2001.

[43] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[44] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: meeting deadlines in datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):50–61, aug 2011.

[45] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. Load is not what you should balance: Introducing prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1285–1299, Santa Clara, CA, April 2024. USENIX Association.

[46] Xing, Jiali. Dagor gRPC Implementation. https://github.com/pennsail/dagor-grpc. Accessed: 2024-09-16.

[47] Xing, Jiali. TopFull gRPC Implementation. https://github.com/pennsail/topdown-grpc. Accessed: 2024-09-16.

[48] Xing, Jiali and Giannoukos, Akis. Rajomon Source Code. https://github.com/pennsail/rajomon. Accessed: 2024-09-16.

[49] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. {MuCache}: A general framework for caching in microservice graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 221–238, 2024.

[50] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, pages 167–181, 2021.

[51] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: admission control for performance-critical rpcs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 1–18, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 149–161, New York, NY, USA, October 2018. Association for Computing Machinery.

A Alibaba Production Trace Samples

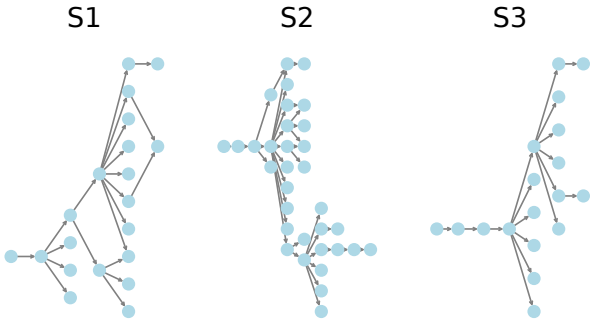


Figure 11: Sampled Alibaba services.

Figure 11 illustrates the service call graphs, *S1*, *S2*, and *S3*, sampled from Alibaba production traces. While shown separately, these graphs share some common microservices. In our experiments, the three graphs were deployed together across 20 servers.

B Overload Control Parameters

Mechanism	**Control Target**	**Update Interval**
Dagor	queuing threshold	threshold update rate
Breakwater	queuing threshold	C_{total} update rate
Breakwaterd	queuing threshold	C_{total} update rate
RAJOMON	queuing threshold	price update rate
Mechanism	**Update Step Width**	**Clients**
Dagor	α, β	u_{max}
Breakwater	α, β	initial credit, client expiration
Breakwaterd	α, β	initial credit, client expiration
RAJOMON	price update step	token update rate

Table 3: Tuned parameters for overload control mechanisms, categorized by their function.

Table 3 shows the parameters tuned for each overload control mechanism.

Operational Systems Track: Learnings from Deploying Network QoS Alignment to Application Priorities for Storage Services

Matthew Buckley[†‡], *Parsa Pazhooheshy*[†‡], *Z. Morley Mao*[†], *Nandita Dukkipati*[†], *Hamid Hajabdolali Bazzaz*[†], *Priyaranjan Jha*[†], *Yingjie Bi*[†], *Steve Middlekauff*[†], *Yashar Ganjali*[‡]

[†]***Google LLC.***, [‡]***University of Toronto***[*]

Abstract

To ensure that application network traffic is prioritized correctly within data center networks, it is critical to align the configuration of network QoS in packets to the intended priority of the application. These QoS configurations, typically encoded in the DSCP bits in the IP header, are interpreted by network switches and routers to determine the resources such as buffer space and scheduling priorities, for network traffic. Conceptually, it appears fairly straightforward to map the application priorities within data center networks to network QoS configurations, as long as the mapping is well defined. In this work, we describe our experience of aligning network QoS settings for intra-cluster storage traffic to application priorities on a per-RPC basis for a large data center network, with well-defined static mappings from priorities to QoS traffic classes. We describe some unexpected insights learned from the deployment experiences, e.g., downgrading traffic to use a lower QoS does not always imply worse network latency due to over-used QoS bands in the network. We also share some challenges encountered along the way to reach the goal of a fleet-wide deployment, including the concerns of potential performance regressions due to QoS downgrades. These lessons provide guidance on the use of a QoS-based scheduling strategy to meet service guarantees and can be deployed to networks of any scale.

1 Introduction

Performance of network services heavily depend on the delay along the network path, as well as other factors such as server load [1–4]. Network delays can be much higher under congestion when the load imposed by arriving traffic exceeds network capacity [5–9]. This affects both wide-area network (WAN) links and paths within data centers, as traffic traversing both types of networks can experience queuing delays from congestion. While various host-based and network-based techniques such as admission control [10–14] can help reduce the traffic demands to minimize congestion, transient congestion can still occur due to bursty traffic patterns [3, 15–17]. This causes switches along the network path to drop packets when faced with full buffers.

The latency of higher-level application entities, such as Remote Procedure Calls and Remote Memory Accesses, plays a crucial role in the applications' performance. Different applications exhibit varying sensitivities to these delays; some, particularly those that prioritize tail latency, cannot tolerate any packet loss, while others focus on average latency or overall throughput. To manage network performance, switches utilize Quality of Service (QoS) bits in packet headers to deliver tailored service levels and latency expectations based on application demands. In general traffic is distributed across various QoS levels based on the specific requirements of workloads, allowing QoS queues to establish priorities according to their latency expectations. When congestion occurs, this prioritization enables the dropping of packets for lower-priority traffic, thus minimizing the impact on more latency-sensitive workloads. Thus, designing the right priority enforcement within data-center networks requires marking traffic according to the intended priority at the application layer.

In [14], Aequitas was introduced as Google's solution for aligning application priorities with network QoS. It was argued that this enforcement was most effective at the Remote Procedure Call (RPC) level and the work primarily focused on the design aspects with some production examples from early rollout. In this work, we expand on ideas introduced there and focus on our deployment experience from implementing the primary phase of Aequitas in Google's storage systems. We show production examples that justify the originally proposed design and offer new insights regarding the complex relationship between QoS queues and network latency.

In summary, our work focuses on these practical aspects of deploying Aequitas in Google networks for the storage services, which all fall under Phase 1 in [14]: (1) defining the static mapping from application priority to network QoS, (2) enforcing these QoS settings on traffic at the RPC level based on the priorities, and (3) supporting any exceptions to bypass this scheme due to special requirements.

[*]Work completed while the students were at Google

Thus, we clarify the goals of deploying Aequitas for storage to prevent priority inversion, which occurs when traffic of higher priority receives less network resources compared to traffic of lower priority. These are a refined set of goals as compared to [14] based on our deployment experience.

- *RPC level enforcement*: The granularity of RPC level enforcement of QoS configuration is necessary so that it is sufficiently fine-grained to enable different priorities of RPCs within a task or job for a given application.
- *Distinction between network regions with different bandwidth management policies*: The enforcement mechanism should distinguish network boundaries when adjusting network QoS. This is needed as traffic provisioning and engineering may vary significantly across different regions. For instance, bandwidth is often more constrained for WAN links—where users are typically charged for bandwidth—compared to intra-fabric bandwidth within the data center, which is generally abundant for applications, except in rare tail cases.
- *Efficient and practical deployability*: We require a method to classify RPCs according to application-level priority and then assign QoS to this traffic based on a pre-defined mapping. This must be performed without incurring excessive overhead or high resource usage.
- *Positive overall performance incentive for applications*: The new enforced QoS mapping needs to bring an overall positive performance for applications (in terms of latency). Thus, it is important that the highest priority traffic experiences minimal regression (if any) in performance after the QoS alignment.

The journey to deploying Aequitas as the default QoS authority across Google storage systems has been a long one, spanning multiple years, and there is still a lot of work to be done. We have gained valuable insights regarding the complicated interaction between network QoS and delay as well as the diverse needs of users and how that influences their view of the network. Even with a partial rollout, Aequitas is currently affecting millions of RPCs per second. Aequitas is a mature technology that has been integrated into RPCs to and from LL storage for an extended period. It has also been actively utilized for various client-facing services. Our findings can serve as a guide for other network operators aiming to differentiate traffic through QoS. The main takeaways are the following:

- **Priority is the correct network scheduling unit:** We show that prioritizing RPCs based on priority, not size, leads to significant latency reductions for critical traffic with minimal impact on lower-priority classes. This highlights that priority is the more effective unit for network scheduling.
- **Lower QoS does not imply higher latency:** Contrary to popular belief, we demonstrate that users can experience decreased latency when moving to a lower weight QoS queue if that queue is sufficiently underutilized relative to its weight (§2.3 and §4.2).
- **All can win:** We show that Aequitas is not a zero-sum game and we observe examples across production clusters where traffic at all priorities see decreased latency (§4.3). We show that these learning are consistent and hold true across different layers of the storage stack.
- **Benefits arise even under incremental deployment:** We provide production examples detailing how even under a gradual rollout process users can experience performance improvements (§5). This shows that network operators can still realize benefits from QoS assignment schemes when an all-or-nothing approach is infeasible.

The paper is organized as follows. In §2, we describe background for this project and the motivation for Aequitas, followed by a discussion of operational challenges in §3.2. Key learnings from our deployment experience are explained in §4. The detailed progress in QoS alignment is covered in §5, followed by a discussion of results and future work in §6. Related work is covered in §7, before we conclude in §8.

2 Background and Motivation

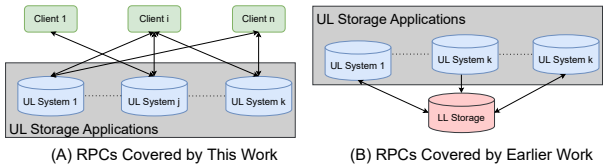
We give an overview of the Aequitas system first introduced in [14], discuss the relevant details of Google storage systems, and highlight the insights that motivated Aequitas' design.

2.1 Aequitas Overview

Aequitas operates at the RPC layer. An RPC is a programmatic request for information in which a client instructs a server to execute a procedure and receives a response. Application-level operations can be composed of hundreds of RPCs so that application performance is dependent on RPC performance. Currently, in Google production datacenters, more than 95% of application traffic is generated by RPCs [14]. Aequitas aims to inform the network of RPC level objectives and increase the number of RPCs that meet their service-level objectives (SLOs). In general, RPCs from micro-services often have SLOs that vary widely based on application needs. Here, we focus on Google storage as $\sim 75\%$ of all fleetwide non-ML related RPCs are to or from these systems [14].

Aequitas uses QoS to differentiate traffic and ensure that the most critical RPCs are given the needed network resources. The feasibility of this design was shown in [14] with results from both simulation and production rollouts. A theoretical framework was also given, based on network calculus [1], detailing the complicated relationship between QoS weights and worst-case network delay.

Figure 1 shows the high-level architecture of Google storage systems. We have n internal clients which connect to any number of k Upper Level (UL) databases (Bigtable [18], Spanner [19], etc.). These UL systems connect to Colossus [20], a



(A) RPCs Covered by This Work (B) RPCs Covered by Earlier Work

Figure 1: Storage RPCs Covered by Aequitas

common Lower Level (LL) system responsible for physically storing data. Thus, the UL systems provide different abstraction layers atop LL storage and clients can use any number of these abstractions depending on application needs.

The figure also highlights the differences between our previous results and the current work. In [14], the deployment and analysis focused on RPCs between UL systems and LL storage (Figure 1(B)). We have since expanded this to also handle traffic between clients and the various UL systems (Figure 1(A)) while also finalizing the deployments started in the previous work.

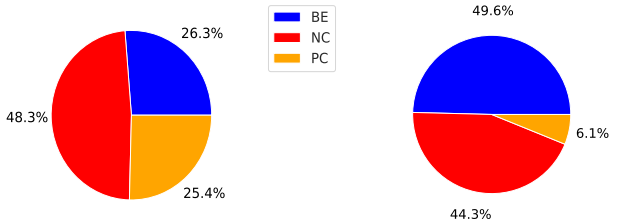
2.2 The Case for Aequitas

In [14], the notion of race-to-the-top was introduced for the first time. That is, in times of network overloads, some applications experience increased delay, causing them to miss their SLOs. In response, the corresponding developers request (and are often granted) higher QoS under the belief that a higher QoS will decrease delay and the probability of the corresponding RPCs missing their SLOs. However, these decisions are often independent of the overall QoS mix in the datacenter. In the extreme all users move their traffic to the highest QoS bands and there is no differentiation among traffic. For a given QoS band, QoS_h, there is a point at which it is no longer beneficial to upgrade traffic to QoS_h.

Over the last several years there has been an explosion of works that aim to integrate application knowledge into network scheduling decisions [21–26]. However these solutions treat all traffic belonging to an application or a coflow as identical from the network's perspective. Aequitas takes the view that there exist different classes of traffic even within one application. In our storage systems we observe that RPCs can be broadly categorized as Performance-Critical (PC), Non-Critical (NC), or Best-Effort (BE). PC RPCs have the most strict SLOs in term of latency and are generally interactive user-facing traffic. NC traffic often constitutes bulk storage operations[1], while BE RPCs are generally the background analytic traffic in our storage systems.

Figure 2 shows the priority distribution for request (response) traffic to (from) Colossus and demonstrates that there is a large amount of traffic for each of the three priorities. Moreover, the UL storage systems are composed of a multitude of jobs from clients around the globe and we see that even within an individual job multiple priorities can be used.

[1] these are large throughput-intensive procedures destined to storage



(a) Request (b) Response

Figure 2: Priority Distribution for Colossus Traffic

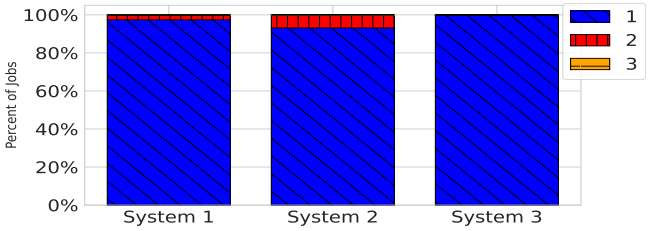


Figure 3: Percentage of Jobs Using 1, 2, or 3 Different Priorities Across Three UL Systems

Figure 3 shows the percentage of jobs that use one, two, or three priorities among BE, NC, and PC, across three different UL storage systems. We see that most jobs stick to only one priority. However, there are a small proportion of jobs that use two or more priorities and these jobs tend to be much larger, accounting for the majority of traffic. This can be seen in Figure 4 which shows the percent of request traffic in each of the three UL storage systems for jobs that use one, two, or three priorities among BE, NC, and PC. The story for response traffic from the UL systems to clients is similar.

These measurements demonstrate that simply setting a network priority at the application or job level fails to account for the different categories of traffic within a job. Instead, the evidence suggests that RPCs are the right unit for QoS enforcement as proposed in the initial Aequitas design [14]. More specifically, Aequitas takes the approach of broadly classifying RPCs as either BE, NC, or PC. It then maps these priorities to different QoS classes, implemented with Weighted Fair Queuing (WFQ)[2]. That is, BE, NC, and PC are mapped 1:1 to QoS classes QoS_l, QoS_m, and QoS_h where QoS_i corresponds to a queue with scheduling weight w_i and $w_l < w_m < w_h$. Thus the network is informed of the relative importance of different classes of RPCs, without the need to differentiate at the application level.

2.3 Determining the Correct QoS Mix

As discussed, clients are usually hesitant to change QoS, especially to one with worse latency guarantees. However, as

[2] In WFQ, switches are set up with various queues where queue i has weight w_i. Then the traffic in queue i receives w_i fraction of a link's capacity in the long run.

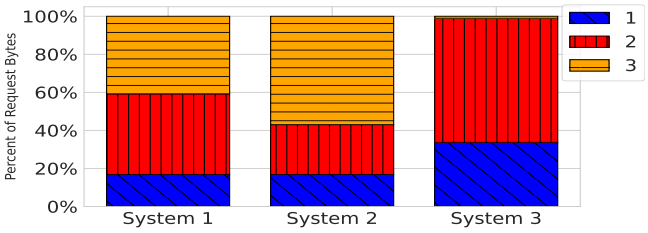


Figure 4: Percent of Request Traffic for Jobs Using 1, 2, or 3 Different Priorities Across Three UL Systems

evidenced by the race-to-the-top phenomenon, clients' QoS selection rules are usually arbitrary. So we aim to answer the following question: *what are the conditions under which a user should prefer one QoS class over another?*

Consider a WFQ with k QoS classes corresponding to k queues with weights $w_0, w_1, ..., w_{k-1}$. We let λ_i be the total load on QoS_i and let β_i^c be the load belonging to client c on QoS_i. Pick two of these QoS classes, QoS_0 and QoS_1 with $w_0 > w_1$. We can show that under a mild set of assumptions, client c can achieve a better service rate by shifting a small amount ε of traffic from QoS_0 to QoS_1 if

$$w_1\left[\frac{\lambda_1-\beta_1^c}{\lambda_1(\lambda_1+\varepsilon)}\right] > w_0\left[\frac{\lambda_0-\beta_0^c}{\lambda_0(\lambda_0+\varepsilon)}\right] \quad (1)$$

This improved service rate occurs despite the fact that a small amount of traffic is moved to a lower weight QoS band and stems from the fact that QoS_0 is overloaded relative to QoS_1. More details on this can be found in Appendix A.

This serves to illustrate that a QoS class with a higher scheduling weight does not always imply better service. The work in [14] showed the conditions for priority inversion (when a higher weight QoS class has worse performance as compared to a lower weight QoS class). But that analysis focused on all traffic sharing the same queue. Here we analyzed the conditions under which a particular user receives better service on one QoS class compared to another. We see that there are scenarios in which it is beneficial for a user to transfer a portion of their traffic from a QoS class with a higher weight to one with a lower weight. Thus, there are situations in which priority inversion exists. Indeed, in §4 we see scenarios in Google production datacenters in which users benefit from a lower weight QoS class.

As the storage systems represent a significant portion of Google traffic, we desire that the mix of (QoS_l, QoS_m, QoS_h) within the storage systems is close to an ideal QoS mix fleetwide. The Google-wide mix should not have any priority inversions, and so for a particular application a we say that a has an unhealthy mix for (QoS_i, QoS_j) with $w_i > w_j$ if β_i^a:β_j^a is greater than w_i:w_j. Note, if a is the only job in the cluster then this is equivalent to the notion of priority inversion we outlined above; a has overloaded QoS_i relative to its weight as compared to QoS_j so that we would expect moving a sufficiently small volume of a's traffic from QoS_i to QoS_j would yield a better service rate. Thus, this is a restriction of priority inversion to a specific application and serves to identify applications or users that are most responsible for an overall unhealthy QoS mix. Appendix B examines instances of unhealthy QoS in Google storage systems.

In the presence of priority inversion, performance on each of the QoS classes becomes unpredictable, making it difficult to decide to which QoS queue traffic should be assigned. RPCs in the wrong queue can miss their SLOs when the latency experienced is higher than the queue weights would suggest. Aequitas' goal is thus to manage the assignment of traffic to the different QoS queues in order to avoid priority inversion and thus provide more predictable performance for each queue. This in turn ensures that SLOs for the given queues can be set accordingly so that traffic priorities can be assigned appropriately, leading to fewer SLO violations.

3 Deployment: Strategy and Challenges

In this section we discuss our strategy to deploy Aequitas and challenges faced along the way.

3.1 Deployment Strategy

At the surface, it appears that the work required to extend Aequitas is fairly straightforward and is simply a matter of enforcing QoS mappings based on application priority. However, even the initial step of defining these mappings is non-trivial as RPCs are scattered across multiple storage systems, each with their own notion of priority. This disparate collection of priorities must each be mapped to the same finite number of QoS queues, while accounting for the different SLAs across these systems. Moreover, in contrast to our previous results, we are now handling RPCs that either directly originate from or are destined to end users. These clients have a diverse set of demands on the network and many directly measure RPC performance. Thus, a large number of users have configured their own QoS and do not necessarily want to conform to the QoS assignment dictated by the mappings, making it difficult to coordinate large-scale changes.

On a per RPC basis, Aequitas leverages information about the source and destination IP addresses as well as the application priority and requested QoS to make its decisions. However, which packets of an RPC are affected by Aequitas' selection depends on the destination. The LL storage is often obscured from users and as a result the Aequitas team elected for a server-side implementation for QoS enforcement on the traffic between UL and LL systems. Under this approach, the first few packets of an RPC are sent using the original, unaltered QoS. Upon arriving at an LL server, Aequitas chooses a QoS for the RPC based on priority and all subsequent packets sent on this connection, in either direction, use the updated QoS. Such an approach allows for quicker adoption as all

users interacting with the same server experience the change.

In contrast, each of the UL systems is visible to clients and clients monitor RPC-level metrics for the storage systems with which they interact. As such, a server-side implementation in these UL systems would obscure information from users and would not account for their varied SLOs. For example, in the case of Spanner, client-side implementations allow Spanner clients to see the QoS changes of their RPCs before they are sent over the network. In contrast, under a server-side implementation, the client's chosen QoS is reflected in the Spanner client library but the actual QoS chosen by Aequitas at the server can be different, causing a mismatch. This obscures information from users, making it unclear why this mismatch occurred. The Aequitas team therefore decided on client side implementations to expose these changes to users and to more easily reason about how the changes affect different clients. In this setting, Aequitas intercepts RPCs before they enter the network and chooses the correct channel based on the decided QoS. Note that clients do not directly interact with Aequitas. Each storage system has its own API for setting priority and Aequitas is integrated into these APIs to seamlessly pick QoS based on the pre-defined mappings. Appendix C discusses the impact these decisions had on deployment.

The Aequitas implementation is currently restricted to intra-cluster traffic. Aequitas over the WAN is a much more challenging issue due to the longer network latencies and the interaction with BwE [27]. While the proportion of intra-cluster RPCs varies across time and clusters, it accounts for a significant portion of all traffic. Customers typically try to keep traffic within the same cluster due to the inherently lower latency. Aequitas uses data in the backend UL systems to map source and destination IP addresses to clusters and determine which traffic is eligible for QoS enforcement.

3.2 Operational Challenges

Downgrading QoS: By far, the most challenging aspect of Aequitas was convincing individual clients and the teams managing the various storage systems to assign a lower QoS than originally requested for some RPCs. Aequitas distributes RPCs across QoS_l, QoS_m, and QoS_h by RPC priority.

Figure 5 shows the distribution of requested QoS within priority for all request traffic associated with Spanner. There is a high degree of misalignment between the scheme proposed by Aequitas and the current state of the cluster. In particular, there is heavy usage of QoS_h for NC RPCs and more than half of the BE RPCs use QoS_m. There is very little usage of QoS_l acoss all priorities. Imposing Aequitas across the cluster leads to both QoS downgrades (ie. BE RPCs on QoS_h downgraded to QoS_l) and QoS upgrades (ie. PC RPCs on QoS_m upgraded to QoS_h). We see similar misalignment across clusters and various storage systems, for both request and response traffic.

As discussed in §2.2, the notion of race-to-the-top suggests that users are often supportive of moving traffic to a higher

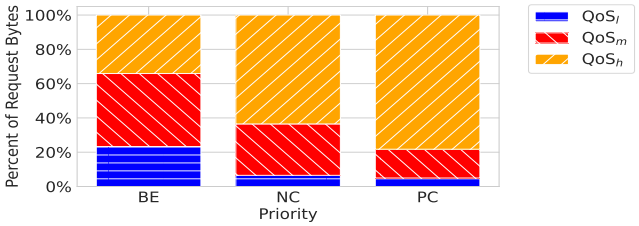


Figure 5: Intra-Cluster QoS Breakdown Within Priority (by Request Bytes)

QoS. This is due to a commonly held belief that higher QoS implies better network performance. This implies that the converse is also a widespread belief: lower QoS leads to worse network performance. §2.3 argued that this is not always the case and traffic can benefit from lower QoS when that QoS class is underloaded relative to its weight. Yet, there can also be cases where users have carefully tailored QoS in order to meet their SLOs and a change in QoS can be the difference between acceptable performance and SLO violations.

As such, enabling Aequitas fleet-wide within a short time could cause multiple production issues. Instead, the Aequitas team used a gradual approach; in some cases enabling the change for individual clients and monitoring performance before moving onto cluster level or multi-cluster rollouts. While this required considerable effort from the team and greatly extended the timeline for widespread adoption, it also allowed for detailed studies of the effects of Aequitas to understand the impact on individual users as well as entire clusters.

Avoiding other Performance Bottlenecks: While Aequitas aims to avoid priority inversion and help more applications meet their SLOs, we do not want to introduce bottlenecks in other areas. A reasonable measure to assess the impact of Aequitas is how it affects RPC network latency (RNL), which is the time that the RPC spends in the network. However, not all developers have intimate knowledge of how the network functions and thus often do not know how to correctly configure network QoS, one of the main motivations for Aequitas.

Developers put most of their attention on end-to-end RPC performance metrics like total RPC latency, memory usage and other general performance indicators. While Aequitas can improve overall network performance through better QoS configuration, for an individual RPC the choice of QoS must be made with minimal overhead to avoid adding unnecessary latency at other points throughout the life of an RPC.

In our evaluations we consider both RNL and overall RPC latency. Throughout the rollout process we also tracked memory usage, error rates, cache misses, and many other metrics, observing no noticeable changes attributed to Aequitas.

Inferring Effects under Incremental Changes: As detailed earlier, due to both Google's scale and availability concerns, we cannot simply roll out Aequitas for all users simultaneously. This challenge is well-known in practice and several methods for running experiments on production servers

while ensuring system availability have been proposed previously [28]. This is especially challenging for client side implementations and often led to several user-level pilot rollouts before moving to cluster level rollouts.

As discussed in [14] and further argued in §2.3, for a particular client the performance of their traffic on one QoS band compared to another depends not only on the QoS distribution of their traffic but also the QoS distribution of all traffic traversing the same switches. This is in addition to all the usual network factors that impact performance: overall load, burstiness of traffic, presence of congestion, etc. This implies that enabling Aequitas for a few clients or a few servers at a time does not paint an accurate picture of its effects. To fully understand the impact of Aequitas requires measurements when no traffic uses Aequitas versus measurements after the policy has been enabled for all RPCs.

However, due to the incremental rollout strategy, the traffic distributions can change significantly between a point in time before Aequitas is enabled and when Aequitas has fully rolled out. This is especially true in the case of client side changes; clients are often synced to different versions of the code. Thus, even in cases where we can enable Aequitas for all clients simultaneously there is often a significant time gap between when the first client picks up the change and when the change is adopted by the final user. Indeed, we saw production examples in which the distribution of a clients' RPCs across priorities greatly changed during the rollout. In the extreme, we observed scenarios where a user's volume of PC traffic grew by 5x during rollout. We thus adopted two major strategies as part of the deployment process.

The first was to target clusters in which a small number of users (ideally one) account for the majority of the cluster's traffic. In this way, once these small number of users have adopted Aequitas we can obtain accurate measures of overall cluster performance. Moreover, it normally requires significantly less time for a small group of users to adopt the change as compared to all clients using the cluster, allowing for a more accurate before and after comparison; in the case of a single dominant user we can measure performance just before the user adopts the change versus performance immediately after adoption. Note that there are in fact clusters at Google that exhibit this behaviour. We have users that account for more than 50% of request and response traffic within a cluster. As a result, applying Aequitas to such a client provides a good indication of overall cluster performance.

Second we implemented a feature that allows us to enable Aequitas for a fraction of traffic, either at the cluster level or at the granularity of individual clients. For example, enabling at 50% for one client means that 50% of the client's RPCs will be randomly chosen to be included in Aequitas whereas the remaining RPCs are unaffected. This is especially useful in cases not covered by the first strategy, where cluster traffic is composed of many small clients, with no client dominating. This random sampling adequately controls for other confounding factors and allows us to tease out Aequitas' effects by comparing traffic sampled versus the RPCs not sampled[3]. This approach is also more appealing to individual customers and owners of the storage systems as it offers a more conservative alternative to the all-or-nothing approach where clients or clusters are either in Aequitas or not.

4 Key Learnings from Production Examples

We next present case studies that highlight key insights we have gained from enabling Aequitas across a diverse user base. As alluded to earlier, the actual code changes for Aequitas are only a small part of the overall challenge. The biggest obstacle is integrating the changes for various clusters that span the globe. In §4.1 we discuss the methodology used to perform analysis and infer the affects of Aequitas. §4.2 and §4.3 provide examples demonstrating our main takeaways.

4.1 Analysis Methodology

Aequitas aims to maximize the volume of PC RPCs that meet their network SLOs. This means lowering the RNL for PC RPCs, but without introducing intolerable latency to NC and BE RPCs. We also want to ensure that Aequitas does not lead to performance regressions in other areas, such as increasing overall latency at the expense of decreased RNL latency, increased memory usage, or general connection errors.

Throughout this section we present experiments that compare traffic that obeys the Aequitas policy (aligned) against traffic that does not (misaligned). For each of these experiments, 50% of traffic was randomly sampled for Aequitas. While cluster utilization and latency numbers can fluctuate during the experiment, we chose highly loaded clusters for these studies and users reported no adverse effects in terms of increased packet loss nor unacceptable memory usage. In fact, in some cases where Aequitas reduced latency we also observed decreased packet loss rates and higher throughput. During Aequitas rollouts, users have a general idea that the change is coming but they are not aware of the exact time of change as it depends on internal deployment cycles. While not a true A/B experiment, we made efforts to simulate it as closely as possible with methods like random sampling to control for other confounding factors. A true A/B experiment is not feasible in our production environments.

Monarch [29] allows clients to monitor the performance of their jobs. It exposes dashboards showing job performance across several key metrics, aggregated over all the RPCs of the job. As detailed in [30], it is not possible to collect a vast array of metrics for every RPC due to Google's scale. Thus, these dashboards allow users to assess overall performance

[3] While there are other sampling techniques to more directly control for confounding factors, we find that either we do not have a priori information to perform such sampling (in the case of RPC size) or that it would add unnecessary overhead (in the case of priority based sampling). However, we see that due to the scale of the Aequitas experiments, random sampling often does a good job of controlling other confounding factors.

but do not contain all the metrics necessary to assess the impact of Aequitas. Crucially, Monarch does not contain the application-specific priorities as these differ by application and collecting these would limit Monarch's scalability.

Instead, Dapper [31] is our primary analysis tool. Dapper is Google's distributed tracing infrastructure that exposes intuitive APIs for developers to add custom annotations to their traces. Fathom [30] metrics have been integrated into Dapper, providing fine-grained decomposition of latency numbers attributed to the various systems traversed by the RPC. We also export custom annotations from the Aequitas logic indicating the priority of the RPC, the requested QoS, and the QoS chosen by Aequitas. This allows us to examine RNL latency, and its contributing factors, on a per priority basis.

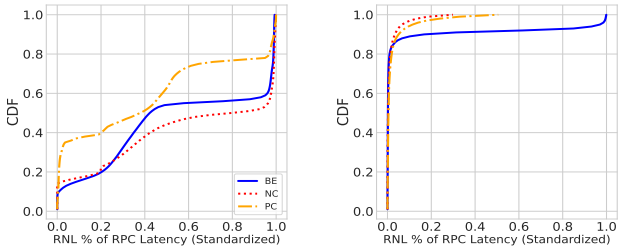
In order to operate Dapper at Google's scale, the RPCs are necessarily sampled. Sampling rates can differ depending on characteristics such as job size, RPC importance, and frequency of errors (with higher error rates necessitating higher sampling to debug performance issues). To account for this, the Aequitas analysis restricts to uniformly sampled Dapper traces and weights each trace by the inverse of its sampling rate. For example, consider three traces from Dapper, one of which represents an RPC of size 1MB sampled at rate 1, while the other two traces represent RPCs having size 2MB and are sampled at rate $\frac{1}{2}$. Then this represents a total of 5 RPCs ($\frac{1}{1} + \frac{1}{1/2} + \frac{1}{1/2}$) having an average size of 1.8MB ($1 * 1$MB $+2 * 2$MB $+2 * 2$MB = 9MB / 5 RPCs).

To assess the performance of Aequitas we are not only interested in average performance but also behaviour at the tail. There is a concern that due to sampling, some behaviour at the extreme is not accounted for. Thus we supplement our Dapper analysis by comparing the results to the high level metrics exposed in Monarch dashboards. We do this for clients with the most traffic impacted by Aequitas and continue to monitor performance after Aequitas adoption. We observe that the findings from the two tools are consistent.

4.2 Key Finding 1: Priority, not Size, is the Correct Network Scheduling Unit

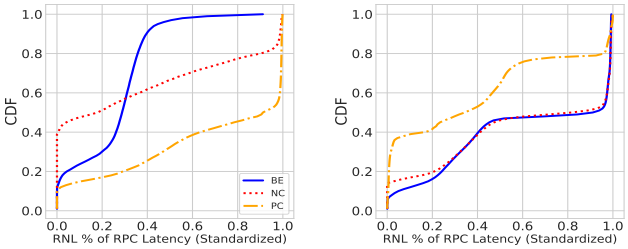
As the name suggests, PC RPCs are the most critical, and thus we want these RPCs to spend as little time as possible in the network. Instead these RPCs should spend most of their time in computation. In contrast, BE is treated as a scavenger class for RPCs, which can thus tolerate slightly increased time in the network in favor of prioritizing NC and PC RPCs [32]. We thus examine RNL as a percent of total RPC latency and would expect that this quantity is lower for PC RPCs as compared to BE or NC RPCs. We deem this quantity as RNL proportion. RNL is also the portion of total latency that the network has control over, as discussed in [14].

In Figure 6, we examine RNL proportion across priorities and RPC sizes. This snapshot was taken from Spanner when



(a) RPC size < 1KB (b) RPC size $\geq$ 2MB

Figure 6: RNL for different RPC sizes under High Misalignment



(a) RPCs on QoS$_l$ (b) RPCs on QoS$_h$

Figure 7: RNL for small RPCs ([0, 1KB)) across different QoS under High Misalignment

there was still a high degree of misalignment[4], allowing us to tease out the effects of QoS. These latency numbers are taken across clusters with differing load characteristics: from underutilized to clusters at or above capacity. Figure 6a shows results for small RPCs ([0, 1KB)) while Figure 6b shows the same plot but for RPCs that are at least 2MB. In both plots we normalize by the maximum RNL proportion so that both figures range from 0.0 to 1.0 with at least one value achieving 1.0. In the case of large RPCs, both NC and PC traffic have a much shorter tail compared to BE, as expected. Moreover, for small RPCs, RNL accounts for a smaller portion of total latency for PC traffic as compared to BE and NC. However, as there is still a high degree of misalignment in this snapshot, we observe that for small RPCs median RNL proportion is higher for NC as compared to BE.

To further understand the effect of QoS on RNL, Figure 7 looks at RPC performance across priorities for different QoS for small RPCs. These numbers were also taken across all clusters showing varying network load profiles. Figure 7a depicts RPCs that use QoS$_l$ whereas Figure 7b shows RPCs using QoS$_h$. In both figures the values are normalized by the maximum value of RNL proportion across all small RPCs, regardless of QoS, to facilitate a comparison across figures. We see a substantial reduction in RNL proportion for PC

[4] As this is production traffic, some traffic is naturally already aligned and so we cannot compare to a perfect control with 100% misalignment

	p50		p95		p99	
Priority	**Mean**	**Max**	**Mean**	**Max**	**Mean**	**Max**
BE	0.02	0.04	0.14	0.17	1.02	1.55
NC	0.02	0.04	0.17	0.56	0.42	1.75
PC	0.17	-0.73	-1.49	-34.01	-3.89	-263.95

Table 1: Aligned RNL Improvement over Misaligned by Priority (Negative Values Indicate Improvement)

	p50		p95		p99	
Priority	**Mean**	**Max**	**Mean**	**Max**	**Mean**	**Max**
BE	0.00	0.00	0.00	-0.02	0.00	-1.82
NC	0.00	0.00	0.00	0.05	-0.04	-3.65
PC	-3.06	-3.50	-81.57	-2419.22	-134.45	-2419.51
All	0.00	0.00	-0.11	-0.17	-0.66	-5.66

Table 2: Aligned RPC Latency Improvement over Misaligned by Priority (Negative Values Indicate Improvement)

RPCs for traffic using QoS_h as compared to QoS_l. Moreover, we see high median and tail numbers for BE and NC RPCs even when they use QoS_h, with slightly better performance for NC RPCs. RNL represents a larger proportion of total latency for NC RPCs on QoS_l as compared to BE traffic. This suggests that better prioritization in the network, in the form of higher-weight QoS, has a more significant impact on PC RPCs as compared to lower priority RPCs and BE traffic is the least sensitive to network prioritization.

Note that Figure 6 shows that both small and large RPCs are distributed across all three priorities, suggesting that prioritizing based on RPC size is too coarse grained and fails to capture the needs of RPCs operating on different priorities. This is consistent with the findings in [14] and combined with the above analysis demonstrates that Aequitas can correctly translate priorities to the network through its QoS mapping.

To further tease out the effects of Aequitas, we look at a particular client (denoted c) in a large, highly loaded cluster. c accounts for over 50% of all traffic in the cluster and was one of the initial pilot customers of Aequitas for Spanner. We look at a snapshot when Aequitas was rolled out at 50% for c. In the absence of Aequitas, all of c's traffic uses QoS_h with a small amount of traffic on QoS_m. Thus Aequitas upgrades a small fraction of c's PC traffic on QoS_m and either downgrades or leaves unchanged all of c's BE and NC traffic.

Table 1 shows the improvements (or regressions) in RNL for traffic aligned under Aequitas versus misaligned traffic for c. A CDF of RPC performance is computed every hour for a 4-day window and then we compute the mean and maximum values seen over the analysis period for the p50, p95, and p99 for each priority. These calculations are done for both aligned and misaligned traffic and then the differences are normalized by the standard deviation of p99 of misaligned traffic across all 3 priorities[5]. For example, if $s^c_{RNL,m}$ is the standard deviation of p99 RNL of misaligned traffic, then the mean p99 of -3.89 for PC traffic means that the 99^{th} percentile latency averaged over all the 1-hour windows for aligned traffic is lower than misaligned traffic by $3.89 \cdot s^c_{RNL,m}$. We see that while the alignment increases RNL latency for BE and NC traffic, this is a small fraction of the reduction in latency experienced by PC RPCs. The reduction in maximum p99 latency for PC traffic is over 150x (263.95 / 1.75) the increase in maximum p99 latency for NC RPCs.

Table 2 shows the same statistics for total RPC latency. Here we normalize by the standard deviation of p99 RPC latency for misaligned traffic, denoted $s^c_{RPC,m}$. We observe practically no change in the total latency numbers between aligned and misaligned traffic at p50 and p95 for BE and NC RPCs. Somewhat surprisingly however, the maximum p99 values see a reduction for both BE and NC traffic. While it is difficult to definitively determine the causes of these reductions, analysis suggests that a small fraction of BE and NC RPCs experience longer queuing delays in the QoS_h queues due to the high volume of traffic assigned to this queue (in the absence of Aequitas, more than 99% of client c's traffic uses this QoS class) and Aequitas is able to better isolate traffic in the different queues, causing less contention overall.

The table also highlights significant reductions in PC RPC latency numbers at all the measured percentiles. The standard deviation of p99 of RPC latency is an order of magnitude larger than the standard deviation of p99 RNL, so these reductions in total RPC latency are significantly larger than the corresponding reductions in RNL. This suggests that for many misaligned PC RPCs, network is the bottleneck, as is consistent with the results in Figure 10 of [32]. By assigning this traffic to the highest QoS we can effectively alleviate the bottleneck without introducing major latency degradation for traffic using the other priorities. The last row of Table 2 shows the same statistics, but aggregated across all 3 priorities. Here we see far less extreme values than those seen for PC RPCs as around 90% of client c's traffic uses BE or NC. However, we still observe a large reduction in maximum p99 RPC latency for aligned traffic compared to misaligned.

These results demonstrate that PC RPCs are far more sensitive to network latency as compared to BE and NC traffic. Thus, by assigning PC traffic to the highest QoS class, Aequitas can significantly reduce both RNL and overall RPC latency for this traffic with comparatively small increases for BE and/or NC RPCs. Moreover, other clients exhibited similar performance and we continued to observe behaviour consistent with these findings over longer time horizons and as the rollout reached 100% for various users across diverse clusters with different load characteristics.

4.3 Key Findings 2&3: Lower QoS does not imply Higher Latency, All can win

As discussed in §2.3, for QoS_i and QoS_j with $w_i > w_j$ it is possible for a client to experience lower latency on QoS_j, despite the lower weight, if QoS_i is significantly overloaded

[5] This is done to keep Google latency numbers confidential and to facilitate comparison across different clients and types of latency.

	QoS_m:QoS_l	QoS_h:QoS_l	QoS_h:QoS_m
Unhealthy Cutoffs	4:1	8:1	2:1
Mix in Cluster	10.69:1	0.52:1	5.54:1

Table 3: Unhealthy QoS mix thresholds and actual QoS mix in one cluster prior to Aequitas rollout

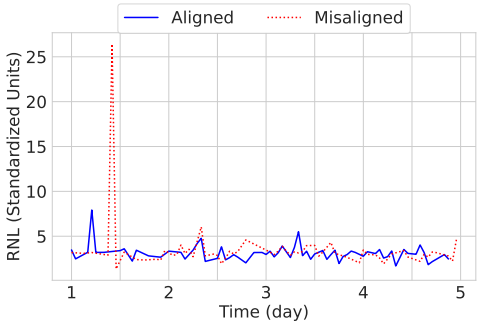


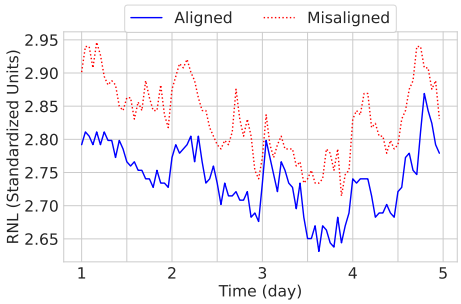
Figure 8: Aligned vs Misaligned p99 RNL for BE traffic of client $\bar{c}$

relative to QoS_j. Here we show a production example of this phenomenon. We piloted changes with a large Spanner client (denoted $\bar{c}$) that had previously exclusively used QoS_m across all three priorities. When approached for the pilot study, client $\bar{c}$ was very willing to participate and acknowledged that the current QoS settings had been decided through trial-and-error without any definitive guiding principle. The Aequitas team started a pilot in one cluster before expanding fleetwide for this user. This user was chosen as they have a high volume of traffic across several clusters that routinely experience high utilization. We study the result of a 50% rollout in one such cluster to highlight the behavior of interest.

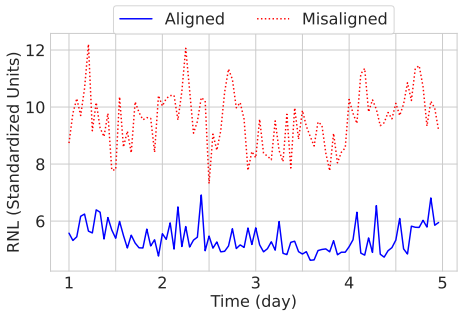
The ratio of weights for QoS_h:QoS_m:QoS_l is 8:4:1. Using the terminology of §2.3 the cluster has an unhealthy QoS mix between QoS_l and QoS_m if the volume of traffic on QoS_m is more than 4 times the amount of traffic on QoS_l. The first row of Table 3 shows the unhealthy QoS cutoffs for the 3 pairs of QoS bands. Ratios higher than these cutoffs correspond to an unhealthy mix. The second row shows the actual mix within the cluster, prior to enabling Aequitas at 50% for $\bar{c}$. QoS_m is severely overloaded relative to QoS_l. Aequitas downgrades BE traffic to QoS_l, so using the analysis of §2.3 we would expect this downgraded traffic to experience reduced latency compared to its misaligned counterpart.

Indeed, Figure 8 shows the p99 RNL for aligned and misaligned BE RPCs. For both types of BE traffic we compute a CDF every hour for a four day period and then normalize by $s^{\bar{c}}_{RNL,m}$, the p99 RNL of all misaligned traffic for client $\bar{c}$.

The average p99 BE for aligned traffic is lower than the corresponding misaligned traffic by $0.35 \cdot s^{\bar{c}}_{RNL,m}$. Moreover, the maximum across all p99 values is $18.51 \cdot s^{\bar{c}}_{RNL,m}$ lower for BE aligned traffic as compared to BE misaligned traffic. The standard deviation of 99^{th} percentile for BE traffic is $2.15 \cdot s^{\bar{c}}_{RNL,m}$ lower for aligned RPCs. Thus, we see that despite



(a) p50



(b) p99

Figure 9: Aligned vs Misaligned RNL aggregated across all priorities for client $\bar{c}$ at the (a) 50^{th} and (b) 99^{th} percentiles

the fact that aligned traffic uses QoS_l while misaligned traffic uses QoS_m, aligned traffic experiences lower latency as QoS_l is severely underloaded relative to QoS_m. In addition, the p99 BE RNL for aligned traffic is more stable over time as compared to misaligned RPCs, providing more predictability for $\bar{c}$ in terms of worst case behaviour of BE traffic.

Figure 9 shows the p50 and p99 for aligned and misaligned traffic sampled from the same user aggregated across all priorities. Again, these numbers are computed hourly over a four day period and normalized by $s^{\bar{c}}_{RNL,m}$. We see clear differences in performance between aligned and misaligned latency with the aligned percentiles showing lower latency values, even at p50. While the gap between aligned and misaligned RNL is not large at p50 (aligned RNL is $0.09 \cdot s^{\bar{c}}_{RNL,m}$ lower on average), the aligned RNL is always lower than its misaligned counterpart, while for p99 we see a larger separation. On average, the aligned values are $4.19 \cdot s^{\bar{c}}_{RNL,m}$ lower than the misaligned values. The maximum p99 BE RNL for aligned traffic is $5.29 \cdot s^{\bar{c}}_{RNL,m}$ lower than the maximum BE RNL for misaligned RPCs. Moreover, QoS queues can sometimes become overloaded due to a high volume of traffic at that QoS, leading to latency spikes. We consistently observed that these spikes were more common with misaligned (non-Aequitas) traffic vs the Aequitas counterpart which shows smoother, more predictable latency, as evidenced by the figure.

Comparing Figures 8 and 9b, we see that on average the reduction in aligned RNL across all priorities is larger than the reduction for BE traffic. However, the reduction in maximum

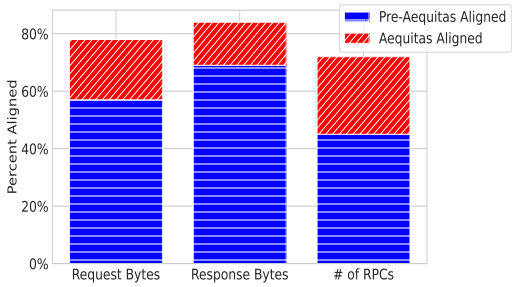


Figure 10: Aequitas Alignment Status of Spanner by Response bytes, Request bytes, and Number of RPCs

for BE traffic is larger as compared to the reduction across all priorities. This not only shows that we can decrease the latency of PC traffic through QoS upgrades but that Aequitas can also lead to improvements in worst-case behaviour for BE traffic by reassigning it to a lower, under-utilized QoS band. In the long run, this also benefits all traffic on the overloaded QoS_m band due to the reduced load at this QoS, lowering contention for RPCs in this queue. After the full rollout for client $\bar{c}$ we observe a 31.04% reduction in RNL for NC RPCs, despite the fact that these RPCs were already aligned before the Aequitas implementation. The results for BE and PC traffic after full rollout are consistent with those at 50% rollout.

This demonstrates that QoS alignment under Aequitas is not a pure zero-sum game and in fact all traffic under a particular user can benefit from better alignment. This is consistent with the theoretical justification given in Appendix A.

5 Fleetwide Performance

Previously, we provided examples of key findings from enabling Aequitas for a diverse set of clients. Here we discuss overall Aequitas performance across Google storage systems and argue that benefits of correct QoS alignment arise even under incremental deployment. Deployment results in [14] are related to traffic to/from the HDD LL storage systems. We have expanded Aequitas to also affect SSD LL storage traffic. The results for SSD LL storage traffic show that the improvements brought by correct QoS enforcement hold true across different layers of the storage stack. The detailed results are discuess in Appendix D.

As mentioned, Aequitas is implemented on the client-side for UL systems, making rollout much more gradual as compared to LL storage. Figure 10 shows the progress for a planet-scale UL system. The quantities for **Pre-Aequitas Aligned** represent the alignment status of the system in the absence of Aequitas, whereas the values for **Aequitas Aligned** represent the additional alignment gained through Aequitas deployment. While the marginal alignment through Aequitas can be small in some cases ($\sim$ 15% of Response bytes), without Aequitas the **Pre-Aequitas Aligned** RPC alignment is not enforced and it can change over time. As discussed in §2, we observe race-to-the-top QoS behavior with users selecting higher QoS for their jobs after SLO violations. Thus, we would expect to see the **Pre-Aequitas Aligned** numbers shrink over time without enforcement. With Aequitas, the alignment is the combination of **Pre-Aequitas Aligned** and **Aequitas Aligned** so that $\sim$ 72% of RPCs are aligned, representing $\sim$ 84% of all response bytes and $\sim$ 78% of all request bytes.

The Aequitas team conducted pilot studies for four of the five largest customers of Spanner and have made Aequitas the default in all clusters. However, some clients have developed a set of complex QoS assignment policies in collaboration with other internal teams and are not currently included in the Aequitas logic. We are currently tackling these exceptional cases, but even in the current state we have a large proportion of RPCs under Aequitas and clients can still see benefits. This shows the power of Aequitas even under partial rollouts.

The first pilot customer from Spanner was a large query service that operates across the fleet. This service can account for over 15% of all RPCs in the system and operates on PBs of data daily. Utilization across clusters varies with time. However, as this service is a crucial part of Google's infrastructure, it is exposed to varying load profiles across the fleet.

Figure 11 shows the priority distribution of the client's response traffic. The workloads are more response heavy (corresponding to reads) and the traffic is dominated by NC and PC RPCs with very little BE usage. The request traffic distribution is more PC-heavy due to the importance of reliable low-latency writes.

Figure 12 examines the QoS distribution of response traffic by priority for the service. The situation for request traffic is similar. The client almost exclusively uses QoS_m and QoS_h, even for its small number of BE RPCs. There is also a high degree of misalignment, especially in the case of BE and NC traffic. Figure 13 further shows that this misalignment varies across cells. There are many cells with a high degree of misalignment (up to 100%) regardless of whether we aggregate by request bytes, response bytes, or number of RPCs.

After Aequitas, almost no traffic is misaligned. The rollout process for this client was gradual, increasing the percentage of RPCs affected in several canary cells before expanding fleetwide. However, after complete rollout, the Aequitas team made a change to facilitate expanding the logic to other customers and inadvertently turned off Aequitas for this service. While this was unintentional, it allowed us to gather data just before enabling Aequitas again across all the clusters.

Figure 14a shows the p99 RNL hourly over a 4 day period for NC and PC RPCs prior to re-enabling Aequitas. Figure 14b shows the same data but for a 4 day period after the Aequitas rollout with the days ranging from 8 to 12 as this is a 4-day period exactly one week after the results in Figure 14a. Both plots are standardized by the value of PC latency. We omit BE numbers as there was almost no BE traffic in the pre-Aequitas period. By monitoring over a longer time horizon, we witnessed only a slight regression in BE RNL and there have been no episodes of intolerable latency in over 2

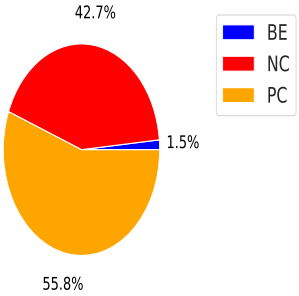


Figure 11: Priority Distribution of Query Service for Response Traffic

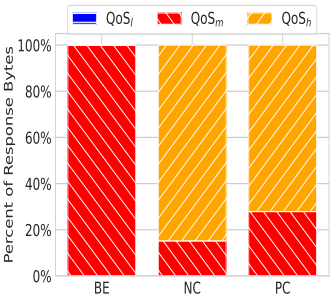


Figure 12: QoS Distribution of Response Traffic for Query Service

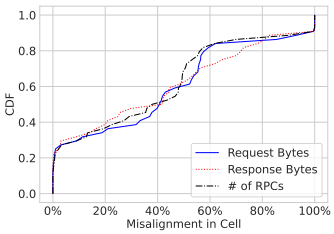
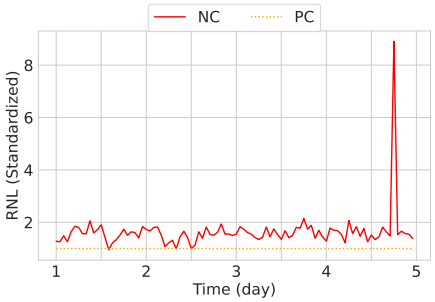
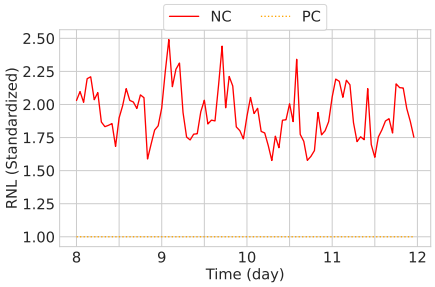


Figure 13: Query Service Misalignment Across Cells prior to Aequitas



(a) pre-Aequitas



(b) post-Aequitas

Figure 14: Latency by Priority (Standardized to PC Latency)

	Avg	Max	Std Dev
NC	16.24%	−68.91%	−68.62%
PC	−3.77%	−36.45%	−70.03%

Table 4: Percentage Change in p99 RNL Statistics for a Query Service Post-Aequitas (Negative Values Imply Improvement)

years since rollout. The small regression for BE is also much smaller than the improvement for PC.

Before Aequitas, the gap in latency between NC and PC RPCs was often small and at times NC p99 RNL was smaller than PC p99 RNL, indicating priority inversion. Post-rollout there is a clear separation in RNL between NC and PC RPCs with the p99 RNL for NC RPCs always more than 1.5x that of the p99 RNL for PC RPCs. We also see a point pre-Aequitas where NC RNL rose to more than 8x that of the PC RNL. Post-Aequitas, NC never reaches 2.5x higher. However, this only shows better separation in RNL between the 2 priorities.

Table 4 shows that the average, maximum, and standard deviation of PC p99 RNL is decreased post-rollout. While we witnessed a marginal increase in average NC p99 RNL, the maximum and standard deviation both decreased. Thus, Aequitas effectively differentiates between RPC priorities. It also leads to more stability overall with improvement in worst case p99 RNL for both PC and NC. The reductions in standard deviation are also expected as now each priority is mapped to a unique QoS, whereas in the absence of Aequitas the same priority can operate across different QoS.

Aequitas has been deployed to LL storage for more than 3 years whereas the query service has been using Aequitas for over 2.5 years. The results discussed have continued to be true over time, demonstrating the robustness of the scheme.

6 Discussion

Generalizability: As outlined in §2.2, Aequitas uses metadata from each storage system to classify RPCs as BE, NC, or PC. These RPC classes are then mapped 1:1 to the QoS classes QoS$_l$, QoS$_m$, and QoS$_h$, respectively. This is a flexible design in the sense that any available metadata can be used to classify RPCs into the three priority classes. In general, the Aequitas system accepts a set of RPC metadata features $x_0, x_1, ..., x_k$ from the application and uses these features to map the RPC to one of the priority classes.

This also gives a strategy to extend Aequitas to other Google systems; we simply classify traffic from these systems into a finite number of priorities that adhere to an ordering. These priorities are then mapped to QoS queues such that the distribution among the queues maintains a healthy QoS mix as described in §2.3. Similarly, Aequitas can be extended to networks outside of Google. In addition to the above, networks would need the ability to enforce differentiated service levels in the form of switch scheduling support.

Extending Aequitas to the WAN: The current implementation is restricted to RPCs having a source and destination in the same cluster. A promising future direction is to extend the implementation to the WAN but there are several features of WAN traffic that make this implementation more challenging. First, latencies inside a cluster are typically only a small

fraction of total latency. In contrast, network latency between clusters can be orders of magnitude larger, thus dominating the breakdown of total latency. Thus, changing QoS of such an RPC can have a profound affect on its latency. As such, changes that assign traffic to a lower QoS as compared to what they are currently using must be carefully managed.

Second, BwE [27] is responsible for bandwidth enforcement over the WAN. It allocates bandwidth to applications based on priority. From a network user's perspective this is seen as quota for the amount of traffic at a given priority level. In times of congestion, when usage exceeds capacity, traffic may need to be downgraded to a lower priority. Currently, this downgrading does not account for application-level considerations. So an Aequitas-for-WAN extension would have to work with BwE to ensure that the right QoS is set across various applications. Yet in doing so, Aequitas must be careful not to upgrade RPCs in such a way that the total usage at a given priority exceeds its assigned capacity, leading to BwE throttlings. This is an avenue for future work.

7 Related Work

Congestion Control: Typical congestion control algorithms, such as the Additive-Increase Multiplicative-Decrease (AIMD) approaches [33, 34] focus on trying to find a close-to-ideal tradeoff betwen high utilization and low delay. But these schemes wait for loss to occur and then adjust sending rates, adding unnecessary delays as they reach the send threshold. More recent schemes try to avoid loss altogether by inferring congestion from signals in the network: such as setting an indicator when buffers exceed a threshold [5] or using some combination of delay, estimated send time, and bottleneck link capacities to infer when applications are sending at too high of a rate [35–37]. Yet these schemes are reactive in nature, signalling back to clients to throttle their rates as queues start to build up. Recent efforts have accelerated this signalling by utilizing innovations in switch design [38], but still there is a delay between when congestion signals are sent and when they are reacted to. Aequitas takes a more proactive approach by encoding some application knowledge via network QoS. It can be run in conjunction with these schemes, allowing service owners to tune QoS weights to ensure critical traffic meets their SLOs in times of congestion.

Application-Network Integration: Coflow [21–23] was an early application-network integration idea that proposed informing the network of application semantics. However, this abstraction was tailored to the applications of the time and it cannot encode the end-to-end semantics of some of the more modern applications. Other solutions target an application-network API, providing a way for applications to inform the network of changes in sending rates or their SLOs [39–42].

These schemes target application integration across a general set of applications, but there has also been considerable work on optimizations for specific workloads [24–26, 43]. Many of these either require input from users in the form of workload patterns or use some form of offline profiling before sending flows and thus are better suited to long flows. Aequitas aligns to the priorities used by a small number of storage systems but does not require any input from the various services that use these systems[6]. Moreover, there are no additional profiling phases or other delays introduced by Aequitas, making it suitable for flows of all sizes.

Scheduling and Prioritization: There are many works that incorporate scheduling or prioritization schemes, both in the network domain and the wider systems community. SRPT is a scheduling algorithm that schedules shortest jobs first. It has been proven to minimize mean response times [44–46] and often performs very well in practice [6, 7, 47, 48]. Yet, it is often difficult to know flow or RPC sizes a priori. In response, a collection of SMART schemes were developed that approximate SRPT scheduling [49, 50] and others have aimed to separate large and small flows [51–55]. Even some of these techniques assume job/flow sizes are known in advance, or use techniques to estimate them. However, flow size alone is often not a sufficient predictor of a flow's network requirements [14]. Instead, Aequitas uses only the RPC priority to set QoS, without any knowledge of flow size.

8 Conclusion

In this work we detailed our experience of deploying Aequitas in Google's planet scale production network. This implementation of Aequitas is aimed at storage and enforces static mappings from RPC priorities to QoS classes for these systems. We present production examples showcasing that prioritizing through QoS assignment is not a zero-sum game and the ecosystem as a whole can benefit from the right policy. These findings are backed by theoretical analysis. While Aequitas requires no input from users, deploying it at the scale of our systems provides many challenges. We outline these concerns and our use of an incremental deployment approach. While this greatly extended the rollout timeline, the scheme was enabled in a safe manner that gave users confidence in its benefits. We hope that our experience can serve as a framework for QoS-based deployments in other networks, especially when an all-or-nothing approach is not possible.

Acknowledgements

The authors would like to thank Hassan Wassel, David Bacon, David Culler and the anonymous NSDI reviewers for their invaluable feedback and our shepherd Ankit Bhardwaj for guiding us through the revision process.

[6] While rolling out Aequitas was a long process at Google, this was because we wanted to be transparent to users and ensure that Aequitas performed as intended. But for an individual application, a user can simply opt-in and does not need to provide any information to the Aequitas system

References

[1] Rene L Cruz. A calculus for network delay. i. network elements in isolation. *IEEE Transactions on information theory*, 37(1):114–131, 1991.

[2] Rene L Cruz. Service burstiness and dynamic burstiness measures: A framework. *Journal of High Speed Networks*, 1(2):105–127, 1992.

[3] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[4] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 139–150, 2012.

[5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[7] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[8] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[9] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2017.

[10] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, 1(4):397–413, 1993.

[11] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. Choke-a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 2, pages 942–951. IEEE, 2000.

[12] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review*, 33(2):23–39, 2003.

[13] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. Fastlane: making short flows shorter with agile drop notification. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 84–96, 2015.

[14] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: Admission control for performance-critical rpcs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 1–18, 2022.

[15] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[16] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202–208, 2009.

[17] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 225–238, 2017.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[19] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[20] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[21] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[22] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 443–454, 2014.

[23] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[24] MR Siavash Katebzadeh, Paolo Costa, and Boris Grot. Saba: Rethinking datacenter network allocation from application's perspective. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 623–638, 2023.

[25] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. {TopoOpt}: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, 2023.

[26] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 428–440, 2022.

[27] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 1–14, 2015.

[28] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[29] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, et al. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment*, 13(12):3181–3194, 2020.

[30] Mubashir Adnan Qureshi, Junhua Yan, Yuchung Cheng, Soheil Hassas Yeganeh, Yousuk Seung, Neal Cardwell, Willem De Bruijn, Van Jacobson, Jasleen Kaur, David Wetherall, et al. Fathom: Understanding datacenter application network performance. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 394–405, 2023.

[31] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[32] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[33] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329, 1988.

[34] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[35] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Communications of the ACM*, 60(2):58–66, 2017.

[36] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020.

[37] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[38] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt:{Sub-RTT} congestion control for {Ultra-Low} latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 219–236, 2023.

[39] Amir Baniamerian, Ali Munir, Ashkan Sobhani, Mahmoud Mohamed Bahnasy, Ron Thomas, Si Yan, Xingjun Chu, and Yashar Ganjali. Nce: an ecn dual mechanism to mitigate micro-bursts. In *Proceedings of the ACM SIGCOMM Workshop on Network-Application Integration*, pages 7–12, 2022.

[40] Seyed Hossein Mortazavi, Hossein Shafieirad, Mahmoud Bahnasy, Ali Munir, Yuanhui Cheng, Anudeep Das, and Yashar Ganjali. Accord: application-driven networking in the datacenter. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 1–10, 2021.

[41] Seyed Hossein Mortazavi, Ali Munir, Mahmoud Mohamed Bahnasy, Haiwei Dong, Shimiao Wang, and Yashar Ganjali. Earlybird: automating application signalling for network application integration in datacenters. In *Proceedings of the ACM SIGCOMM Workshop on Network-Application Integration*, pages 40–45, 2022.

[42] Ali Munir, Seyed Hossein Mortazavi, Mahmoud Mohamed Bahnasy, Amir Baniamerian, Shimiao Wang, Shichao Guan, and Yashar Ganjali. Smarttags: bridging applications and network for proactive performance management. In *Proceedings of the ACM SIGCOMM Workshop on Network-Application Integration*, pages 46–52, 2022.

[43] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[44] Linus Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[45] Donald R Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1978.

[46] Nikhil Bansal and Mor Harchol-Balter. Analysis of srpt scheduling: Investigating unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International conference on Measurement and modeling of computer systems*, pages 279–290, 2001.

[47] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems (TOCS)*, 21(2):207–233, 2003.

[48] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. Srpt for multiserver systems. *ACM SIGMETRICS Performance Evaluation Review*, 46(2):9–11, 2019.

[49] Adam Wierman, Mor Harchol-Balter, and Takayuki Osogami. Nearly insensitive bounds on smart scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):205–216, 2005.

[50] Chang-Woo Yang, Adam Wierman, Sanjay Shakkottai, and Mor Harchol-Balter. Tail asymptotics for policies favoring short jobs in a many-flows regime. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 97–108, 2006.

[51] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.

[52] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. {Information-Agnostic} flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, 2015.

[53] Abdullah Bin Faisal, Hafiz Mohsin Bashir, Ihsan Ayyub Qazi, Zartash Uzmi, and Fahad R Dogar. Workload adaptive flow scheduling. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 241–253, 2018.

[54] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015.

[55] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

Appendices

A QoS Arbitrage

Here we aim to answer the question posed in §2.3: *what are the conditions under which a user should prefer one QoS class over another?* To answer this question we consider a Weighted Fair Queue (WFQ) with k QoS classes QoS_0, ..., QoS_{k-1} having weights w_0, ..., w_{k-1}, respectively. That is, the long run average service rate for QoS_i is $\frac{w_i}{w_0+\ldots+w_{k-1}}$. Note that a WFQ is work-conserving, thus we can assume the queues corresponding to QoS_0, ..., QoS_{k-1} are non-empty.

Without loss of generality, we will relabel the queues so that we consider the queues corresponding to QoS_0 and QoS_1 with $w_0 > w_1$ (note here we assume $>$ and not $\geq$ because if $w_0 = w_1$ then there is no differentiation between the QoS classes). So in this setting QoS_1 represents a lower priority than QoS_0. Now suppose that the current loads on QoS_0 and QoS_1 are λ_0 and λ_1, respectively. Race-to-the-top arises from the belief that for a particular client c, c's traffic on QoS_0 will experience lower delay as compared to c's traffic on QoS_1. This assumption is due to the fact that $w_0 > w_1$, however, we will show that this is not true in all cases, and depends on the load on both QoS classes.

If client c's traffic on QoS_0 experiences lower latency compared to c's traffic on QoS_1 then this is the desired outcome and we denote this by $\text{QoS}_0 \succ \text{QoS}_1$. If on the other hand, c's traffic experiences worse performance on QoS_0, we refer to this as priority inversion, which is denoted by $\text{QoS}_0 \prec \text{QoS}_1$. Let client c's load on QoS_0 and QoS_1 be β_0^c and β_1^c, respectively. Note that $\beta_0^c \leq \lambda_0$ and $\beta_1^c \leq \lambda_1$ as the client's load on a QoS class is lower than the total load on a QoS class.

Define an epoch to be a unit of time in which each QoS class is served in proportion to its weight. If we assume that a client's traffic is independent from the rest of the traffic in the queue[7] then we can reasonably assume that client c's traffic is evenly distributed in the queue[8]. That is, in a period in which QoS_i is served, we can expect $\frac{\beta_i^c}{\lambda_i}$ proportion of that traffic belongs to client c. Then of all the traffic served from QoS_0 and QoS_1 during the epoch the proportion that belongs to client c (regardless of how much traffic is in other QoS bands) is given by

$$w_0 \frac{\beta_0^c}{\lambda_0} + w_1 \frac{\beta_1^c}{\lambda_1} \tag{2}$$

Now suppose that we shift some small amount ε of client c's traffic from QoS_0 to QoS_1. Then the amount of traffic on QoS_0 and QoS_1 now becomes $\lambda_0 - \varepsilon$ and $\lambda_1 + \varepsilon$, respectively. Whereas, the amount of client c's traffic on these two queues is given by $\beta_0^c - \varepsilon$ and $\beta_1^c + \varepsilon$, respectively. Under this new distribution, equation (2) becomes

$$w_0 \frac{\beta_0^c - \varepsilon}{\lambda_0 - \varepsilon} + w_1 \frac{\beta_1^c + \varepsilon}{\lambda_1 + \varepsilon} \tag{3}$$

Note that client c's traffic distribution in the other $k-2$ queues did not change. So if the quantity in equation (3) is larger than the quantity in equation (2), then of all traffic served in the epoch, a larger proportion of this traffic belongs to client c after the redistribution, and thus client c receives better service. That is, more of client c's traffic is served in a given epoch if

$$\begin{aligned} & w_0 \frac{\beta_0^c - \varepsilon}{\lambda_0 - \varepsilon} + w_1 \frac{\beta_1^c + \varepsilon}{\lambda_1 + \varepsilon} > w_0 \frac{\beta_0^c}{\lambda_0} + w_1 \frac{\beta_1^c}{\lambda_1} \\ \implies & w_1 \left[\frac{\beta_1^c + \varepsilon}{\lambda_1 + \varepsilon} - \frac{\beta_1^c}{\lambda_1} \right] > w_0 \left[\frac{\beta_0^c}{\lambda_0} - \frac{\beta_0^c - \varepsilon}{\lambda_0 - \varepsilon} \right] \\ \implies & w_1 \left[\frac{\lambda_1 - \beta_1^c}{\lambda_1 (\lambda_1 + \varepsilon)} \right] > w_0 \left[\frac{\lambda_0 - \beta_0^c}{\lambda_0 (\lambda_0 + \varepsilon)} \right] \end{aligned}$$

Note that the last equation precisely matches (2.3) in §2.3. Moreover, $\lambda_0 - \beta_0$ is the total amount of traffic on QoS_0 across all customers excluding client c, and similarly, $\lambda_1 - \beta_1$ is the total amount of traffic on QoS_1 across all customers excluding client c. Also notice that there is an admissible region for equation (2.3): Take $w_0 = 0.6$, $\lambda_0 = 70$Gbps, $\beta_0 = 20$Gbps, $w_1 = 0.1$, $\lambda_1 = 5$Gbps, $\beta_1 = 0.1$Gbps, $\varepsilon = 1$Gbps. Then the left hand side of equation (2.3) simplifies to 0.0163 while the left hand side is 0.0060. Thus, even though $w_0 > w_1$, it is advantageous for the client to transfer a small amount of traffic from QoS_0 to QoS_1, due to the low load on QoS_1 relative to its weight.

B Unhealthy QoS

In this section, we examine the state of QoS in Google storage systems. We are specifically interested in clusters with an unhealthy QoS mix, as defined in §2.3. Aequitas is targeted towards the queues for QoS_l, QoS_m, and QoS_h so that is what we focus on here as well.

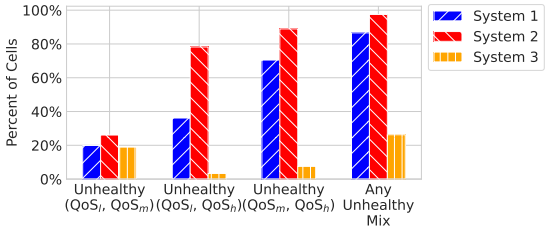
Figure 15 shows the proportion of clusters that have an unhealthy QoS mix across 3 UL storage systems, in the absence of Aequitas. We show the results for the three pairs of QoS classes[9] as well as a column that shows the percentage of cells that have an unhealthy mix between any of the three pairs. The figure contains results for both request and response traffic. For example, in the absence of Aequitas, System 2 request traffic has an unhealthy (QoS_h, QoS_m) mix in 89% of the clusters in which it is located, meaning that System 2's ratio of QoS_h to QoS_m request traffic in 89% of clusters is higher than $w_h : w_m$. Similarly, System 2 request traffic has an unhealthy mix for at least one pair of QoS classes in 97% of the clusters in which it is located.

As can be seen from the figure, a significant proportion of clusters have an unhealthy mix for at least one pair of QoS classes, especially for systems 1 and 2. An unhealthy mix between QoS_h and QoS_m is the biggest culprit, but there are instances of unhealthy QoS across all three pairs. This is in line with the notion of race-to-the-top discussed in [14], which suggests that users gravitate towards higher QoS over time.

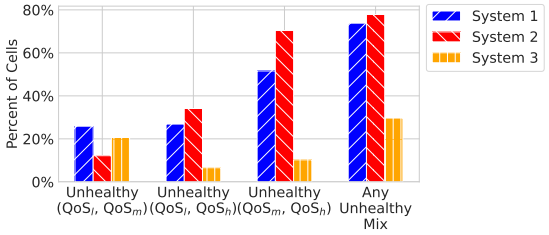
[7] While this assumption does not apply in every scenario, due to Google's scale we can reasonably assume one user's traffic is independent from other traffic in the datacenter

[8] Note that this does not assume that client c's traffic is necessarily smooth over time, it can occur in bursts. This just assumes a relatively small period of time in which client c's traffic is roughly evenly distributed among the rest of the traffic destined to that queue

[9] The QoS classes are naturally ordered by their weights; this also implies an ordering on the pairs



(a) Request Traffic



(b) Response Traffic

Figure 15: Percent of clusters with priority inversion across three UL systems based on (a) request traffic and (b) response traffic

C Client-side vs. Server-side Implementation

As detailed in section 2.2, Aequitas is implemented on the server-side for LL storage, whereas we used a client-side approach for UL storage implementations. The choice of where to implement depends on many factors that were discussed previously. Here we focus on how this affects the rollout process and analysis of the results.

A server-side implementation offers several advantages. First, it facilitates faster rollout and adoption as all traffic using a particular server is affected by Aequitas as soon as the implementation is deployed to the server. Thus the rollout is able to progress at the speed at which the servers are updated with the latest changes. This is generally much faster than client-side updates as the servers are controlled by a small number of teams that can coordinate simultaneous upgrades. Second, as seen in [14] and shown in section 2.3, the performance of traffic using a certain QoS class depends on both the weight of that QoS class and the load of all traffic passing through the same switch queues at that particular QoS. Server-side implementation implies that all traffic using the same server is affected simultaneously. Thus we see large-scale changes with each server updated and the behaviour of the system converges to the Aequitas QoS distribution quickly, making it possible to assess user-level and cluster-level performance changes more easily.

In contrast, for client-side implementations, the rollout process is much slower. It is far more difficult to coordinate simultaneous upgrades across users due to the various teams involved and the different jobs that these users are running. For example, some clients have long running jobs that take on the order of months to complete and thus will not pick up changes until these jobs finish. At the same time, there are other clients that can pick up changes almost immediately. This results in many incremental QoS changes over a long time period, making it difficult to see the impact at a datacenter level. Moreover, for clients who pick up the changes early, their QoS changes as a result of Aequitas can be a small portion of the overall QoS in the datacenter. As a result, it can be difficult to assess the long-term impact for an individual client as most of the datacenter remains unaffected by Aequitas and the client's performance is likely to change as the datacenter QoS distribution changes due to increased adoption of Aequitas. As discussed in §2.2 and seen in §4, this required extra considerations to facilitate a smooth rollout while carefully teasing out the effect of Aequitas.

D Aequitas for SSD LL Systems

In this section, we expand on the results for deploying Aequitas on LL systems previously presented in [14]. The production deployment results explained in [14] were presented for HDD storage traffic between LL and UL systems, wheres since then, we have expanded Aequitas to cover SSD traffic to/from LL systems as well. Here, we present the results related to SSD LL systems performance.

At a high level, $\sim$ 30% of all LL-bound SSD related RPCs were misaligned prior to Aequitas and this number is now $\sim$ 0%[10]. Prior to Aequitas deployment, 34%, 24% and 90% of PC, NC and BE.

The reduction in aggregate misalignment for RPCs, among clusters, varies. There are cluster with close to 100% misalignment, whereas the smallest percentage of misalignment in a single cell, observed during the sampling, is around 10%.

After expanding Aequitas to cover SSD LL system traffic, one of the users with most misaligned RPCs sees significant changes to their PC RPC latency. The average RPC latency is improved by 6.4% while the 99th percentile of RPC latency is improved by 11.2% for this user. Furthermore, we measured improvement of 2.6% and 5.4% for average and 99th percentile of NC RPC latency respectively. This is inline with our goal of having positive overall performance incentive for applications as the corresponding user experiences improvement in both of their latency sensitive RPC priorities.

The aggregated fleet-wide results, also show improvements after full deployment of Aequitas. The average PC RPC latency has reduced by 1.8% and the 99th percentile has reduced by 4.8%. These results, along results in [14] and results presented in rest of this paper for UL storage systems, show the robustness of our learning. That in fact, the correct QoS enforcement holds true consistently across different layers of storage systems stack.

[10]The number of misaligned LL-bound RPCs is $\sim$ 0% as Google has an intricate web of applications and a vast array of jobs running from users around the globe. As a result, while we have completed the Aequitas deployment for LL storage, there can be transient traffic that operates outside of the Aequitas framework

DISC: Backpressure Mitigation in Multi-tier Applications with Distributed Shared Connection

Brice Ekane*, Djob Mvondo*, Renaud Lachaize†,
Yérom-David Bromberg*, Alain Tchana†, Daniel Hagimont‡
*Univ. Rennes, Inria, CNRS, IRISA, France
†Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
‡IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3 Toulouse, France

Abstract

Most data-center applications are based on a multi-tier architecture, involving either coarse-grained software components (e.g., traditional 3-tier web applications) or fine-grained ones (e.g., microservices). Such applications are prone to the backpressure problem, which introduces a strong performance coupling between tiers, thus degrading scalability and resource consumption. This problem is due to the fact that, on the response path towards the initial client, a significant fraction of the payloads in the messages exchanged between tiers correspond to "final" data that are simply relayed (i.e., without further modifications) from a backend tier such as a database. This traffic results in additional pressure on the intermediate and frontend tiers.

To address this problem, we introduce DISC, a system allowing several tiers within a multi-tier chain to jointly act as endpoints of the same TCP connection. This enables the selective bypass of one or several tiers on the response path. Unlike existing solutions, DISC is (1) flexible — it accommodates arbitrary multi-tier topologies and heterogeneous application-level protocols, (2) fine-grained — it allows multiple tiers to be involved in the generation and emission of a given response message (e.g., to decouple the network path of the response headers and footers from the path of the response body), (3) and non-intrusive — it requires only minor and localized/modular modifications to the code base of legacy applications and is transparent for external clients. Evaluation results with several micro- and macro-benchmarks show that DISC can reduce the cumulative CPU load on servers by up to 41.5%, decrease the average and tail latencies respectively by up to 74.1% and 5.71×, and also improve the request rate by up to 45%.

1 Introduction

For reasons of modularity and, above all, scalability, Internet services are built in the form of an assembly of components in an architecture formerly referred to as *multi-tier* or more recently as *microservices*. In such an architecture, components are loosely coupled and communicate through messages using protocols such as HTTP, IMAP, gRPC, etc. In the remainder of this paper, we use the terms *"tier"* and *"service"* interchangeably, as our contribution is agnostic to the number and granularity of these components.

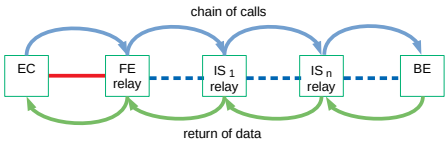


Figure 1: A schematic view of a chain of tiers. Tier BE returns data which is relayed by tiers IS_i, up to FE toward client EC.

Tier *independence* is a fundamental property that makes the popularity of multi-tier architectures. However, this independence is often compromised regarding performance, making it challenging to scale each tier independently. Our contribution aims at reducing redundant data transfers, which are a major impediment to *performance independence* of tiers. As illustrated in Fig. 1, there are redundant data transfers among different tiers involved; a request initiated by an External Client (EC) is received by a first-tier, known as the FrontEnd (FE). FE may require a call to another tier, thus forming a chain of invocations traversing Intermediate Servers (IS) up to a final tier named the BackEnd (BE). Such a chain can be noted as the following: $EC \leftrightarrow FE \leftrightarrow IS_1 \leftrightarrow \ldots \leftrightarrow IS_n \leftrightarrow BE$. In this chain, when BE produces a reply that must be returned to EC without further (read/write) processing by the IS and FE tiers, it still must go up the chain in the reverse order of request calls, generating redundant communications. We refer to such replies as *final data*. An example of final data is the mail (text and attached files) in a (multi-tier) webmail application such as Zimbra [7]. Even if IS and FE tiers only relay the final data, they undergo a load proportional to the activity of BE. Consequently, the performance of IS and FE is strongly coupled with the activity of BE. This is known as the *backpressure* problem [15]. Performance dependency among tiers makes it challenging to manage the scalability of the global application

because of the risk of the FE being the bottleneck (see §2).

For more than 20 years, numerous research works [2, 11, 14, 17, 18, 20, 21, 24, 27] have regularly proposed techniques to overcome this problem, yet with limitations. A first limitation is that these works mainly target 2-tier architectures composed of a load balancer distributing the load between a set of backends whereas applications may include much longer chains. For instance, JEE clustered architectures may include many tiers such as: an HTTP load balancer, a web server, a servlet container, an EJB server, a database load balancer, and a database. We also observed such long chains in the Zimbra clustered email server. The same observation applies to modern applications deployed in data centers, such as LAMP/MEAN stacks and microservices, which rely increasingly on long chains of existing tiers for scalability purposes.

A second limitation of all existing approaches is that they rely systematically on the full transfer of the connection from the EC to the BE, which is analogous to a migration. These solutions are radical as they cannot be generalized; they bring two key constraints. (1) Transferring the connection from a tier T_1 to a tier T_2 requires that the two tiers implement both the same communication protocol (e.g., HTTP) and API. This is why previous works have been limited to a 2-tier architecture. As is generally the case for the scalability of storage services [17], these two tiers comprise only a load balancer and a backend stage. Conversely, a complex multi-tier architecture, such as Zimbra, must generally use different protocols and APIs among tiers and is therefore incompatible with the full transfer principle. (2) Transferring the connection from a tier T_1 to a tier T_2 requires that T_1 can work correctly and efficiently without receiving (after connection transfer) requests/replies, which is only possible for simple and coarse-grained load balancing policies. Modern load balancers are not compatible with such connection transfers as they leverage the metadata included in the headers/footers of requests and replies. For example, to select the target backend, HAProxy [4] uses attributes of the request header and NGINX [5] tracks the return time of the reply header.

In a nutshell, the ideal solution to the backpressure problem is to remove redundant communications of final data. To this end, the following conditions must be fulfilled: (1) allow final data to bypass tiers when required, (2) support heterogeneous protocols (e.g., combining HTTP and IMAP) and APIs in the tier chain (which is not compatible with the full transfer principle), (3) guarantee the correct operation of multi-tier architectures (e.g., prevent inconsistencies when tiers are bypassed), (4) do not require the modification of clients, (5) support secure communication channels (e.g., TLS).

This article presents DISC, the first approach (to the best of our knowledge) that meets all of the above 5 requirements at once. DISC bypasses IS and FE adequately by allowing a BE to directly send the final data to an EC. We design DISC in such a way that the bypass can be applied to any sub-part of a chain of arbitrary depth. To operate seamlessly, DISC disaggregates the reply, which is decomposed into two parts: (i) metadata (i.e., headers/footers), and (ii) payload. DISC lets metadata (generally limited in size) follow the reply path along the chain whereas payloads (generally substantial in size) are subject to bypass IS and FE. DISC can be implemented atop any transport protocol; we chose TCP for our prototype.

Designing DISC has raised 5 major challenges. (C_1) ***TCP sequence number management.*** Since a tier can return payloads directly on the TCP connection established between two other tiers, it is necessary to coordinate the emission of the sequence numbers so that it is transparent to those tiers. DISC allocates and translates the sequence numbers to take into account distributed emitters. (C_2) ***Management of TCP acknowledgments.*** Since the data transmitted comes from different tiers, acknowledgments received from the destination must be propagated to emitters so that they can move forward in the payload transmission process. DISC routes acknowledgments towards the concerned emitter. (C_3) ***Integration into protocol stacks.*** To avoid modifications of the Linux kernel protocol stacks, DISC extensively relies on BPF. (C_4) ***TLS support.*** Many communications over the Internet are secured using TLS. Therefore, all tiers involved in a chain must be able to return encrypted payloads in the same TLS session. To this end, DISC uses a serialization mechanism to ship TLS sessions among emitters. (C_5) ***Modification of applications.*** To minimize the impact of DISC on the code base of the different tiers, we present a pattern allowing to systematically modify applications.

We evaluated DISC using different representative multi-tier and microservice-based applications, in particular: SpecWeb [13], SpecMail [12], Train Ticket [22] and DeathStarBench Social Media [15]. The quantitative results show that DISC can significantly reduce the propagation of pressure in a chain of tiers, thus reducing CPU load on FE and IS but adding CPU load on the BE. Globally, with DISC, the cumulative load on servers is reduced by up to 41.5% and the supported request rate is improved by up to 45%. DISC has no impact on mean latency and can improve tail latency significantly. For instance, on a micro-benchmark with ten IS, DISC reduces tail latencies down to 5.71×. The reduction of the load on the bypassed tiers makes it possible to ensure greater independence of the tiers from the point of view of performance and scalability. With DISC, saturation occurs first at BE up to FE (thus promoting service scalability), whereas without DISC, it is the inverse; saturation occurs first at FE.

The remainder of the article is structured as follows. Section 2 details the motivations for DISC and its position w.r.t. previous works. Section 3 present the main principles of our contribution. Section 4 reports the results of evaluations. We conclude in Section 5. Additional details about the implementation and the evaluation are provided in appendices.

2 Motivation and previous works

The architectural pattern of two common Internet applications, e-commerce web applications and email services (respectively illustrated by the SpecWeb [13] and SpecMail [12] benchmarks), are described in §2.1, and, in §2.2, we evaluate the impact of their inherent redundant communications on server load and application scalability. Finally, we position DISC with respect to state of the art solutions in §2.3.

2.1 Motivating use cases

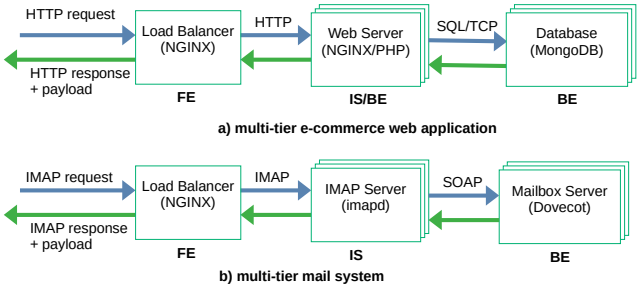


Figure 2: Two typical N-tier applications: e-commerce (SPECweb) and mailing system (SPECmail).

Coarse-grained multi-tier applications remain very popular in many production contexts. For example, according to a study [26], the WordPress content management system [10] is used by 43% of the websites on the Internet. To explain the motivation behind DISC, we use two multi-tier benchmarks due to their simplicity. These two applications involve different protocols such as HTTP, IMAP or SOAP. Notice that, we also demonstrate later how DISC can be beneficial for microservices-based applications (see §4.5 and Appendix B).

E-commerce application. Fig. 2-a depicts a common setup of a 3-tier architecture for an e-commerce application. In this architecture, the FE is a load balancer (NGINX) distributing requests received from EC to a set of web servers. These web servers (NGINX/PHP) can serve as BE, and return static pages (e.g., images). They can also act as IS when executing PHP code that requires database access; in this case, a request is sent to the database (MongoDB), which can in turn serve as BE, and can return dynamic content as final data (e.g., a JSON collection). In our experiments, we used the e-commerce application (and load injector) from SpecWeb [13].

Email serving. Fig. 2-b illustrates a possible deployment setup of a 3-tier architecture for the Zimbra email system. As in the previous application, the FE is also a load balancer (NGINX). It distributes requests received from EC to a set of IMAP servers (imapd). These IMAP servers act as IS, and send requests to Mailbox servers where client emails and folders are managed. The Mailbox servers are implemented using Dovecot [3], and serve as BE. In our experiments, we used SpecMail [12], which provides a load injector.

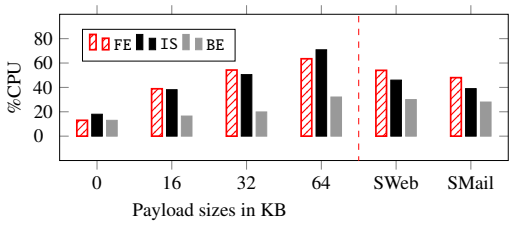


Figure 3: CPU load across each tier (vanilla setting).

2.2 Problem assessment

Final data returned by the BE must go up the chain in the reverse order from tier to tier until it reaches the EC.

Backpressure. To highlight the problem, we study the communications in a traditional multi-tier design (hereafter named *"vanilla"*). We assess the impact, in terms of CPU consumption, of relaying final data on each tier of a 3-tier architecture. We evaluate three applications: a micro-benchmark made of a chain of 3 NGINX servers, SpecWeb, and SpecMail. For the micro-benchmark, the EC submits a constant load to the first tier of the chain (FE), while the payload size of the final data returned by the BE varies from 0KB to 64KB (all the returned data is final). Conversely, for SpecWeb and SpecMail, the amount of final payloads and their sizes vary according to the benchmark instantiated. The detailed setups for these evaluations are provided in §4. At the left of the dashed line on Fig. 3, we show the CPU load on each tier of the chain (i.e., FE, IS, and BE) for the micro-benchmark. We observe a higher load on the FE and IS tiers compared to the BE tier. Furthermore, as the size of the payload returned by the BE increases, the CPU load logically increases on the BE, but it rises more significantly on the FE and IS tiers. This clearly illustrates the issue of backpressure, whereby the increased load on the BE propagates to the FE and IS tiers. Additionally, a payload size of 0 provides an estimate of the CPU load on FE and IS tiers if the payloads were directly returned to EC by the BE, thus bypassing FE and IS tiers. At the right of the dashed line on Fig. 3, we show the results for SpecWeb and SpecMail. We observe similar trends compared to the micro-benchmark.

Scalability. This backpressure problem has a significant impact on the application's scalability. To demonstrate this, we consider the same micro benchmark based on an FE-BE architecture. A load injector (EC) generates requests at a growing rate (+2K req/s every 3 seconds). We start each tier with 2 CPU cores allocated and we double the number of cores of a

tier (FE or BE) when it becomes saturated (vertical scalability). The results are shown on Fig. 4-top (Fig. 4-bottom displays the results achieved with DISC, discussed later in §4.2).

As shown at the left of the solid line on Fig. 4-top, the FE (grey curve) is the first tier to saturate due to the load on the BE that has propagated to its preceding tier. This saturation impacts the average latency measured from the client (dashed curve). At the right side of the solid line on Fig. 4-top, we added 2 cores to the FE, thus allowing the application to scale.

The goal of DISC is to ensure that an increase of the load on the BE (which returns final data to EC) does not propagate to the preceding tiers, and therefore does not lead to a saturation of the FE first, requiring the allocation of additional resources on these tiers. Ideally, an increase of the load on the BE should only lead to an increase of allocated resources on the BE.

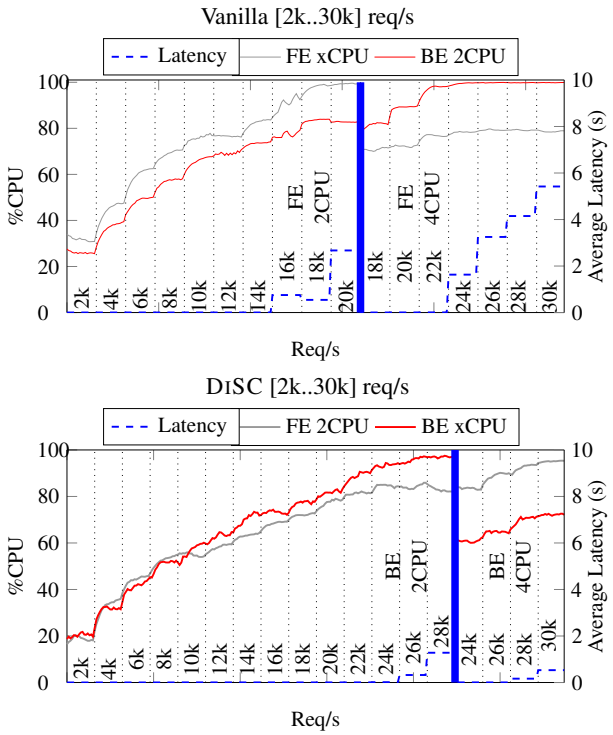


Figure 4: CPU load on the FE and BE and overall latency when increasing the request rate. Vanilla (top) vs DISC (bottom).

2.3 Shortcomings of existing solutions

Over the last decades, several works have partially addressed the backpressure problem [2,17,20,21,27]. They are designed around the *connection handoff* concept [11,18], i.e., the migration of connections between servers. However, existing handoff-based solutions have three major limitations: (i) short depth path, (ii) API lock-in, and (iii) side effects of shortcuts. We also discuss the limitations of QUIC [23], which could falsely appear to be an alternative to DISC. Given our focus on compatibility with existing datacenter applications and infrastructures, we do not discuss data transfer optimizations needing invasive software or hardware modifications [28,29].

Short depth path. The two most recent contributions, CRAB [20] and Prism [17] only support a 2-tier architecture; the FE acts as a load balancer to distribute requests to several BE. As a result, they can only be used to bypass a single server. Moreover, most common n-tier architectures, like those described in §2.1, are based on 3 tiers or more. DISC is independent of the number of tiers, and allows bypassing more than one tier. Besides, unlike Prism, DISC does not require specific hardware support (smart switches).

Application protocol/interface lock-in. Existing solutions rely on connection migration and therefore they require all tiers (FE and BE) to implement the same protocol (e.g., HTTP) and API, which is a strong constraint. As mentioned in §2.1, a multi-tier architecture may involve a combination of different protocols (e.g., HTTP and IMAP). DISC is independent of the application-level protocols and the APIs being used.

Side effects of shortcuts. Connection migration fully bypasses the FE, for both onward and backward traffic. A full bypass may have a significant impact on a FE that acts as a load balancer, which can lead to malfunctioning. This comes from two main reasons: (1) metadata attached to the final data (e.g., header, or footer) may contain information provided by the BE required by the FE to adjust its behavior a *posteriori* (e.g., for feedback-driven load balancing leveraging the request backlog size of each BE). (2) More importantly, receiving responses is an invariant of the execution logic of a multi-tier application. Completely bypassing a tier requires in-depth modifications of its code, which breaks its genericity.

QUIC limitations. QUIC [23] is a transport protocol recently standardized by IETF, implemented in user level and based on UDP. QUIC provides multiple streams within a connection (similarly to HTTP/2, although QUIC does not suffer from head-of-line blocking). It relies on a Connection ID to identify connections instead of IP/port, thus easing connection migration across IP address changes. This way, it provides a mechanism for client-side connection migration, e.g., when a client changes its network connectivity (thus its IP and port) on network handover. Thus, QUIC provides a mechanism equivalent to TCP hand-off. As stated above, hand-off solutions require all servers to implement the same protocol and API (e.g., HTTP) on top of the transport protocol (TCP or QUIC). Besides, some works [25] investigated server-side connection migration in QUIC, especially for migrating to a closer server in an edge computing context. However, such an extension requires modifications both on the server and client sides of QUIC (while DISC is transparent for the client).

QDSR [30] brings the benefits of DSR to QUIC-based L7 load balancers and offloads the responses for distinct streams to different backend servers. This approach also suffers from the limitations discussed in the three previous paragraphs.

3 Our approach: DISC

To operate shortcuts, DISC relies on a new protocol, DISC-PROT. This protocol adds header information to the request and response messages on top of the underlying L4 protocol (TCP). It enables any node in the chain to indicate in a request, which transits through the forward path, that it can be later bypassed, i.e., that it can be skipped in the backward path for the transmission of the final response's payload.

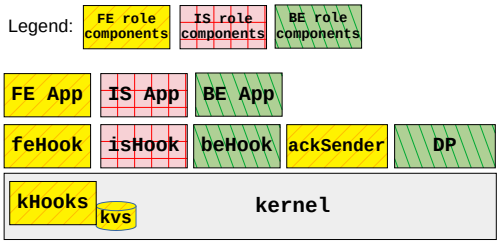


Figure 5: Software architecture of DISC on a server, with components used according to the tier role.

3.1 DISC architecture

A cloud service, invoked by an external client, is made of a set of tiers (hosted by servers in datacenters) structured as a directed acyclic graph that is traversed through request-response invocations following a chain of calls.

Roles. In a chain, depending on whether it is bypassed or not, a tier T_i can play one or the other of the following roles: (i) BE, (ii) EC, (iii) IS, and (iv) FE. T_i acts as a BE (*i.e., it has a backend behavior*) if it sends the final response. T_i acts as an EC (*i.e., it has a client behavior*) if it emits the initial request, and hence is the target of the final response. Usually such a role is taken by the external client, but nothing prevents a tier from having this role if the shortcut occurs in the middle of the chain. T_i acts as an IS (*i.e., it has an intermediary tier/service behavior*) if it is bypassed in the chain, while still receiving the metadata of the final response (without its payload). Finally, T_i acts as FE (*i.e., it has a frontend behavior*) if it is directly connected to the one that acts as an EC. In this particular case, it will ask the first encountered previous BE to send the final response to EC.

Components. DISC is implemented as a kernel extension, and is bundled with several additional components in both the kernel and user space. These components enter in action according to the role played by tiers, i.e., FE, IS, BE (see Fig. 5). Note that to act as an EC, DISC is not required as it is transparent — external clients do not need DISC. However, DISC must be deployed on all servers of the datacenter to perform shortcuts. From a kernel-space perspective, DISC includes two kernel modules: (i) a key-value store, named *kvs*, to register information about in-progress requests, and (ii) a kernel hook called *kHook* for treating incoming/outgoing L4 packets (e.g., connection establishment). From a user-space perspective, DISC provides different hooks (i.e., *feHooks*, *isHook* and *beHook*) to act at the L7 layer (such hooks allow adapting applications) and two user-level modules, noted *ackSender* and *DP*. The former manages the acknowledgment of L4 packets, and the latter takes care of sending the payload to be sent directly from BE to EC (i.e., it performs the shortcut).

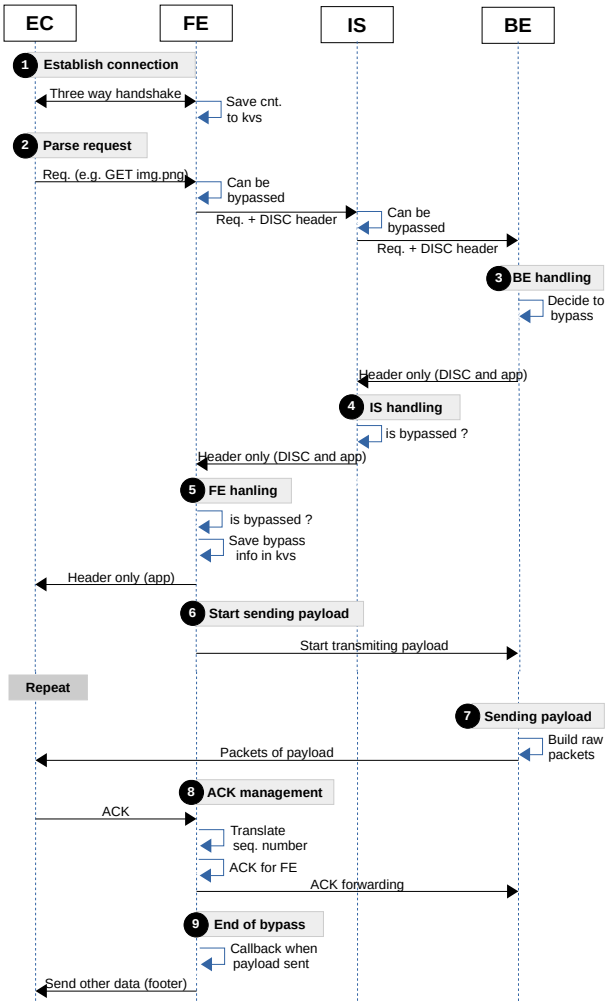


Figure 6: End-to-end coordination with DISC.

DISC shortcut mechanisms. A shortcut is the result of a coordination among FE, IS, BE and EC in 9 phases. Fig. 6 illustrates the different phases in terms of packet sequence

diagram for a scenario where BE sends the final response on the initial connection between FE and EC. The 9 phases are:

(1) Establish connection. When EC connects with FE, kHook intercepts three handshake messages and when the connection is established, it creates a key in kvs that identifies the TCP session. This key is associated with a record `<source IP, source port, ...>` that includes all information required for sending packets to EC and handling acks. Notice that all packets received on that connection are intercepted by kHook, especially acks (detailed in §3.3). As we show in §4, this interception does not impact request latency.

(2) Parse request. When EC sends a request, FE can indicate (*via* feHook, which adds an option in the DISC-PROT protocol) that it can be bypassed for the returned payload (see details in §3.2). Here, feHook is an adaptation in the L7 layer, where any policy can be applied for accepting or not the bypass, depending on the need for that tier to have or not access to the returned payload. Notice here that if FE is bypassed regarding the payload, it will anyway receive the application response header (e.g., IMAP header) and footer (e.g., IMAP footer) that can be used for internal processing. Then, FE forwards the request to IS, which behaves similarly (with its isHook) and forwards in turn the request to BE.

(3) BE handling. BE uses beHook to enable the current tier to decide whether the payload is eligible to bypass the previous tiers. If bypass is enabled, the payload is sent to the DP module, instead of the TCP connection with the preceding IS. At this point, DP generates an identifier (ids) that designates the payload in DP. Then, beHook builds a DISC-PROT header that includes the following options: a flag (*bypass*) saying that the payload is subject to bypass, the previous DP identifier (ids), the size of the payload and the IP address of the BE (allowing the FE to later request from the DP the emission of the payload, thus bypassing other tiers, FE and IS). Then, the BE application-level protocol builds its header, but with a payload size set to zero. Both headers (DISC-PROT and application-level) are included in the response message.

(4) IS handling. On IS, the response message is received and isHook checks the bypass option to skip the reading of the payload if bypassed. IS then handles the response normally and sends the response (including the DISC-PROT header) to its preceding FE, except it omits the payload if bypassed.

(5) FE handling. Like on IS, feHook checks the bypass option to read or not the payload. If the FE is bypassed, then feHook extracts all information about the bypass from the (DISC-PROT protocol) response header and stores them in the kvs entry associated with the associated initial connection. This information includes the IP of BE, the ids of the payload and its size, that will be used later to handle acks. Thus, FE can send the application-level response header back to EC.

(6) Sending the payload on FE. On FE, feHook then invokes (remotely) the DP on BE (it has its IP address) asking the emission of the payload identified with ids. This invocation provides to DP all information required to spoof FE (including TCP sequence numbers to be used), thus sending the payload on the initial connection.

(7) Sending the payload in DP. DP builds raw packets and sends them on the initial connection towards the EC. From that time, FE will receive acks for these packets (as DISC does not modify the Client). To allow the DP to continue emitting the payload (increase its emission window), received acks on FE (related to the payload) are forwarded to the DP.

(8) ACK management. All received acks on FE are intercepted by kHook (the kernel module), analyzed and handled. kHook is responsible for ensuring consistency of TCP sequence numbers. Indeed, the TCP stack on FE is not aware of packets sent by DP, thus kHook has to apply translations on sequence numbers. Also, acks received from EC may acknowledge packets emitted by the TCP stack on FE, or by DP. Acks concerning FE's stack have to be translated and acks concerning DP are forwarded to DP. Every ack concerning DP is stored in kvs and the ackSender process (started on FE) polls kvs to extract acks and forward them to DP. TCP sequence number management is detailed in §3.3.

(9) End of bypass. An application-level protocol may include a header and a footer. This is for instance the case with the IMAP protocol. Therefore FE, which sent on the initial connection its response header and delegated the emission of the payload to BE, may need to send a footer after the payload. In order to provide a means for that to the feHook, we provide an API for registering a callback function. This callback function is called when kHook observes from received acks that the full payload was received by the Client. feHook can then prepare the footer and register a callback function that will send the footer on the initial connection when the payload was totally sent. Notice here that we did not want to address this footer need through a synchronous invocation of the DP (which would have returned when the payload is sent), as it would have affected the threading model of FE. With this callback-based mechanism, the handling of footers is kept asynchronous (without blocking a thread). With DISC, we observe that the capacity to emit on the initial connection is temporarily delegated to BE and gets back to FE for sending the footer or sending the response header of the next request.

3.2 Instantiation

The previous section gave the general principle of DISC in a scenario with a unique shortcut, where a payload sent by BE bypasses IS and FE and is sent directly to EC. Now, we consider a more general example with 4 tiers chained as follow: $T_0 \leftrightarrow T_1 \leftrightarrow T_2 \leftrightarrow T_3$, as depicted in Fig. 7, and with two shortcuts at once; T_0 and T_2 can be bypassed but not T_1. The DISC-PROT protocol includes two options for managing shortcuts in its header: `NB_TIER` and `BYPASSED_TIERS`. The former is a counter initialized to 0 on T_0, incremented each time a request is forwarded to the next hop and decremented each time a response is returned to the preceding hop. It

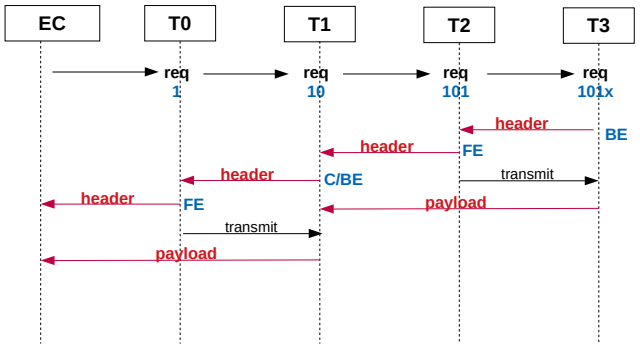


Figure 7: Concrete example of DISC with an external client invoking a chain of 4 tiers. Here, tiers T_0 and T_2 are bypassed.

permits a tier to know its place in the path and to adequately set the bit of the bitmap `BYPASSED_TIER`, indicating whether or not it wants to be bypassed.

Roles. T_3 acts as BE. Bypassing is possible if T_3 wants it for the current payload according to its policy (e.g., payload size greater than 20KB) and if the next tier (i.e., T_2) agrees to be bypassed. To perform a bypass, T_3 splits the final response into two responses and sends (i) one response without payload, i.e., a header and/or a footer, to T_2 to follow the backward path (lightweight forward), and (ii), one response that includes the payload to the local DP (shortcut). Otherwise, if there is no bypass, the final response is sent to T_2 (heavyweight forward).

Regarding T_1 or T_2, they can play different roles depending on whether they accept bypassing. If it does not agree to be bypassed (e.g., T_1), it receives a response that includes a payload. Hence, it plays simultaneously both EC and BE roles. Particularly, it acts as BE from the perspective of the next tier (e.g., T_0), and it acts as EC from the perspective of the previous tier (e.g., T_2). Otherwise, if it agrees to be bypassed (e.g., T_2), it receives a response without payload as it was checked by the tier that sent that response. Then, in this case, if the next tier (e.g., T_1) agrees to be bypassed, it plays the IS role (lightweight forward), otherwise it plays the FE role. In this case, it (e.g., T_2) remotely asks the DP of the first previous BE encountered (e.g., T_3) which held the payload to send it on the connection with the next tier (e.g., T_1). In this latter case, the next tier will receive a final response, acting as EC and BE as seen earlier.

Regarding T_0, if it receives a response with a payload (it plays an EC role), it simply sends it on the initial connection. Otherwise, if it receives a response without payload, it plays the FE role. Accordingly, it remotely asks the DP of the first previous BE encountered (e.g., T_1) that held the payload to send it on the connection of either the next tier (and so on) or the external client if the end of the chain is reached.

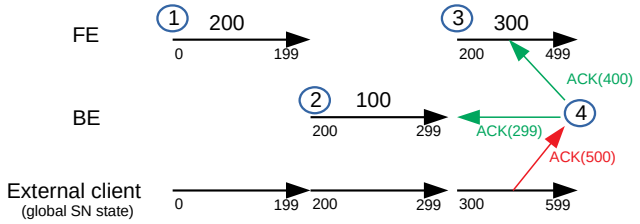


Figure 8: Coordination of TCP sequence numbers.

Bitmaps. In the scenario of Fig. 7, the final value of the bitmap is *101x* (the first bit is the leftmost one). T_3 decides to enable bypass (e.g., based on the payload size). Thus, it plays the BE role. As T_2 agrees to be bypassed (in Fig. 7, the 3rd bit of the bitmap is 1), T_3 sends a response without payload, and the payload is sent to DP. T_2 plays the FE role since T_1 does not agree to be bypassed (in Fig. 7, the 2nd bit is 0). Therefore, T_2 sends a response without payload, but it is sent from T_3 (its DP) to the T_2-T_1 connection. T_1 receives a response with payload, so it plays the EC and BE roles, and since T_0 agrees to be bypassed (in Fig. 7, the 1st bit is 1), T_1 sends a response without payload, and the payload is sent to its DP. T_0 receives a response without payload, and since its predecessor (EC) cannot be bypassed, it plays the FE role, thus sending a response without payload, but the payload is sent from T_1 (DP) on the T_0-EC connection (initial connection).

3.3 DISC atop TCP

First, we stress that DISC's design and implementation do not hinder the characteristics of applications built atop TCP. In particular, DISC is compatible with keep-alive connections, piggybacking (an ACK carries a new request), and request pipelining and does not negatively impact (or remove) TCP's congestion control mechanism. Below, we describe the main challenges introduced by DISC w.r.t. TCP. Implementation details for our DISC prototype are provided in Appendix A.

As we do not want to modify the TCP stack deployed on the servers (DISC pervasiveness), DISC needs to coordinate the TCP Sequence Number (SN) emitted by FE and BE. Indeed, EC is likely to receive TCP packets from multiple sources, such as from FE and BE, so both must send packets with consistent sequence numbers. This is illustrated in Fig. 8 where, for more clarity, we do not present sequence numbers in terms of bytes but in terms of packets. If FE sends 200 packets with SN from 0 to 199 (❶), and thereafter BE sends 100 packets (❷); then the SN for the latter should be from 200 to 299 (BE must be informed about the SN it should use). Now, if FE sends 300 additional packets (❸); from its point of view, the SN will be from 200 to 499 since it is not aware of the packet emissions from BE. Consequently, these SN must be translated to be consistent from the point of view of EC, so they will be from 300 to 599 (instead of 200 to 499).

The management of SN in DISC is implemented in kHook as follows. kHook maintains a global SN state in kvs, considering all emitted packets (from FE and BE). The invocation of BE (especially its DP) to trigger the emission of the final payload takes as an additional parameter the global SN state so that BE can send packets with consistent SN. The global SN state can be updated as we know the payload size (from the DISC-PROT response header). In Fig. 8 at emission (❷), the global SN state is 200, and the global SN state is updated to 299 after emission. Packets emitted by FE are modified by kHook to adapt their SN according to the global SN state. In Fig. 8 at emission (❸) from FE, SN from 200 to 499 are modified and become from 300 to 599. kHook maintains in kvs the emitted SN and the servers (FE or BE) which emitted them. Any packet acknowledgment (ack/sack) received by FE is intercepted by kHook. Its SN is analyzed to either send an ack/sack packet to FE 's TCP stack (and the SN is translated back) or either send an ack/sack packet to BE which emitted it (allowing to resend a packet, free its internal buffers, and adapt its emission window). In Fig. 8, an ack with SN 500 is received (❹) on the initial TCP connection. kHook, according to its internal state, modifies the SN to 400 (since 100 packets were sent by BE) so that the ack is consistent for FE. Moreover, it generates an ack packet for the preceding emission from BE with SN 299.

Last, SN for packets received from EC (e.g., requests) are all handled by the FE and will generate ack/sack towards EC. However, all payload packets emitted by the BE towards EC must include an ack with the last acknowledged received data SN. This SN (that we call the *client SN*) is passed to the DP of the BE when invoked for payload emission so that a proper ack can be added in the emitted packets.

Due to a lack of space, we only briefly discuss the behavior of DISC in the event of failures. Without DISC, in a multi-tier architecture, the failure of one tier generates delays, ultimately leading to a disruption in the connection with the external client. The application code in the tiers is responsible for implementing precise semantics in the event of such failures. Message losses are handled by TCP, thus by DISC. In the case of DISC, sending the payload to the client is delegated to the BE by the FE (recall that the headers follow the normal path). The BE (and its DP) behaves like TCP, meaning that it retransmits lost messages and will stop (and clean up) sending the payload if the connection with EC gets broken.

3.4 TLS support

The integration of TLS in our prototype relies on wolfssl [6]. The design of TLS support extends the work described in [17]. It relies on two functions that behave as serialization primitives: `wolfSSL_tls_export()` (which transforms an `SSL_SESSION object` into an `ASN1` representation) and `wolfSSL_tls_import()` (which performs the reverse operation). In our DISC prototype, feHook and DP are mutually exclusive, i.e., they run on different machines. When feHook invokes the DP to request the emission of a payload, it invokes `wolfSSL_tls_export()` and passes as a parameter the `ASN1` representation of the SSL session. DP receives this representation and then invokes `wolfSSL_tls_import()` to recreate the SSL session. The latter is then associated with the file descriptor used to emit payload chunks directly to the client. Once the full payload has been emitted, DP sends the `ASN1` representation of the SSL session back to the FE. On the FE, kHook receives this representation and restores the SSL session. Note that the encrypted chunks of the payload are buffered in DP so that they can be re-sent if a SACK or DUP ACK is received. The FE cannot emit data toward the client while the SSL session is located on the DP (until the SSL session is returned on the FE).

This main impact of DISC's TLS support is the shift of TLS processing from the FE to the BE. From the datacenter point of view, this is not an overhead but a shift, and this can moreover be seen as a benefit since the FE (which is often a load balancer) can be a bottleneck while the BE is often sharded and replicated for scalability.

3.5 Application integration

We need to modify applications that run on FE, IS, and BE tiers to implement DISC. These modifications are hereafter referred to as feHook, isHook and beHook. In this section, we give pseudo-code, which describes the algorithms of such hooks in the case of HTTP communications. The code has to be adapted to receive and send HTTP responses without payload. As depicted in §3.2, headers (of the DISC-PROT protocol) are added in requests and responses. A DISC-PROT request header indicates whether the current tier can be bypassed, and a DISC-PROT response header indicates whether the payload is present or not.

First, for IS, the program below is a prototype of the code of the IS and its adaptation (in bold). We assume the server has two socket file descriptors: `fdc` and `fds` (respectively towards the client and the server side). This code is executed when a request is received on `fdc` and has to be forwarded to `fds`.

```
1  discprot_req_hdr = receive_discprot_request_header(fdc);
2  update_discprot_request_header(&discprot_req_hdr);
3  send_discprot_request_header(fds, discprot_req_hdr);
4   http_req_hdr = receive_http_request_header (fdc);
5   send_http_request_header (fds, http_req_hdr);
6   payload = receive_http_payload (fdc);
7   send_http_payload(fds, payload);
8     ...
9  discprot_resp_hdr = receive_discprot_response_header(fds);
10 send_discprot_response_header(fdc, discprot_resp_hdr);
11  http_resp_hdr = receive_http_response_header (fds);
12  send_http_response_header(fdc, http_resp_hdr);
13 if (! bypass(discprot_resp_hdr)) {
14        payload = receive_http_payload (fds);
15        send_http_payload(fdc, payload);
16 } else {
17      // NOTHING
18 }
```

Onwards, this program receives the DISC-PROT request header, updates it as described in §3.2, and forwards it (lines

1-3). It then receives and forwards the HTTP request header and payload (lines 4-7). Backward, it receives and forwards DISC-PROT and HTTP response headers (lines 9-12). Then, depending on the information in the DISC-PROT response header (extracted by the `bypass()` function), it receives and forwards the payload if no bypass occurs (lines 13-15).

For FE, the code adaptation is very similar to the above one, with few differences. First, the DISC-PROT request header is not received from the external client but constructed (line 1). Second, when a response is received without a payload (line 17, `NOTHING` comment), FE has to request the emission of the payload by the BE. Here (line 17), FE must execute the following additional code:

```
17    register_range(discprot_resp_hdr);
18    start_transmit(discprot_resp_hdr);
```

`register_range()` registers in the kvs the SN of the packets sent by BE. `start_transmit()` remotely invokes BE (its DP) to start the emission of the payload. The `discprot_resp_hdr` structure includes all the required parameters for these functions.

Finally, for BE, the code below is a prototype of the code of BE and its adaptations (in bold):

```
1   discprot_req_hdr = receive_discprot_request_header(fdc);
2   http_req_hdr = receive_http_request_header (fdc);
3   payload = receive_http_payload (fdc);
4
5   // process request and compute response
6   // (http_response_header and payload)
7
8   discprot_resp_hdr = make_discprot_response_header(discprot_req_hdr);
9   send_discprot_response_header(fdc, discprot_resp_hdr);
10  send_http_response_header(fdc, http_resp_hdr);
11  if (bypass(discprot_resp_hdr)) {
12          dp_send_payload(fd_disc, payload);
13  } else {
14          send_http_payload(fdc, payload);
15  }
```

Onwards, this program receives the DISC-PROT, HTTP request headers, and the request payload (lines 1-3). The application processes the request, and a response is computed. Backward, a DISC-PROT response header is computed (line 8), deciding whether a bypass occurs. The DISC-PROT and HTTP response headers are returned on the `fdc` client socket (lines 9-10). Then, according to information in the DISC-PROT response header (extracted by the `bypass()` function), the payload is either sent to the local DP component[1] (line 12) on a dedicated file descriptor (`fd_disc`) if a bypass occurs or to the preceding tier (line 14) if no bypass takes place.

4 Evaluation

Testbed. We use c220g5 nodes from CloudLab [1]. Each server has 2 Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, 192GB memory, and a 10GB Intel X520-DA2 NIC. They run Debian server 11 with Linux v5.19. For CPU measurements, in order to concentrate on DISC impact on CPU consumption, each tier is executed on one CPU core unless otherwise specified.

[1] As discussed previously (§3.1), the DP buffers the payload locally and sends it to the EC when it receives a transmission request from the FE.

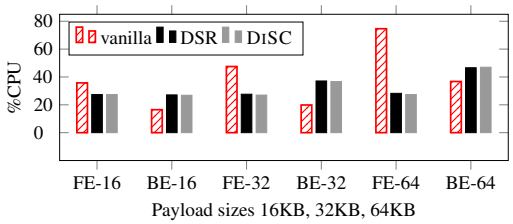


Figure 9: CPU load on each tier in `[FE-BE]` configuration.

Benchmarks. We use one micro-benchmark and four macro-benchmarks to evaluate DISC. The micro-benchmark is a simple N-tier web application that returns static contents of different sizes (16–64KB). The FE and IS tiers only relay requests/responses. We choose the size ranges based on statistics from the Internet/HTTP Archive about typical statistics for HTML pages and images (whose average sizes are respectively 32 and 48KB). It is important to note that the larger the response size, the greater the benefits of DISC.
We use wrk [16] as client, generating 750 req/sec. FE, IS, and BE are implemented with NGINX. Using this micro-benchmark, we can evaluate DISC with various numbers of IS. Concerning the macro-benchmarks, we use SpecWeb [13], SpecMail [12], Train Ticket [22], and Social Media (from the DeathStarBench suite [15]). The first two applications follow a 3-tier architecture (`FE-IS-BE`), presented in §2.1. We use them for the majority of the experiments. We simulate 700 clients. The requested payload sizes are respectively in the [1–64 KB] and [1–100 KB] ranges for SpecWeb and SpecMail. The two other macro-benchmarks, Train Ticket and Social Media, are based on the microservices paradigm. Their characteristics and setups are detailed in §4.5 and §B, respectively.
Goals. We consider five questions: (Q_1) Does DISC effectively offload load from FE and IS, thus addressing the back-pressure problem (§4.1)? (Q_2) What is the impact on scalability (§4.2)? (Q_3) What effect does DISC have on network latency as perceived by the application client (§4.3)? (Q_4) What is the impact of TLS support (§4.4)? (Q_5) What is the impact of DISC on microservices architectures (§4.5)?

We compare DISC with vanilla settings and Direct Server Return (DSR) [19], a standard TCP Handoff [11] solution implemented by many load balancers as stated in §2.3. DSR does not allow fine-grained bypass nor several-tier bypass, and is limited to a single protocol (HTTP). Consequently, we use DSR only for bypassing the FE tier, from its preceding tier which is BE in the `[FE-BE]` architecture or IS in the `[FE-IS-BE]` architecture. Note that, for the 4 macro-benchmarks, we only compare DISC against vanilla because they all are based on architectures featuring multiple protocols and one or several

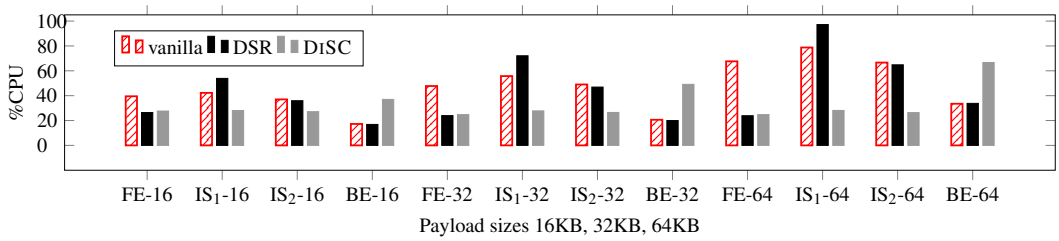


Figure 10: CPU load on each tier in [FE-IS$_1$-IS$_2$-BE] architecture.

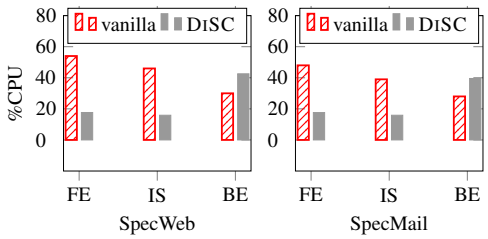


Figure 11: CPU load on SpecWeb and SpecMail tiers.

intermediate tiers.

4.1 CPU load offloading from FE and IS (Q_1)

Micro-benchmark, [FE-BE] architecture. Fig. 9 shows the CPU load for each tier while varying the size of the requested content. With vanilla, FE consumes a significant amount of CPU compared to BE. The gap increases as the content size increases. With DISC and DSR, the load is pushed on BE, where it should since most of the request processing is related to BE. Additionally, with DISC and DSR, as the size of the response increases, the load on FE remains almost constant while the load increases on BE. Overall, DISC and DSR lead to almost the same results. They significantly offload the CPU load on the FE. **DISC efficiently addresses Q_1, it has almost the same results as DSR in a [FE-BE] architecture**.

Micro-benchmark, [FE-IS *-BE] architecture. As explained earlier, DSR can only support a single bypass (of the FE tier). Accordingly, in a [FE-IS$_1$-IS$_2$-BE] architecture, we compare DISC with its vanilla counterpart (with no bypass at all) and with DSR with a bypass of the FE tier only (initiated by IS$_1$). In Fig. 10, we can see that, with vanilla, considerable CPU time is spent on FE and IS, whereas the load on BE does not increase in the same order when we increase the payload size. This is explained by the fact that FE and IS tiers are both receiving, and sending the payload while the BE is only sending the payload. With DISC, the load on FE and IS tiers remains constant. Further, the load is partially pushed on the BE (as it performs additional tasks such as payload buffering and ACK management). For instance, in the micro-benchmark with a 64KB payload size, DISC offloads FE/IS tiers respectively by 63.4% (FE), 64.3% (IS$_1$), 60.4% (IS$_2$), and increases the CPU load on the BE by 98.8%. With DSR, the load on BE and IS$_2$ is the same as vanilla (nothing changed on these tiers), and offload takes place between IS$_1$ and FE. As depicted in Table 1, for the [FE-IS *-BE] 64KB experiment, the cumulative CPU load across all tiers is 246% for Vanilla, 208% for DSR, and 145% for DISC. Thus, a reduction of 41% of the load with DISC compared to vanilla and 30.3% compared to DSR. **In a [FE-IS *-BE] architecture, DISC also efficiently addresses Q_1, it drastically offload load from FE/IS tiers.**

SpecWeb and SpecMail. As shown in Fig. 11, we observed the same trends as above. Overall, DISC achieves a global reduction of the load, about 41% for SpecWeb and 36% for SpecMail (Table 1). **DISC addresses Q_1, we observe a significant offload of the FE/IS tiers, and a limited increase of the load on the BE.**

Scenario	Vanilla	DSR	DISC
[FE-BE] 16KB	52	54 (-3.8%)	54 (-3.8%)
[FE-BE] 32KB	67	65 (3%)	64 (4.5%)
[FE-BE] 64KB	111	75 (32.5%)	74 (33.3%)
[FE-IS*-BE] 16KB	136	132 (2.9%)	120 (11.7%)
[FE-IS*-BE] 32KB	173	158 (8.7%)	128 (26%)
[FE-IS*-BE] 64KB	246	208 (15.4%)	145 (41%)
SpecWeb	130	—	76.11 (41.5%)
SpecMail	115	—	73.06 (36.5%)

Table 1: Cumulative CPU load on all tiers for all our scenarios and reduction percentages with respect to Vanilla.

4.2 Impact of DISC on scalability (Q_2)

Setup. To study scalability of [FE-BE] architectures, we generate requests, with a payload size of 32KB, at a growing rate, i.e., increasing the rate by 2K req/s every 3 seconds. We run this scenario while varying the number of cores on the FE and BE tiers. Initially, each tier is allocated 2 cores. Upon reaching saturation, the number of cores for the saturated tier doubles.

Results, vanilla settings. In Fig. 4-top, using the vanilla settings, we notice that the front-end (FE) is the first tier to reach saturation, primarily because of the load (pressure) propagated from the back-end (BE). It validates the initial findings

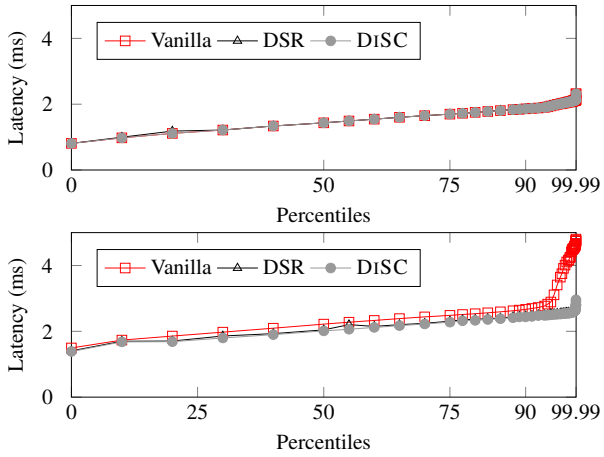


Figure 12: Latency percentiles for the micro-benchmark with the [FE-BE] (top) and [FE-IS$_1$-IS$_2$-BE] (bottom) setups.

presented in §2.2. As soon as the FE saturates, the average request latency raises up, and the doubling of the number of cores is triggered. It appears that with only 2 cores on the FE and BE (on the left side of each sub-figure), tiers achieve a rate of 18K req/s without latency degradation. Having twice the number of cores enables at most a rate of 22K req/s.

Results, DISC settings. In Fig. 4-bottom, using the DISC settings, we observe a notably reduced load on the front-end (FE) in comparison to the vanilla settings. Interestingly, the first tier to reach saturation is the back-end (BE), leading to the increase of: (i) the average latency and therefore (ii) the number of cores (x2). We can observe that with DISC, the tiers achieve a rate of 26K req/s with 2 cores, reaching a plateau at 30K req/s (due to the saturation of the network adapter) with 4 cores.

Compared to the vanilla settings, DISC achieves a significantly higher rate with the same amount of resources: with 2 cores, a rate of 18K for vanilla and a rate of 26K for DISC (an improvement of 45%).

DISC successfully addresses Q_2 by effectively mitigating the backpressure problem. This results in a reduction of the load on tiers preceding the BE, ultimately promoting scalability in the last tiers of a tier chain.

4.3 Impact of DISC on latency (Q_3)

Setup. As previously, the evaluations of latency have been realized with both the micro- and the macro-benchmarks (i.e., SpecWeb and SpecMail). Additionally, we performed evaluation with and without IS tiers involved.

Results, micro-benchmarks. Fig. 12 respectively show the results for zero IS (top) and two IS (bottom) using the micro-benchmarks. From Fig. 12 (top), when no IS is involved, DISC yields nearly identical results to vanilla and

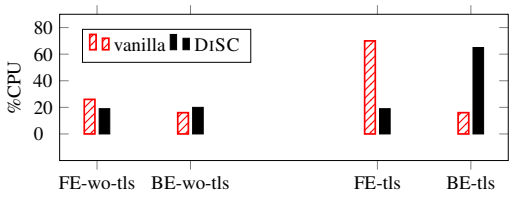


Figure 13: CPU load on each tier in [FE-BE] configuration without and with TLS (micro-benchmark, 32KB payload).

DSR settings. As more IS are involved (Fig. 12 bottom), DISC demonstrates improved latency, with approximately a 0.35ms average difference for 2 IS compared to the vanilla setting and 0.18ms better than DSR. Furthermore, from Fig. 12 (bottom), one can also observe that the tail latency (see the 99.99 percentiles on the figure) is significantly reduced $4.803s \rightarrow 2.959s$. The latency is further improved with more IS tiers, reaching up to $5.71 \times (8s \rightarrow 1.4s)$ for tail latencies with a longer chain of 10 IS (not plotted).

Results, macro-benchmarks. For SpecWeb and SpecMail, we measured that DISC provides a better latency than vanilla by up to 12% in average.

DISC successfully addresses Q_3. It clearly improves latency from the client perspective.

4.4 TLS impact (Q_4)

All the previous experiments were conducted without TLS. To evaluate the impact of TLS, we run the previously used micro-benchmark on a [FE-BE] architecture with and without TLS. Compared to Fig. 9 in §4.1, we reduced the submitted load in order not to saturate CPU with TLS.

As explained in §3.4, when a payload is sent by a BE (when its DP is invoked), the `ssl_session` object is serialized and transmitted from the FE to the DP which deserializes it and uses it for payload encryption. The consequence is that the encryption workload is shifted from FE to BE.

In Fig. 13, we observe the same trend as in previous evaluations. Most of the load is pushed from FE to BE. This is an important benefit for scalability as it prevents FE from becoming a bottleneck. In some way, encryption is part of emission and should be logically handled by the BE without back-propagating the pressure to the preceding tiers.

To answer Q_4, TLS integration in DISC amplifies the mitigation of the backpressure problem.

4.5 Impact on microservices applications (Q_5)

Train Ticket. This macrobenchmark [22] is composed of 43 microservices, which is more complex than the SPEC applications we evaluated above. The microservices communicate with each other via gRPC. An external client request triggers the execution of a specific chain of microservices. The chains vary in length and may involve different sizes of final

data, generated by the database (MySQL). For example, the request corresponding to page `client_adsearch` generates very little final data, which would be irrelevant for DISC. Conversely, the requests triggered by `client_order_list` and several other pages[2] are likely to pull a large amount of final data from MySQL. For our experiments, we focused on the microservices involved in these requests. To illustrate the evaluation of DISC, we use the `client_order_list` request, which involves 4 microservices: `ui` (FE), `gateway` (IS_1), `order` (IS_2), and `mysql` (BE). We keep the default configuration for each service. An important detail to understand the results is that the `gateway` implements traffic shaping, calibrated at 20 req/s.

The modifications made to the application to integrate DISC primarily consisted of extending the SQL language to signal when the return path of a microservice can be bypassed. We also extended the MySQL JDBC driver [9] to handle this SQL extension and implement DISC's DP component.

For this experiment, we use virtual machines (VMs), unlike the previous experiments, which were conducted on bare-metal. This allows DISC to be evaluated in typical application execution environments. Each service is deployed in a separate VM (2 vCPU and 2GB of memory), resulting in a total of 43 VMs. To simplify the interpretation of results, we opted for a setup without microservice replicas. In this virtualized environment, for the `MySQL` VM, the virtual network interface (virtio) is configured with multiple transmission queues, with one queue specifically dedicated to XDP. Using wrk2, we launched 400 clients, each requesting a payload size of 32KB.

Fig. 14 shows the results. We observe that DISC does not significantly reduce the CPU load on the `gateway` compared to other microservices. This is because the `gateway` applies rules to throttle incoming requests upon reception (traffic shaping), which adds computational overhead. To support this, we conducted experiments with a 0B payload and we observed that the CPU load on the `gateway` remains high, indicating that it is primarily driven by the computations from its traffic shaping module. The load on the `gateway` increases with DISC, compared to the vanilla setup, when the payload is 0B because DISC allows the application to receive more traffic, resulting in more traffic shaping processing on the `gateway`. DISC shifts the CPU load from the `order` service (49% reduction w.r.t. vanilla), while moderately increasing it on the MySQL BE (+18.5%).

We also observe that DISC improves both latency and throughput. Average latency decreases by 74.1%, from 3.57 to 0.928 seconds, and throughput increases by 39.8%, from 635.80 to 889.20 req/s.

Social Media. Due to a lack of space, we only provide a brief summary of the results for this macro-benchmark. More details on the benchmark, setup and experiments, are available in Appendix B. Similarly to Train Ticket (§4.5), DISC shifts

[2] `admin`, `admin_travel`, `admin_user`, `client_consign_list`, `client_enter_station`, and `client_ticket_collect`.

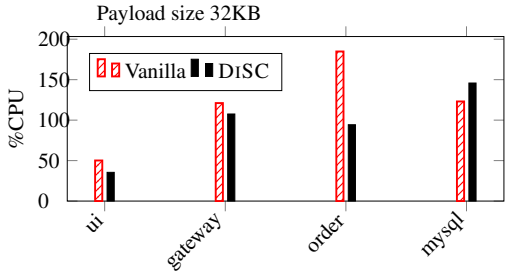


Figure 14: CPU load on the Train-ticket microservices triggered by a `client_order_list` request. `ui` and `mysql` correspond to FE and BE, respectively.

the CPU load from frontend to backend tiers. Compared to vanilla, DISC offloads the frontend tiers by up to 41.7% and increases the backend tier by up to 48.8%. Moreover, DISC reduces the average latency by up to 66.7% by dropping it from 510ms to 221ms, while increasing the throughput by 21.9%, from 80355 req/s to 98009 req/s.

DISC successfully addresses Q_5: its benefits are applicable to coarse-grained as well as to microservices applications.

5 Conclusion

This paper tackles the backpressure problem, which compromises both: (i) the independence of tiers regarding performance, and (ii) the scalability of multi-tier architectures. We presented the drawbacks of state-of-the-art solutions and identified six requirements (flexible, protocol agnostic, transparent, pervasive, consistent, TLS support) that a solution should meet to efficiently address this problem in a holistic manner. We introduced our solution, DISC, which meets all these requirements. With a prototype built atop TCP, we demonstrated that DISC significantly: (i) reduces the CPU load, and (ii) improves tail latencies on heavy-load scenarios.

Note that, like CRAB [20], DISC relies on IP spoofing, which is generally not possible for IaaS tenants. Thus, in the context of public cloud platforms, DISC can only be deployed by cloud providers to complement (or replace) their existing offerings (load balancers and managed applications).

Acknowledgements

We thank our shepherd, Ankit Bhardwaj, and the anonymous reviewers for their insightful comments. Théo Boury, Maxime Bruno, and Adrien Vannson contributed to early versions of the DISC implementation. This work was supported by the French *Agence Nationale de la Recherche* under the ANR projects "PicNic" (ANR-20-CE25-0013) and "PEPR DiVa" (ANR-23-PECL-0004), by the CNRS "MLSN2" International Research Project, and by Inria "Défi OS".

References

[1] CloudLab. https://www.cloudlab.us/ (Accessed on 2023-04-12), 2023.

[2] Connection Migration in QUIC. https://datatracker.ietf.org/doc/html/draft-tan-quic-connection-migration-00 (Accessed on 2023-04-16), 2023.

[3] Dovecot. https://www.dovecot.org/ (Accessed on 2023-12-06), 2023.

[4] HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. https://www.haproxy.org/ (Accessed on 2023-03-20), 2023.

[5] NGINX. https://www.nginx.com/ (Accessed on 2023-03-23), 2023.

[6] wolfSSL Embedded TLS Library. https://www.wolfssl.com/ (Accessed on 2023-03-20), 2023.

[7] Zimbra email & collaboration. https://www.zimbra.com/ (Accessed on 2023-10-18), 2023.

[8] Apache Thrift. https://thrift.apache.org/ (Accessed on 2024-07-23), 2024.

[9] MySQL Community Downloads - Connector/J. https://dev.mysql.com/downloads/connector/j/ (Accessed on 2024-06-18), 2024.

[10] Carl Alexander. A Look At The Modern WordPress Server Stack. https://www.smashingmagazine.com/2016/05/modern-wordpress-server-stack/ (Accessed on 2024-05-05), 2016.

[11] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-Based Networks Servers. In *Proceedings of the USENIX Annual Technical Conference*, USA, 2000. USENIX Association.

[12] Standard Performance Evaluation Corporation. SpecMail2009. https://www.spec.org/mail2009/ (Accessed on 2023-12-06).

[13] Standard Performance Evaluation Corporation. SpecWeb2009. https://www.spec.org/web2009/ (Accessed on 2023-12-06).

[14] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[15] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Will Glozer. wrk - a HTTP benchmarking tool . https://github.com/wg/wrk (Accessed on 2023-12-06).

[17] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the Pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 535–549. USENIX Association, April 2021.

[18] Guerney Hunt, Erich Nahum, and John Tracey. Enabling Content-Based Load Distribution for Scalable Services. Technical Report TR-97. IBM T.J. Watson Research Center, 1997.

[19] Matt Klein. Introduction to Modern Network Load Balancing and Proxying. https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236/ (Accessed on 2024-05-05), 2017.

[20] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the Load Balancer without Regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 193–207, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association.

[22] Fudan University Software Engineering Laboratory. Train Ticket - A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket/tree/master (Accessed on 2024-05-05), 2018.

[23] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik

Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Carlo Puliafito, Luca Conforti, Antonio Virdis, and Enzo Mingozzi. Server-Side QUIC Connection Migration to Support Microservice Deployment at the Edge. *Pervasive and Mobile Computing*, 83:101580, 2022.

[26] Aigars Silkalns. WordPress Statistics: How Many Websites Use WordPress in 2024? https://colorlib.com/wp/wordpress-statistics/ (Accessed on 2024-05-05), 2024.

[27] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-Grained Failover Using Connection Migration. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)*, San Francisco, CA, March 2001. USENIX Association.

[28] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22)*, EuroSys '22, page 352–367, New York, NY, USA, 2022. Association for Computing Machinery.

[29] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A Distributed Futures System for Fine-Grained Tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021.

[30] Ziqi Wei, Zhiqiang Wang, Qing Li, Yuan Yang, Cheng Luo, Fuyu Wang, Yong Jiang, Sijie Yang, and Zhenhui Yuan. QDSR: Accelerating Layer-7 Load Balancing by Direct Server Return with QUIC. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 715–730, Santa Clara, CA, July 2024. USENIX Association.

Appendices

The appendices are organized as follows: Appendix A provides implementation details regarding the implementation of the different components of DISC, Appendix B provides additional details regarding the Social Media benchmark, setup and experiments (discussed in §4.5).

A Implementation details

DISC is implemented with a kernel module (kHook, kvs), two user-level modules (ackSender and DP) and a set of L7 hooks to act at the L7 layer (i.e., feHook, isHook, beHook). kHook relies on BPF (extended Berkeley Packet Filter) for intercepting and adapting packets at the L4 layer. kvs is implemented with BPF maps (a generic storage system for sharing data between the kernel and the user space). L7 hooks can be implemented with patches or application extensions when such a support is provided (e.g., NGINX allows the definition of filters). The overall design of DISC is depicted in Figure 5. The implementation details of each component is given below.

kHook. The kHook is activated on each server of the datacenter hosting a tier that may act as FE. kHook is composed of three interceptors called `in_tc`, `out_tc`, and `in_drv`. Both `in_tc` and `out_tc` are implemented in the TC (Traffic Control) layer while `in_drv` is integrated in the NIC driver. kHook could be fully implemented at the TC level, but we introduced `in_drv` for performance reasons (to prevent needless walks in the protocol stack). `in_tc` intercepts incoming TCP connections and initializes kvs with the characteristics of the established connection. All other incoming packets are handled by `in_drv`.

`in_drv` intercepts all packets received from the client after connection. Such packets may either include data or not, and must include an ack. If the packet includes data, then data is transmitted to the stack of FE. But anyway (whether the packet includes data or not), acks must be handled, so that they are consistently (regarding sequence numbers SN) sent to the stack of FE or to DP (which emitted the payload). If there were not any bypass yet, packets are transmitted to the FE stack as is. As soon as a bypass occurred, SN included in acks have to be translated as depicted in §3.3. A map of the SN that were emitted (returned to the client) by the FE stack and the DP is registered in the kvs. This map includes emitted SN ranges and the IP addresses of the emitting servers. Both are kept in the kvs as long as they are not acknowledged by EC. A SN in a received ack that acknowledges data from a range generates an ack which is sent to the emitting server of the range (FE or DP) with the highest acknowledged SN from the range (notice that several ranges may be acknowledged by an ack). If the generated ack is transmitted to FE, the SN is translated back into the SN as sent by the FE. If the generated ack is to be sent to the DP, it is actually stored in kvs and consumed by the ackSender process.

Regarding out_tc, it translates SN sent by FE to the client, taking into account the global SN state (see §3.3) that is returned when DP is invoked (for sending a payload).

lib_kHook. This library provides an API for instantiating kHook within the kernel and for accessing kvs. It provides: (i) setup_khook() to instantiate the hook, (ii) poll_ack() to poll acks from kvs to be sent to DP by ackSender, and (iii) register_range() to register a packet range sent by BE.

register_range() is called by feHook after the reception of the response header that includes in its DISC-PROT header the ids (identifier) of the payload, its size and IP of the BE (location of DP). Those parameters are passed to register_range(), which registers in kvs the packet range associated with the payload and information for handling acks for these packets (IP of BE and ids). register_range() also receives as parameters the SN associated with the registered packet range (deduced from the global SN state and the size of the payload) and also the IP address and port of the client. Notice that register_range() also receives a callback function, invoked by kHook when the client receives the full payload (thus allowing to handle protocol footers, e.g., in the case of IMAP).

lib_comm. This library implements an API for communication between DISC components located on different machines. It provides two key functions: (i) start_transmit() to remotely invoke a DP, and (ii) forward_ack() to forward acks to a DP. start_transmit() takes as parameters the IP of BE and ids of the payload, address of EC (IP and port) and the global SN state to be used for sending the payload.

ackSender. ackSender is a process started on each FE. It relies on poll_ack() from lib_kHook and forward_ack() from lib_comm to extract acks from kvs and send them to DP. Since acks can be any of the following ACK, SACK, DUP ACK and ZERO WINDOW, DP can therefore manage the buffering of the payload (freeing buffers), the re-emission of lost packets and its emission window.

feHook. feHook calls register_range() after the reception of the response header to store in kvs information about the emitted payload. It then calls start_transmit() to request the emission of the payload from DP.

beHook and DP. The implementation of DP relies on two modules called ctr_msg_manager and payload_sender. Both modules include a thread for handling to incoming requests. ctr_msg_manager handles requests issued by FE when either: (i) an ack is forwarded *via* forward_ack(), or (ii) the emission of a payload is requested *via* start_transmit()). payload_sender handles the emission of payloads (i.e., packets on the connection preceding the FE).

When beHook (which acts at the L7 layer of BE) decides to send a payload with bypass, it asks ctr_msg_manager to create a job that is inserted into a red-black tree structure. This job includes a unique identifier (ids) of the payload, and a data structure representing the content of the payload. In our prototype, the content of the payload in a job can be of two types. In the first case (type 1), the payload content is a buffered stream of data sent by the L7 layer of BE. BE starts filling the buffer and blocks until buffers are freed by received acks. In the second case (type 2), the payload content is a file, meaning that emitted packets are directly read from a file. This is equivalent to the sendfile() Linux primitive, which directly copies data from a file descriptor to a socket descriptor.

When feHook (on FE) calls start_transmit() to request the emission of the payload, ctr_msg_manager receives that request, and retrieves the job identified by ids from the red-black structure. It adds in the job the parameters received in the request (IP address and port of the client, SN associated with the payload, size of the reception window of the client) and asks payload_sender to insert a task in its internal task list (a FIFO list of tasks consumed by the handler thread). A task is a request to emit a data chunk. When starting transmission of a payload, the created task corresponds to the emission of a chunk whose size is the reception window of the client. Then, when ctr_msg_manager receives an ACK, DUP_ACK or SACK, it implies that either a chunk has to be re-sent or that an additional chunk within the payload can be sent. Therefore, ctr_msg_manager handles such acks by adding tasks in the task list. The handling of a task reads the chunk either from the buffer or the file (according to the job) and generates a set of packets on the TCP connection preceding the FE (thus DP is spoofing FE).

B DISC with the Social Media benchmark

We perform additional experiments to show how DISC performs with complex microservices. To meet this aim, we modified the Social Media benchmark from the DeathStarBench benchmark suite [15]. The Social Media benchmark comprises 36 services, primarily written in C, C++, Java, and Node.js, communicating through Thrifts RPC [8]. The storage services run MongoDB with caching services that run Memcached or Redis. A client request queries a load balancer, which distributes a client request across Nginx web servers. The request processing chain length depends on the client's request input. Like for Train Ticket (§4.5), to apply DISC, we modify the storage and caching layers and their connectors (MongoDB, Memcached, and Redis) to obtain the microservices that can be bypassed when receiving a read or write request. Consequently, we reduce the load on the intermediate nodes hosting the bypassed microservices.

We deploy our modified version of the Social Media benchmark where each service runs in VMs that have the same size as the ones from Section 4.5 (2vCPU and 2GB memory) and

use the built-in load generator (built atop wrk) to evaluate the impact of DISC. Concretely, we use the load generated to simulate 1000 clients that read randomly previously registered posts for 600 seconds. 1000 clients is enough to stress the frontend tiers cores up to 70% overall. Each client triggers a request every 10ms. We chose this scenario because it is among the scenarios with the longest request processing chain. The input payload size for such a request is 4KB, and the response payload size depends on the requested post, which varies from 64KB to 8.2MB. During the experiment, we record the CPU usage on all microservice nodes. Figure 15 reports the CPU usage on the frontend tier `LB`, the intermediate (IS) tiers `Media frontend, Nginx` and the backend tier `MongoDB`. The caches hit ratio is, on average, 2.8%. Thus, we do not focus on the CPU load on the caching nodes.

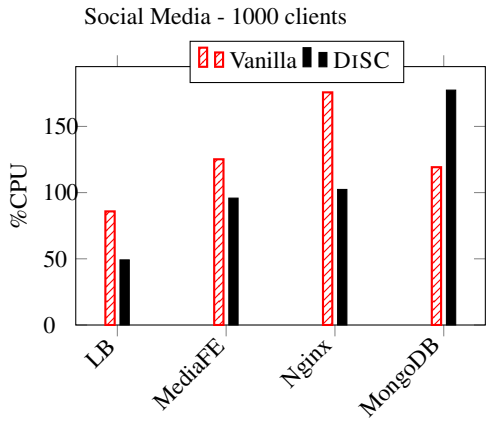


Figure 15: CPU load on the main microservices triggered during the execution of the read post requests performed by 1000 clients during 600s. `LB` correspond to the `FE`, `Media frontend (MediaFE), NGINX` correspond to `IS`, and `MongoDB` correspond to the `BE`.

Similarly to Train Ticket (§4.5), DISC shifts the CPU load from frontend to backend tiers. Concretely, DISC offloads the frontend tiers by up to 41.7% and increases the backend tier by up to 48.8%. Moreover, DISC reduces the average latency by up to 66.7% by dropping it from 510ms to 221ms, while increasing the throughput by 21.9%, from 80355 req/s to 98009 req/s. Thus, DISC efficiently shifts load from FE to BE, improving overall end-end latency and throughput compared to vanilla.

Enabling Silent Telemetry Data Transmission with InvisiFlow

Yinda Zhang[†], Liangcheng Yu[†⟆], Gianni Antichi[§⋆], Ran Ben Basat[‡], Vincent Liu[†]
[†]*University of Pennsylvania,* [⟆]*Microsoft Research,* [⋆]*Queen Mary University of London,*
[§]*Politecnico di Milano,* [‡]*University College London*

Abstract

Network applications from traffic engineering to path tracing often rely on the ability to transmit fine-grained telemetry data from network devices to a set of collectors. Unfortunately, prior work has observed—and we validate—that existing transmission methods for such data can result in significant overhead to user traffic and/or loss of telemetry data, particularly when the network is heavily loaded.

In this paper, we introduce InvisiFlow, a novel communication substrate to collect network telemetry data, silently. In contrast to previous systems that always push telemetry packets to collectors based on the shortest path, InvisiFlow dynamically seeks out spare network capacity by leveraging opportunistic sending and congestion gradients, thus minimizing both the loss rate of telemetry data and overheads on user traffic. In a FatTree topology, InvisiFlow can achieve near-zero loss rate even under high-load scenarios (around 33.8× lower loss compared to the state-of-the-art transmission methods used by systems like Everflow and Planck).

1 Introduction

Network management and monitoring tasks often rely on fine-grained telemetry data to empower network administrators and systems with insight into current conditions [1–3]. For instance, when monitoring for failures and other network anomalies, switch telemetry enables the detection of unusual traffic patterns, errors, or drops [3–5]. Similarly, when optimizing or provisioning the network, administrators can analyze trends in traffic volume, bandwidth usage, and port utilization to make decisions [6–9].

Recent research has provided compelling arguments in favor of expanding the scope and diversity of collected telemetry data. Rationales vary. Some researchers note a reduction in the timescales of congestion events and its impact on the granularity required to observe them [10–12]. Others propose enhanced measurement capabilities enabled by modern network devices that can capture buffer statistics, sketches, per-QoS counters, and more [13–16]. Still, others advocate for the advantages of per-packet in-band metrics [17–19], and some suggest increasing telemetry to strive for the highest levels of availability and Service Level Objectives (SLOs) [20–22].

Regardless of the reason, high volumes of telemetry data produced at switches are typically collected at machines placed throughout the network that are responsible for processing, collating, analyzing, and storing the gathered data [1,3–5]. The traditional method used to send telemetry reports (*e.g.*, in [23]) routes them as standard data packets that can impact user traffic. For example, in §2, we observe increases in user traffic flow completion time of up to ≈19%, which affects latency-sensitive workloads. Techniques like In-band Network Telemetry (INT) [18], In-situ Flow Information Telemetry [24], and Broadcom's IFA [25] that embed data directly into user packets impose similar overheads [19].

Acknowledging this issue, recent work has tried to reduce overheads on user traffic by sacrificing the completeness of the telemetry data. One approach is exemplified by systems like [19, 26] that reduce telemetry's frequency and/or volume through approximation, saving some bandwidth but at the cost of information loss. When tracing paths of flows, for instance, PINT [19] will miss path changes for small flows (large flows as well under flowlet routing). Even when incomplete/approximate data is sufficient, the data is still eventually transferred across the network, incurring the associated overheads of those transmissions. Another approach is taken by systems like [21, 27–30] that de-prioritize telemetry traffic in favor of user traffic. This solution, however, risks starvation and telemetry loss when links are shared. The loss can affect the accuracy/utility of monitoring applications that rely on the data. For instance, our benchmark of a path tracing utility missed around 11% of paths under this strategy (§2).

In this paper, we aim to design a novel communication substrate for telemetry data that provides both (a) high sustainable throughput (*i.e.*, throughput given the constraint of zero loss) and (b) little-to-no impact on user traffic. We present InvisiFlow, a new mechanism to isolate telemetry data from user traffic. InvisiFlow is agnostic to both the content of the telemetry data and the network's topology; consequently, it can be used as a drop-in replacement for many existing monitoring systems.

The main insight behind our work is that exploring unused network bandwidth using *congestion gradients* maximizes sustainable throughput and minimizes the external impact of telemetry traffic—similar in spirit to water always flowing toward a lower elevation, telemetry packets are sent along the gradient to neighbors with lower congestion and more space in their telemetry buffers. To tackle challenges in applying existing theory on congestion gradients (§3.2) to practical environments, InvisiFlow constructs a pull-based transmission channel in which network switches frequently send low-priority pull requests to their physical neighbors that quantify the amount of telemetry congestion at each node, and utilizes shared pipelines in switches to support the late-binding of egress ports based on telemetry buffer usage.

We implement a hardware prototype[1] of InvisiFlow on a commodity switching ASIC with P4 programmability. With testbed experiments on a leaf-spine topology, we demonstrate that InvisiFlow can efficiently explore available network bandwidth to simultaneously prevent impact on user traffic *and* loss of telemetry data, even under high load. In contrast, baselines based on the communication protocols of state-of-the-art telemetry systems exhibit drop rates of >80% when prioritizing user traffic on the same workload. Simulation results on larger networks and a range of workloads, network applications, and topologies show that InvisiFlow remains resilient and effective across a wide range of scenarios. Specifically, InvisiFlow can improve the accuracy of prior systems in per-flow accounting approximation (*i.e.*, ApproSync [31] with Count-Min sketches [32]) up to 22.5×.

2 Background and Motivation

We begin by briefly summarizing existing telemetry data transmission systems and their limitations.

2.1 Telemetry Data Transmission

Network operators design and deploy systems to collect data from the network, including host/switch counters, flow records, and log information. While some systems ingest and exploit such data locally [8, 33], most of the time, operators need to collect the data at centralized databases/repositories for subsequent processing, storage, and analysis to provide a comprehensive view of the network [34, 35].

These modern telemetry systems—regardless of their target data—share a high-level workflow, depicted in Figure 1. The systems comprise three major components. The *source* can reside on any network device (*e.g.*, switches, routers, or end hosts), encompass any target data (*e.g.*, SNMP port utilization counters, drop notifications, or NetFlow records), and may be generated at any interval, periodic or event-based. A set of *collector sinks* receive, process, and store this telemetry. Monetary incentives typically imply that the number of collectors is small compared to the number of devices (*e.g.*, < 1% as shown in Everflow [28]) and that they must aggregate telemetry data from multiple devices.

Between the above two components is a *transmission channel*, responsible for forwarding telemetry packets from sources to collectors (which is the focus of this paper). We note several requirements for this transmission channel:

(R1) High/full sustainable throughput for telemetry data. As the mechanism responsible for delivering telemetry data, network applications depend on this channel to provide high throughput with minimal loss of data. Finer-grained telemetry data allows network operators to make more informed decisions [10] and provide better services to end users; data loss can result in incomplete or inaccurate insights [30].

[1] Code is available at https://github.com/eniac/InvisiFlow.

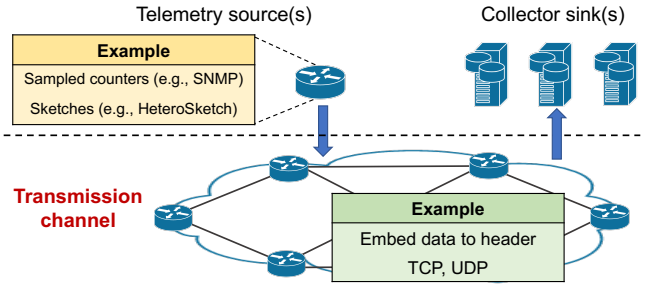


Figure 1: Workflow of a typical telemetry system.

(R2) Minimal impact on user traffic. Telemetry data often shares the same network with user traffic [19, 26, 29] as this can lead to more efficient resource utilization and beneficial fate-sharing properties [21]. While doing so, it is crucial to avoid introducing excessive latency, packet loss, or congestion due to the telemetry data transmission process, which may degrade the quality of service for user applications. As discussed in [19], this overhead can lead to up to 20% degradation of application-level throughput and a corresponding 25% increase in flow completion time.

(R3) ASIC-implemented operation. Finally, while early switches and routers often punted all non-forwarding tasks (including telemetry) to the switch CPU, growing telemetry granularity and ASIC capabilities have led to heavier reliance on data planes to both generate and send telemetry data [19, 27–30]. In particular, data planes can collect and export data at much higher resolution and power efficiency than CPUs. This trend toward data plane implementation is visible in line-rate monitoring systems like INT [18] and Broadcom's IFA [25], but it is also increasingly true for classic, decades-old telemetry systems like NetFlow and sFlow [36, 37].

2.2 Existing Solutions and Limitations

We note that recent work [14, 15, 19, 26, 38, 39] that seeks to reduce the size of collected data at the telemetry source (encompassing techniques like sampling, sketching, and encoding) can mitigate some of the above challenges; however, they are not a panacea. First, these solutions often sacrifice generality for the required bandwidth, limiting their applicability to specific applications and types of telemetry tasks. The approximation can also introduce significant errors in higher-layer analysis [5]. For example, applying PINT [19] to path tracing and the Hadoop workload [40] of §6 can limit bandwidth overheads to <2% compared to a configuration without telemetry but will miss path records for >60% of flows that are too small to track. Finally, regardless of whether the techniques are applicable/acceptable, enhancing the underlying transmission channel can mitigate the degradation in the accuracy of higher-layer applications and its impact on user traffic.

We, therefore, focus on the orthogonal problem of telemetry transmission. In this subsection, we examine the channels used by existing systems in the context of the above requirements. Per R3, we look only at data-plane solutions, of which current systems take one of the following two approaches.

Tx Method	Examples
Default-priority UDP	NetFlow [36], sFlow [42], INT-XD (Postcard) [29]
Low-priority UDP	Everflow [28], Planck [27]

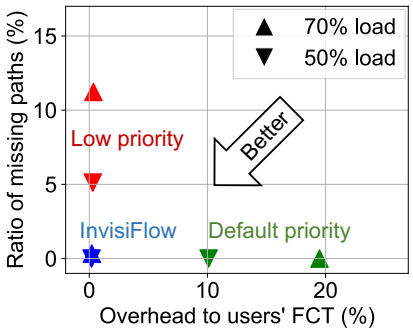


Figure 2: Tradeoffs between different prioritization methods for the telemetry transmission channel under the path tracing application. FCT dilation is an average over applications in the same rack as a collector.

Embedding data into packets. One way to transmit telemetry data is to embed data directly into user packets [18, 19, 26, 41]; end-host receivers are responsible for extracting the telemetry data from these packets and sending it to collectors. Unfortunately, as others have noted and we discussed in R2, embedded data can introduce substantial overheads [19]. For example, for small packets, adding additional data to packets can increase transmission times by a large relative proportion, increasing load on the network. For large packets, reserving header bits or reducing MTU sizes to prevent fragmentation incurs a tax on every packet, regardless of whether real data is embedded or not.

Generating packets and sending them directly. The other approach is to generate dedicated packets that are sent to collectors directly [27–30, 36, 42, 43]. These systems typically layer on top of existing protocols like UDP, which is preferable to TCP due to the impracticality of implementing end-to-end reliability in the data plane [44]. By sending telemetry data as datagrams separate from user packets, these systems can better control the rate and overhead of telemetry data transmission.

However, managing the tradeoff between the throughput of the transmission channel (R1) and its impact on user traffic (R2) can be challenging. Aside from generating less telemetry data, the primary knob to tune the overheads of today's transmission channels is Class-of-Service packet prioritization, where there are two primary options: (a) treat telemetry and user packets equally or (b) send telemetry data with a lower priority than user traffic.

Figure 2 categorizes a few popular telemetry systems along this axis and illustrates the challenge of balancing sustainable telemetry throughput and user-traffic overhead (measured via FCT dilation). Results are presented for a FatTree data center [45] using a Meta Hadoop workload [40] and a range of average utilizations, as detailed in §6.1. We employ a path tracing application similar to [5], where each switch maintains a record of user flows and forwards it to a collector for path reconstruction. As seen in Figure 2, as load increases, network operators are forced to choose between prioritizing telemetry (and increasing the FCTs of collector-collocated applications by 19%) or prioritizing user traffic (and accepting 11% missing paths). As a preview of our results, InvisiFlow can keep both minimal, even in a heavily loaded network.

3 Design Overview

In this paper, we present InvisiFlow, a communication substrate for telemetry data transmission that can minimize the network-wide loss rate of the telemetry data given the constraint of little-to-no impact on user traffic. As previously mentioned, InvisiFlow is agnostic to the content of the telemetry data and the network's topology. It also leaves user traffic transmission untouched. As such, it can be used as a drop-in replacement for many existing monitoring systems.

3.1 Opportunistic Gradient Forwarding

InvisiFlow is driven by two overarching design principles:

1. Prioritize user traffic.
2. Instead, seek out spare capacity, wherever it may be.

For the first principle and to satisfy R2, InvisiFlow keeps telemetry and user traffic separate and ensures that all control and telemetry packets are de-prioritized compared to user packets. Although packet processing and transmission are generally not preemptable in modern network devices, prior work [21, 27] and our results in §6.3 show that proper configuration of priorities at every stage of modern network devices can ensure that overheads are negligible, even when every available gap between user packets is filled with lower-priority packets. With these techniques, InvisiFlow can operate with minimal impact on user traffic.

Instead, InvisiFlow seeks out spare network capacity for telemetry data transmission, aggressively leveraging any such opportunities. Over a decade of measurement studies on large-scale networks have frequently observed low average utilization in modern networks [10, 40, 46–48], with data centers, for instance, sometimes seeing ∼5% average utilization (comprised mostly of small bursts), even for supposedly high-throughput applications like Hadoop and distributed ML training. Telemetry, generally off the critical path of user-facing requests, is a particularly good fit for leveraging the opportunities provided by these gaps in sending.

InvisiFlow's approach to finding spare capacity is based on classic theoretical foundations regarding multicommodity flows and the stability of queuing networks [49, 50]. More formally, given a network and a vector of flows with associated sources, sinks, and arrival rates, a scheduling/routing policy π is said to provide buffer stability if it satisfies the following.

Definition 1 (Buffer Stability)**.** *Let $B(t)$ be the total amount of packets remaining in the buffers at time t. Assume no*

constraints on buffer size. The system is stable under π iff:

$$\limsup_{t\to\infty} \frac{1}{t} \sum_{\tau=0}^{t-1} \mathbb{E}[B(\tau)] < \infty \quad (1)$$

The alternative to buffer stability is a system where buffer sizes grow to infinity. Note that it is often useful to define scheduling policies that are robust to changes in arrival rates. Prior work has termed this a problem of characterizing and maximizing a policy's stability region.

Definition 2 (Stability Region)**.** *Let g be the vector of the packet arrival rates at all telemetry sources. Stability region S_π of policy π is the set of all g for which the system is stable under π.*

Results in *max-weight scheduling theory* [50] have shown that transmission based on the *congestion gradient* can provide an optimal policy that maximizes the stability region and sustainable throughput. The congestion gradient is defined as the difference in telemetry buffer usage between neighboring switches. Unlike traditional forwarding [51], where nodes blindly push packets along the shortest path to the destination, for every pair of neighbors in congestion gradient-based transmission, the one with a larger telemetry buffer usage sends toward the one with a smaller buffer usage. This is similar in spirit to water always flowing toward a lower elevation. These approaches have been found to be robust to diverse topologies and dynamic arrival patterns. As with prioritization, we argue that telemetry, typically amenable to asynchronous and out-of-order delivery, is a particularly good fit for our approach.

3.2 InvisiFlow Architecture

While InvisiFlow is inspired by the simplicity and efficacy of prioritization and congestion gradient forwarding, the theory is, of course, significantly different from practice.

(C1) How do we calculate the gradient over time and address situations where the gradient itself is not known (*e.g.*, because control packets are subject to prioritization behind user traffic)? (C2) How do we address limitations in packet processing pipeline structures for sending telemetry data, *e.g.*, the need to specify an output port before egress availability is known? (C3) How do we handle failures of both network components and telemetry collectors? The primary challenge of InvisiFlow is to develop practical solutions to the above issues and others.

Components. The high-level architecture of InvisiFlow is shown in Figure 3. It contains several types of components:

- *Switches:* To obtain the gradient (C1), every switch in the network, when it has free capacity, will continually generate low-priority pull requests (with its current telemetry buffer usage) and send them to all neighboring switches. Neighboring switches will respond to this per-hop pull request if the other side's buffer usage is lower. The current switch may also send packets if it would otherwise have dropped data. To enable the late binding of egress ports for telemetry data (C2), we store telemetry data in the egress pipeline, which is shared by multiple egress ports, as detailed in §4.1.
- *Collectors:* Telemetry collectors operate similarly to today, ingesting and processing data as it arrives. In InvisiFlow, they also act as true sinks in the congestion gradient, equivalent to having a constant telemetry buffer utilization of zero. This can be implemented by either the collector sending pull requests on its own or via offload to its last-hop switch [43]. In either case, telemetry data is naturally drawn in from surrounding devices. InvisiFlow supports sharding and backups of these devices and can support flow control through an extension of the switch protocol.
- *Available Servers:* Finally, InvisiFlow supports optional usage of spare server memory capacity, similar to [52, 53]. These also extend the per-hop pull protocol of the switches, with their advertised utilization providing a knob for tuning their usage. For example, advertising 80% usage means that they will only be used when nearby switches are >80% full. Configuring these servers is optional but can be beneficial when there is a high load or failures of either network components or telemetry collectors (C3).

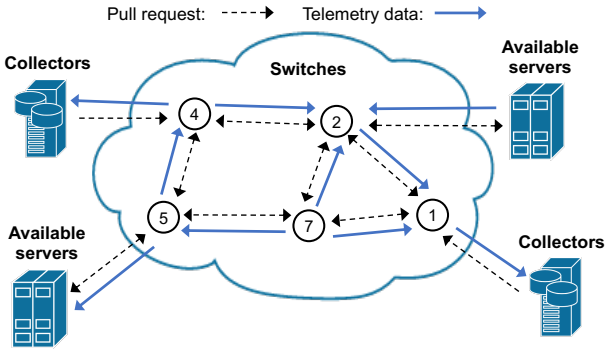
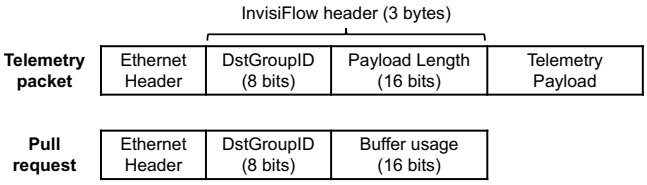


Figure 3: Overview of an InvisiFlow-enabled network.

Workflow. InvisiFlow creates a network where switches continually trade pull requests and telemetry data with their neighbors in the gaps between microbursts of user traffic. Through these gaps, InvisiFlow implements a variant of the edge-balancing approach to approximating a maximal-stability-region routing policy. As long as gaps exist, InvisiFlow will leverage them to route telemetry packets. In the rare case that one or both directions of a link are occupied for an extended period, InvisiFlow will leverage congestion gradients to find alternate paths and take risks when appropriate.

While inevitably not a perfect reproduction of the formalisms of results like [50] (*e.g.*, due to the asynchrony of pull requests), our extensive evaluation over a wide range of topologies and workloads in §6 demonstrates that InvisiFlow can—entirely in the data plane (R3)—maximize the sustainable throughput of telemetry data (R1) while incurring little-to-no impact on user traffic (R2).



Ethernet header: *Ethertype* identifies the InvisiFlow header.
DstGroupID: The destinations (collectors) group ID for the packet.
Payload length: The total length of the telemetry payload measured in bytes.
Buffer usage: The number of telemetry packets recorded in the telemetry buffer.
Telemetry payload: The telemetry data provided.

Figure 4: The packet format of InvisiFlow

Ethertype	Egress port
Generated seed packet	Multicast to all neighbors
Pull request	Ingress port
Telemetry packet	Free ports

Table 1: Default routing table used by the ingress pipeline.

4 The InvisiFlow Transmission Channel

In this section, we discuss the design of the InvisiFlow transmission channel in detail. To keep our discussion general, we assume (a) telemetry sources can be placed at arbitrary locations in the data plane and generate arbitrary amounts of data, and (b) one or more collectors are placed in the network, any of which can ingest that data. Note that InvisiFlow can also support differentiated collectors and sharding (*e.g.*, based on hashing over a partition key) through a `DstGroupID` header field and a straightforward extension to parallel transmission channels, but we omit that discussion for simplicity.

Message format. The packet formats of InvisiFlow messages are illustrated in Figure 4. There are two types of packets, both gated by `DstGroupID` as mentioned above. Pull requests are generated periodically and contain the current telemetry buffer usage, which can be scaled to account for heterogeneous buffer allocations. Telemetry data packets are only sent in response to received pull requests or upstream telemetry, and they contain a variable-length payload. Both types of packets are set to the lowest priority at every stage of the pipeline, including the ingress arbiter, traffic manager, and egress arbiter, per [21].

4.1 Switch Operation

The routing table for InvisiFlow's ingress pipeline and the pseudocode for egress pipeline processing logic are provided in Table 1 and Algorithm 1, respectively.

An important building block of this logic is the gap-filling low-priority packet generation mechanism introduced by the OrbWeaver framework [21]. OrbWeaver uses the existing per-pipeline packet generator capabilities of modern switches to generate low-priority packets sent to all ports with configurable frequency and payload, with negligible impact on the average power draw of the switch (<2%) and minimal worst-case impact on pipeline bandwidth (<1.5%). Specifically, it periodically generates 'seed' packets in the ingress pipeline

Algorithm 1: Egress logic for InvisiFlow packets.

```
Input: Packet P
switch Ethertype do
    case generated seed packet do
        Ethertype ← 'pull request';
        Set the buffer usage field;
    case pull request do
        if local.buffer_usage > P.buffer_usage then
            Ethertype ← 'telemetry data';
            Pop telemetry from buffer and append it to P;
        else
            Set the buffer usage field;
    case telemetry packet do
        Store the payload in the local buffer;
        Drop the packet;
```

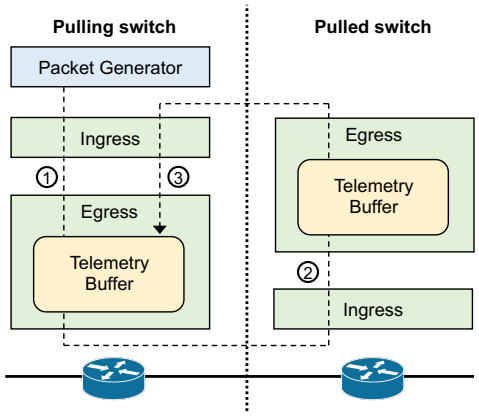


Figure 5: A potential lifecycle of a InvisiFlow packet. Circled numbers correspond to the steps in §4.1.

and then multicasts those seed packets to all egress ports, with additional details in §5 and Appendix A. InvisiFlow uses this mechanism to initiate pull requests, from which the rest of the InvisiFlow protocol stems.

Another building block is InvisiFlow's telemetry buffers in the ingress and egress pipeline of every switch. These telemetry buffers are distinct from the traditional packet buffers used by user traffic and managed by the ASIC; instead, SRAM arrays are easier for data plane programs to manipulate. Note that while both types of buffers are used to store telemetry data, the egress buffer(s) of the pipeline(s) closest to the collector(s) serves as the primary storage location. To see why, consider a system that creates and appends telemetry data to packets in the ingress. Packets must be assigned an egress port in the ingress pipeline, but the fine-grained usage of any given egress port can be difficult to ascertain, especially if multiple pipelines may be sending to the same egress port. Storing data in the egress instead enables late binding—whenever the egress arbiter can schedule a low-priority telemetry packet (destined for any port in the same pipeline), the next chunk of data can be appended immediately before it is placed on the wire. The ingress buffers are only used as a temporary

holding area for data that will eventually be transferred to the egress pipeline.

Given these components, Figure 5 shows one potential lifecycle of an InvisiFlow packet. For this packet:

1. *The pulling switch generates/sends pull requests:* As shown in Table 1, every switch in the network will periodically generate seed packets and multicast them to all neighbors. Before each copy leaves the egress pipeline, it will be converted to a pull request with the current telemetry buffer usage as shown in line 2-4 of Algorithm 1.
2. *The pulled switch replies to the pull requests:* As shown in Table 1, after receiving a pull request, the neighbor switch sets the egress port equal to the ingress port to return telemetry data. In the egress pipeline, the switch compares the telemetry buffer size recorded in the pull request with its local buffer size as shown in line 5-10 of Algorithm 1. If the local buffer size is larger, the switch pops telemetry data from the buffer and sends it back. Otherwise, the switch reflects the pull request back to the sender as its own.
3. *The pulling switch processes the telemetry data:* There are two possible replies from the pulled switch: (1) If the reply is still a pull request, the procedure remains the same as described above. (2) If the reply is a telemetry packet, we store the packet in the telemetry buffer at the egress pipeline as shown in line 11-13 of Algorithm 1.

Example. Suppose a seed packet, denoted as p, is initially generated by $S1$ and multicast to all neighbors in the ingress pipeline. In the egress pipeline of $S1$, p is converted to a pull request, p' with its buffer usage field set to the current telemetry buffer usage (*e.g.*, 10). Upon receiving the pull packet p', the neighbor switch, $S2$, routes it back out the same port. In the egress pipeline of $S2$, if the buffer usage (*e.g.*, 20) of $S2$ is larger than the buffer usage field in the pull request p', $S2$ pops telemetry data from the buffer and sends it back to $S1$. Finally, $S1$ forwards the telemetry packet through free ports and stores it in the buffer at the egress pipeline.

There are several important aspects and possible variations of the above procedure.

Egress port choice and its impact on packet buffers. First, we note that the choice of egress port of Table 1 and their configurations in each branch of Algorithm 1 is deliberate. Pull requests are valid for a single port and are reflected back for egress handling. Because the seed and pulls are regenerated periodically, the loss of these requests is acceptable, and the ASIC-level shared buffers for these packets can be shallow (*i.e.*, enough for only a handful of outstanding requests). Packets containing telemetry data should not be dropped as readily, so InvisiFlow only sends them if capacity is guaranteed. It does so by using free ports (*e.g.*, recirculation ports) that are not typically used by user traffic, as shown in Table 1. The advantage of using these ports is that it leverages higher aggregate bandwidth within the switch and reduces the usage of non-evictable packet buffers in the traffic manager.

Adding a slope to prevent zero congestion gradient. When the network contains only a limited number of telemetry packets, there are potential issues where the congestion gradient remains consistently at zero. Consider the scenario with only one telemetry packet, p, sitting in switch $S1$. A neighbor, $S2$, will pull p because it has lower telemetry buffer usage; however, $S1$ now has less telemetry buffer usage than $S2$, so there is a chance that p can oscillate between these switches.

To address the problem, we add a 'slope' to the congestion gradient. Suppose the buffer usage of switch $S1$ is u_{S1}. Switch $S1$ can pull telemetry data from switch $S2$ if and only if $u_{S1} + \delta_{S1,S2} < u_{S2}$, where $\delta_{S1,S2}$ indicates the slope. In InvisiFlow, we set δ based on the distance to the collector. If switch $S1$ is closer to the collector than switch $S2$ as defined by the routing protocol, then $\delta_{S1,S2} = 0$. Otherwise, $\delta_{S1,S2} = 1$. This approach improves the directness of paths and ensures that even when there is just one telemetry packet, it can efficiently flow toward the collector without getting stuck.

Adjust the pulling rate based on local buffer usage. Adjusting the pulling rate is crucial to avoid pulling excessive telemetry data, which could result in buffer overflow. In the egress pipeline, InvisiFlow randomly drops generated seed packets, with the probability of dropping packets equal to the buffer usage. Moreover, if a switch's buffer usage exceeds a certain threshold (*e.g.*, > 95%), it will revert to blindly pushing out telemetry packets to neighboring switches, as in prior work [54]. While push-based sending carries risks for telemetry packets, extreme buffer usage suggests they would likely have been dropped otherwise. We find that this approach effectively prevents the loss of telemetry data, particularly in scenarios with asymmetric bandwidth usage.

4.2 Server Operation

Two types of servers participate in InvisiFlow: those that collect telemetry data and those that serve as temporary storage.

Collectors. To get telemetry data from switches, collectors simply send pull requests to the network in the same manner as switches. We set the buffer size recorded in the pull request to 0. Based on §4.1, neighbor switches will send any available telemetry packets in response to these pull requests. Note that it also works if there are multiple collectors, and the telemetry data will flow to any one of the collectors.

Note that collectors also have the option to *not* send pull requests if, for instance, they lack sufficient CPU resources or storage bandwidth to keep up with the current inflow. Alternatively, collectors can offload the collection tasks to the last-hop switch as in prior works [43], *i.e.*, the last-hop switch is responsible for generating pull requests and directly writes the telemetry data to servers' memory throughput Remote Direct Memory Access (RDMA).

Available servers. To further minimize telemetry data loss, InvisiFlow can optionally explore available memory/storage in servers to store telemetry packets. We assume that the

Resource	Baseline	Marginal Increase
Gateway	11.46%	2.08%
VLIW Actions	4.69%	0.78%
Hash Bit	5.07%	0.46%
Exact Match Input xbar	3.58%	0.39%
Meter ALU	14.58%	0.00%
SRAM	1.77%	0.00%
Stages	7	0

Table 2: Additional H/W resources used by InvisiFlow. Baseline is the low-priority UDP using OrbWeaver [21] to generate telemetry packets.

memory/storage capacity in these available servers is much larger than that of switches. Similar to collectors, available servers join InvisiFlow by sending pull requests. The difference is that we set the buffer usage in the pull request to a constant value α (*e.g.*, 50%). These available servers can pull telemetry packets from their neighboring switches if the buffer usage of the switch is larger than α. Otherwise, the neighbor switches pull telemetry packets from these available servers. The setting of α determines the usage rate of the memory/storage in available servers and the possibility of data loss as shown in §6.2. A larger α may lead to a larger possibility of data loss but a lower usage rate of the memory/storage in available servers, and vice versa. The available servers can also dynamically adjust α based on their available memory and leave enough space for user applications.

5 Implementation

We implement an InvisiFlow prototype on a testbed with two Wedge100BF-32X switches. Servers are equipped with a Mellanox ConnectX-5 100 Gbps NIC, dual-socket AMD EPYC 7313 16-Core Processor, and 64 GB memory.

InvisiFlow servers. Per §4.2, each InvisiFlow-server (*i.e.*, a collector or a free server) sends out pull requests if there is available space and processes pull requests or telemetry packets from the network. In our testbed, we implement the collectors and free servers using DPDK (v21.11).

InvisiFlow switches. We implement InvisiFlow's switch component in P4-16. Our implementation leverages the built-in packet generation and strict prioritization features present in today's switches. In total, the data plane implementation consists of around 600 lines of P4 code. Our implementation scales to today's multi-pipeline switching architectures.

Egress pipeline register buffer for telemetry packets: Similar to PayloadPark [55], we use an array of registers in the egress pipeline to store telemetry packets. Due to the constraints on the number of stages and the number of bytes read/written per stage, the size of telemetry packets should be small (*e.g.*, <160 B) as in [55]. This amount is sufficient for most applications [5, 27, 28] and even allows for multiple telemetry records within a single packet in many cases. Larger telemetry messages can be handled with per-hop segmentation of telemetry packets. To improve the goodput of telemetry data, we omit the IP and UDP header and include only Ethernet. Different *EtherType*s distinguish different types of packets, *e.g.*, user packets, pull requests, or telemetry packets.

Switch-local teleport of telemetry data: Besides front-panel ports, modern switches also have per-pipeline internal ports for pktgen traffic and recirculation. InvisiFlow consumes the bandwidth of one of these ports to move the telemetry data to the egress pipeline, as discussed in §4.1. Note that InvisiFlow does not recirculate any telemetry packets nor suffer from recirculation overhead. The egress pipeline drops telemetry packets from the ingress pipeline immediately after storing the telemetry data in the egress pipeline register.

Hardware resource usage: Table 2 summarizes the key resources required by our prototype. Compared with low-priority UDP, which uses OrbWeaver [21] to generate low-priority telemetry packets, the additional resource used by InvisiFlow is small: 2.08% additional gateways and 0.78% additional VLIW actions to process the telemetry packets. The ALU and SRAM are mainly used to store and read/write telemetry data in the egress pipeline.

6 Evaluation

We evaluate InvisiFlow using a combination of testbed experiments and large-scale ns-3 simulations [56]. Our evaluation focuses on the following three key questions.

- Can InvisiFlow minimize the accuracy degradation on the telemetry application (R1)?
- Can InvisiFlow simultaneously minimize the overhead of transmitting telemetry data (R2)?
- Is InvisiFlow robust to different settings and workloads?

6.1 Simulation Methodology

Topology and workload. We simulate a FatTree [45] topology with 9 core switches and 4 pods that, combined, have 12 aggregation switches, 12 top-of-rack (ToR) switches and 144 servers. All links have a capacity of 100 Gbps, leading to a 4:1 oversubscription ratio similar to prior work [57–59]. The propagation delay for all links is 1 µs, resulting in a maximum base RTT of 12 µs. Switch buffer sizes are set proportionally to the bandwidth-buffer ratio of Intel Tofino switches [60], following prior research [58, 59]. Specifically, each switch has approximately 2 MB of buffer for up to 600 Gbps of user traffic and 170 KB for up to 50 Gbps of telemetry traffic[2]. These buffers are shared by all ports on the same switch and are fully isolated to prevent any impact on user traffic. For DCTCP, we set $K = 65$ based on [59, 61]. User traffic is load-balanced using ECMP over the 5-tuple.

Experiments are performed with one of two workloads: a Hadoop workload (FB_Hadoop) and a machine-learning (ML) workload. Details of the workloads can be found in Appendix B. Unless otherwise specwified, FB_Hadoop is the default setting.

[2] In experiments, the total telemetry data generated is generally <30 Gbps.

Stressed scenarios. In addition to the default setting, we introduce two other scenarios to test InvisiFlow's robustness.

- *Load imbalance:* We simulate a load imbalance scenario caused by a faulty configuration, where ECMP is configured to hash based only on the destination IP, which causes packets to a single destination to use only one path, leading to high load imbalance, as discussed in prior work [62].
- *Asymmetric topology:* We also stress the collector under an asymmetric topology with failed fabric links [20, 63]. We reduce the capacity of two links to 50 Gbps. These links include a Core-Aggregation link and an aggregation-ToR link, both toward the collector.

Telemetry sources, channels, and sinks. Next, we describe the evaluated configurations of the three components of the telemetry frameworks depicted in §2.1.

Sources: Four telemetry applications run concurrently between 20% and 80% of the simulation time.

- *Path tracing (NetSeer [5]):* Following [5], each switch maintains a record of user flows passing through it. When encountering a new flow, the switch stores the tuple (switch ID, flow ID, TTL value) in its buffer. The collector uses the TTL value to calculate the hop ID of each node and reconstruct the entire path of each flow.
- *Load imbalance detection (NetFlow [36]):* Following [36], every switch records the number of bytes passing through each egress port in the data plane. The switch then stores the tuple (switch ID, port ID, time, # of bytes) in the buffer and reports the results (by default) every 5 µs. The collector aggregates the byte counts to calculate each switch's load. It measures the load imbalance rate as the maximum load relative to the minimum load among switches.
- *Per-flow accounting (ApproSync [31]):* We implement ApproSync [31] with Count-Min Sketch (CMS) [32] to track the size per-flow. In the CMS, we use 3 arrays of 2^{17} counters per array. ApproSync sends sketches to the collector by checking the divergence of each counter in the sketch. Divergence is quantified using relative error, with a threshold based on an expected bandwidth of 10 Gbps.
- *Packet drop notification (NetSeer [5]):* Similar to [5], when a user packet drop occurs, the switch stores the tuple (switch ID, flow ID) in the buffer.

Channels: We compare InvisiFlow (IF) with three communication substrates for telemetry data:

- *Default-priority UDP:* Prior telemetry systems like NetFlow [36], sFlow [42], and NetSight [29] often use default-priority UDP to send telemetry data. We implement default-priority UDP in postcard mode [29]: When there is telemetry data to be sent, the switch generates a telemetry packet using mirroring and sends it to the collector. Telemetry packets are assigned the same priority as user traffic. If multiple shortest paths are available to the collector, we use packet spraying [51] for load balancing.
- *Low-priority UDP:* Some telemetry systems like Everflow [28] and Planck [27] assign low-priority to telemetry packets to avoid impacting user traffic. To implement low-priority UDP, we use OrbWeaver [21] to generate low-priority telemetry packets. These packets are forwarded to the next hop when there is available bandwidth. As before, we use packet spraying if there are multiple shortest paths.
- *Pull-based transmission:* As an ablation study, we include pull-based transmission without using congestion gradients. Each switch generates pull requests only to switches farther from the collector, so that telemetry packets follow the shortest paths. A switch pulls only when its telemetry buffer usage is below 50%. We reserve some buffer capacity to prevent overflow, as there could be locally generated or late telemetry data due to low-priority transmission.

Sinks: In our experiments, one of the 144 servers is always reserved as a collector server. When additional sinks are required, they are taken from other clusters. We assume that every switch is pre-configured with their addresses.

6.2 Telemetry Application Performance

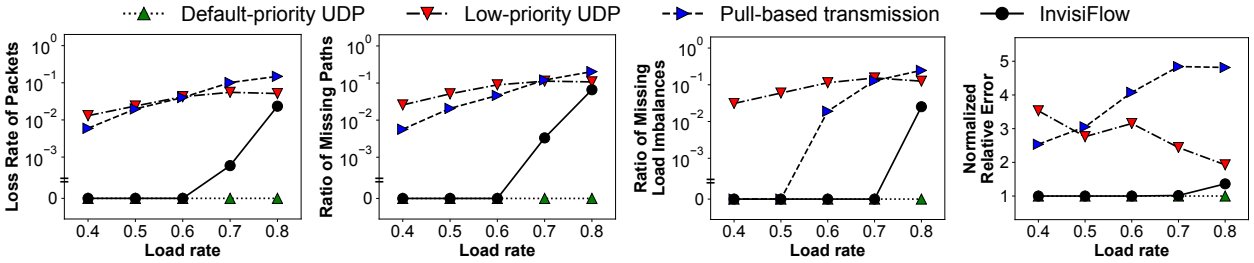
This section compares the impact of InvisiFlow to alternative communication substrates on telemetry applications, highlighting InvisiFlow's ability to minimize accuracy degradation, particularly for low-priority transmissions.

Performance metrics. We consider the following metrics:

- *Ratio of missing paths:* Following [3], the collector reports all paths it receives. If the collector fails to capture one or more switch IDs along a path, we mark the path missed.
- *Ratio of missing load imbalance events:* We calculate the ratio of missing events in core switches for load imbalance detection as in [10]. A missed event occurs when the collector cannot identify a period where the imbalance rate at the core switches exceeds 2.
- *Normalized relative error of the flow size:* As in [31], all core switches estimate the number of packets per flow using sketches before sending them to the collector. To quantify the impact of telemetry loss, we calculate the average relative error of those estimates normalized to the case without loss (i.e., we exclude errors introduced by ApproSync and the Count-Min sketch). The errors for heavy hitters (*i.e.*, $> 10^3$ packets) are shown in §C.

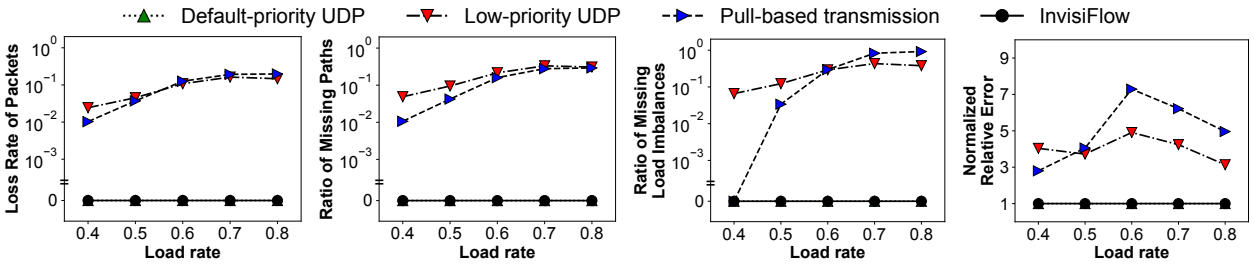
Since user packet drops are rare in our setup, we do not present the loss rate for packet drop notifications. Ground truth results are captured with the help of extensive out-of-band logging in the simulator.

Default setting (Figure 6). Figure 6 illustrates the performance of different telemetry applications under varying load rates. InvisiFlow maintains a zero loss rate for telemetry packets when the load rate is below 70%. Consequently, the ratio of missing paths and load imbalances are also 0, and the normalized relative error is 1 below this threshold. However, InvisiFlow has to drop telemetry packets when the load ex-



(a) Telemetry packet loss (b) Path loss for NetSeer (c) Imbalance loss for NetFlow (d) Flow error for ApproSync

Figure 6: Default setting. InvisiFlow maintains a zero loss rate when the load rate is below 70%.



(a) Telemetry packet loss (b) Path loss for NetSeer (c) Imbalance loss for NetFlow (d) Flow error for ApproSync

Figure 7: Load imbalance. InvisiFlow maintains a zero telemetry packet drop rate even under an 80% load, whereas the loss rates of low-priority UDP and pull-based transmission increase compared to the default setting.

ceeds 70% due to limited available bandwidth for low-priority transmission, which could be smaller than the generation rate of the telemetry packets. The performance of pull-based transmissions deteriorates under heavy loads compared to low-priority UDP because it cannot fully utilize the buffer and bandwidth when the available bandwidth is insufficient for the telemetry data throughput. Default-priority UDP does not drop packets even under 80% load, but increases the FCT by approximately 26.4%, as discussed in §6.3.

At a 70% load, InvisiFlow's ratio of missing paths is approximately 33.8× and 36.3× lower than that of low-priority UDP and pull-based baselines, respectively. The gap between them narrows under an 80% load, as there is limited room to reduce loss due to insufficient available bandwidth for low-priority packets at such a high load rate. However, InvisiFlow still shows a 1.6× and 3.1× improvement over low-priority UDP and pull-based baselines, respectively. Regarding the relative error of all flows, InvisiFlow at a 70% load exhibits a reduction of approximately 2.4× and 4.8× compared to low-priority UDP and pull-based baselines, respectively. Normalized relative errors may decrease with higher load rates due to increased error in the original Count-Min sketch caused by higher traffic volumes.

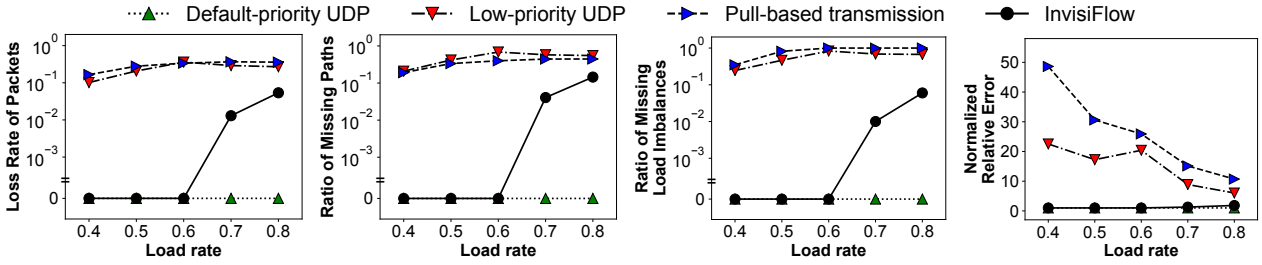
Load imbalance (Figure 7). In this scenario, InvisiFlow maintains a zero telemetry loss rate even under an 80% load, whereas the loss rates of low-priority UDP and pull-based transmission increase compared to the default setting. This is because the load imbalance reduces the throughput of user traffic, potentially leaving more available bandwidth for telemetry data. However, low-priority UDP and pull-based baselines fail to fully utilize this additional bandwidth as they remain constrained to the shortest path.

At a 70% load, the ratio of missing paths for InvisiFlow is 0, while it is 33.2% and 27.9% for low-priority UDP and pull-based baselines, respectively. Furthermore, the normalized relative error of all flows for InvisiFlow is 1, while it is 4.2 and 6.2 for low-priority UDP and pull-based, respectively.

Asymmetric topology (Figure 8). In an asymmetric topology, the loss rates of low-priority UDP and pull-based transmission experience significant increases. In contrast, InvisiFlow only drops telemetry packets when the load rate exceeds 70%. Specifically, the ratio of missing paths for low-priority UDP often exceeds 40%, and for the pull-based baseline, it often exceeds 30%. Furthermore, the normalized relative error of all flows for low-priority UDP and pull-based baselines can reach up to 22.5 and 48.6, respectively.

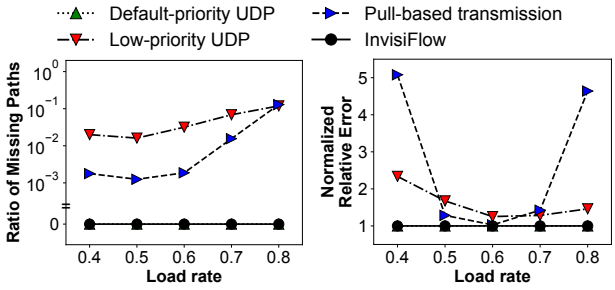
ML workloads (Figure 9). In ML workloads, InvisiFlow has a zero telemetry packet drop rate even under an 80% load, while low-priority UDP and pull-based transmission lose approximately 11.9% and 12.9% of paths at an 80% load, respectively. Furthermore, at a 70% load, the relative error of all flows for InvisiFlow is approximately 1.5× and 4.6× lower than that of low-priority UDP and pull-based, respectively.

Different buffer sizes (Figure 10). In addition to the default buffer size of 170 KB, we experiment with different buffer sizes under the default setting. We observe that InvisiFlow achieves a 0% loss rate when the buffer size exceeds 400 KB, because the congestion gradient-based transmission efficiently utilizes all available telemetry buffers across the network. In contrast, low-priority UDP and pull-based transmission lose around 5.8% and 9.8% of paths, respectively, even with a 500 KB buffer. With a 500 KB buffer,



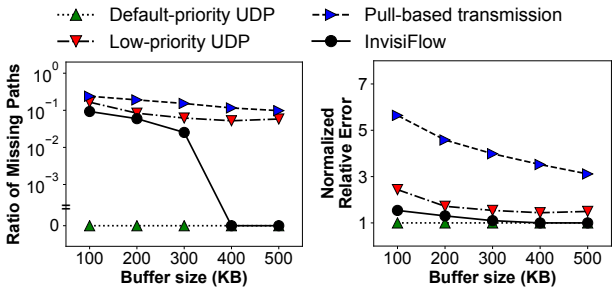
(a) Telemetry packet loss (b) Path loss for NetSeer (c) Imbalance loss for NetFlow (d) Flow error for ApproSync

Figure 8: Asymmetric topology. InvisiFlow only drops telemetry packets when the load rate exceeds 0.7. The ratio of missing paths for low-priority UDP often exceeds 40%, and for the pull-based baseline, it often exceeds 30%.



(a) Path loss for NetSeer (b) Flow error for ApproSync

Figure 9: ML workload (Default setting).



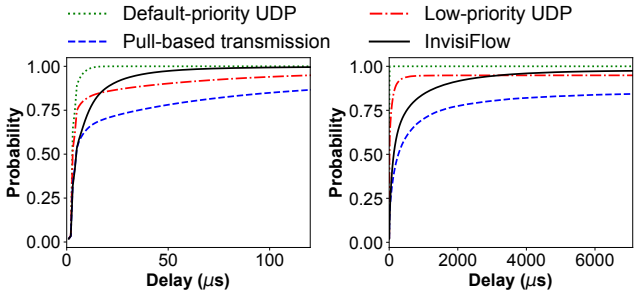
(a) Path loss for NetSeer (b) Flow error for ApproSync

Figure 10: Different buffer sizes (80% load).

the relative error of all flows for InvisiFlow is approximately 1.5× and 3.1× lower than that of low-priority UDP and pull-based baselines, respectively.

CDF for delay of telemetry packets (Figure 11). Figure 11 illustrates the end-to-end delay of the telemetry packets. We calculate the delay by subtracting the timestamp at which the collector receives the packet from the timestamp at which the telemetry packet is generated. In case of packet loss, we consider the delay infinite. Notably, InvisiFlow significantly improves tail latency compared to other low-priority transmissions. Under a 40% load, the 99% latency of InvisiFlow is approximately 80 μs, which is around 3.4× and 10.9× lower than that of low-priority UDP and pull-based baselines, respectively. Under an 80% load, InvisiFlow increases the delay to avoid both telemetry packet loss and impact on user traffic, with a 99% latency of around 4.9 ms.

Using multiple servers (Table 3). Using multiple collectors or additional available servers as temporary storage can further reduce telemetry data loss. When using two collectors,



(a) 40% load rate. (b) 80% load rate.

Figure 11: Delay of telemetry packets (Default setting).

telemetry data can be directed to any available collector, enabling a 0% loss rate. When using one available server, the threshold of the buffer usage determines the storage usage in available servers and the data loss rate. A higher threshold increases the risk of data loss but reduces the storage usage in available servers, and vice versa. Setting the threshold to 25% allows zero loss rate while utilizing up to 13.63 MB of storage for telemetry packets under the default setting. Unlike using multiple collectors where telemetry data can be distributed, using available servers results in all telemetry data being sent to the single designated server without further aggregation.

6.3 Impact on User Traffic

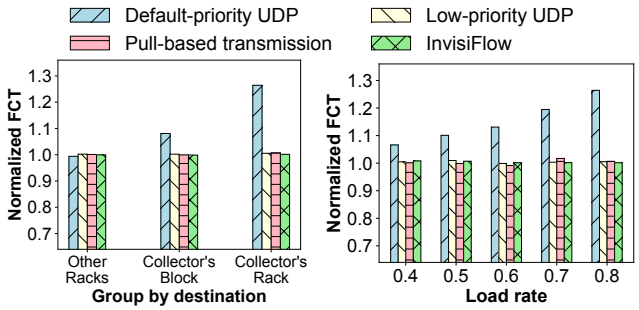
In this section, we compare InvisiFlow with other communication substrates and show that InvisiFlow achieves near-zero overhead to user traffic. The experimental setup is consistent with that in §6.2.

Metric. We evaluate the flow completion time (FCT) of user traffic initiated between 20% and 80% of the simulation time (*i.e.*, when the telemetry applications are running). The FCT is normalized to the case where no telemetry data is generated.

Default setting (Figure 12). In Figure 12a, we group the FCT by the destination of the user traffic and calculate the ratio. We find that user traffic destined for the rack of the collector is primarily influenced by the telemetry data under the default setting. Assigning telemetry data default priority results in an FCT increase of ∼26.4% under an 80% load. In contrast, InvisiFlow and low-priority baselines exhibit minimal overhead, often <1%. Figure 12b shows the ratio under different load rates, showing that the overhead of InvisiFlow and

Default Setting	Telem. Loss	Peak Avail. Util
1 Collector	2.34%	N/A
+ 1 Collector	0.00%	N/A
+ 1 Available Server *Threshold: 50%	1.94%	2.27 MB
+ 1 Available Server *Threshold: 25%	0.00%	13.63 MB
Asymmetric topology	**Telem. Loss**	**Peak Avail. Util**
1 Collector	5.38%	N/A
+ 1 Collector	0.00%	N/A
+ 1 Available Server *Threshold: 50%	1.79%	10.92 MB
+ 1 Available Server *Threshold: 25%	0.00%	25.83 MB

Table 3: Telemetry packet loss rate and peak storage utilization of available servers when using multiple servers for telemetry data collection. We evaluate the default and asymmetric topologies under 80% load.



(a) 80% load rate. **(b) In Collector's rack.**

Figure 12: Normalized FCT (Default setting).

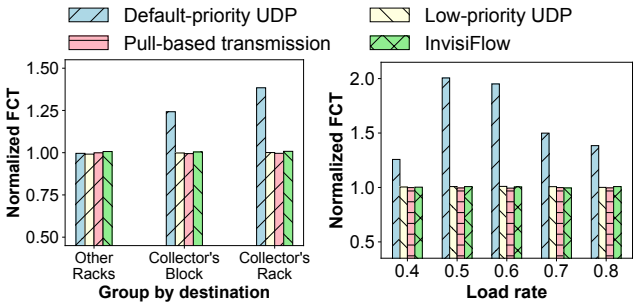
low-priority remains minimal, while the overhead of default-priority UDP increases with higher load rates.

Asymmetric topology (Figure 13). As depicted in Figure 13a, user traffic not only to the rack but also to the block containing the collector is significantly influenced by telemetry data under the asymmetric topology. For default-priority UDP, the FCT of user traffic to the rack increases by ∼38.4%, and the block by ∼25.2%. In contrast, InvisiFlow and low-priority baselines maintain near-zero overhead (<1%). Figure 13b illustrates that under a 50% load, the FCT of user traffic is doubled. The normalized FCT may decrease with increasing load rates due to user traffic causing more congestion.

6.4 Testbed Experiments

Topology (Figure 14a). We emulate a 2-tier leaf-spine topology in our testbed. With two physical switches, similar to [11], we divide one 128-port physical switch into four virtual leaf switches and the other into two virtual spine switches. The virtual switches are connected with 10 Gbps links. In the data plane, every logical switch has its isolated buffer of telemetry packets and runs InvisiFlow independently.

Setup. We emulate the case that a link to the collector is fully



(a) 80% load rate. **(b) In Collector's rack.**

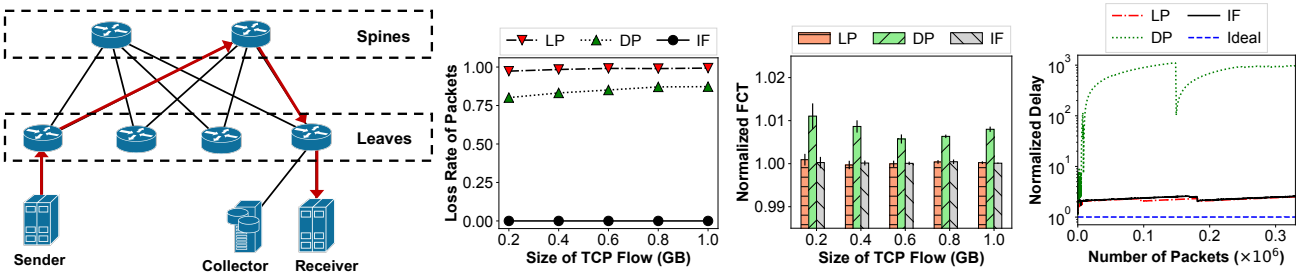
Figure 13: Normalized FCT (Asymmetric topology).

occupied by user traffic. Specifically, as shown in Figure 14a, we build a TCP connection between two servers and send traffic. Then, we measure the statistics in the right spine switch, which is the main bottleneck for telemetry data transmission. We use the load imbalance detection as the application and compare InvisiFlow (IF) with the low-priority UDP (LP) and the default-priority UDP (DP) as above.

Loss rate of telemetry packets (Figure 14b). In Figure 14b, we measure the loss rate of telemetry data in the right spine switch. We can find that the loss rate for low-priority UDP is higher than 97%. Because the only shortest path to the collector is fully occupied by the user traffic, low-priority UDP can hardly send telemetry data. In contrast, the loss rate of InvisiFlow is always 0 because InvisiFlow can find other available paths to send telemetry data. Note that the loss rate of default-priority UDP is also higher than 80% and increases over the size of TCP flows. It may be due to the switch architecture, *i.e.*, mirroring could fail when the corresponding egress port is fully utilized by user traffic.

Impact on user traffic (Figures 14c and 14d). In Figure 14c, we measure the ratio between the flow completion time (FCT) of user traffic with telemetry data against that without any telemetry. The overhead incurred by low-priority UDP and InvisiFlow is negligible (<0.1%), while the overhead of the default-priority UDP is higher (≈0.8%). The overhead of the default-priority UDP is mitigated because the switch already drops over 80% of telemetry packets when user traffic fully utilizes the link, as illustrated in Figure 14b.

In Figure 14d, we measure the delay of user packets within the right spine switch over time. Specifically, we calculate the switch delay by subtracting the timestamp at the egress pipeline from the timestamp at the ingress pipeline. The ideal delay in the figure represents the delay without any telemetry data. We can find that the default-priority UDP will gradually increase the delay of user packets over time, leading to the queue becoming full and subsequent packet drops. The delay for default-priority UDP could be up to $10^3\times$ larger than the ideal delay. Conversely, the delay of low-priority UDP and InvisiFlow is similar, which keeps around $2\times$ as the ideal delay. In other words, the delay of default-priority UDP could be up to $500\times$ larger than that of InvisiFlow. The reason for the additional delay ($1\times$ more normalized delay) in low-



(a) Testbed topology. (b) Telemetry packet loss. (c) Flow completion time. (d) Queuing delay.

Figure 14: Testbed experiments.

priority UDP and InvisiFlow might be that the switch will start strict prioritization only after there is a user packet waiting in the queue. However, it will not influence the overall flow completion time as shown in Figure 14c.

7 Discussion

Limitations of InvisiFlow. We note that InvisiFlow, in its effort to minimize the impact on user traffic, may result in drops and arbitrary delays of telemetry packets, particularly if all ports of the switch are fully utilized by user traffic. Under an extreme case where none of the switches have available bandwidth, all low-priority transmissions, including InvisiFlow, would have to drop packets. However, such scenarios are rare in modern high-speed networks. As demonstrated in §6.2, even under 80% load, InvisiFlow can reduce missing paths by around 1.63× and 3.08× compared to low-priority UDP and pull-based baselines, respectively. If a portion of telemetry traffic is essential, it can always fall back on standard transport mechanisms. InvisiFlow may also struggle with networks that need a large number of distinct collector types, as it requires the use of a separate buffer allocation for every type. Switching ASICs with more flexible memory access patterns may alleviate this limitation [64, 65].

Low-latency transmission. In some cases, telemetry reports may only be valuable if collected in a timely manner. Heretofore, InvisiFlow was presented as a framework that minimizes loss but does not bound the latency of reports. To support time-sensitive reports, we can adapt InvisiFlow by restricting sending to only hops that are strictly closer to the eventual destination. Packets would have fewer options for forwarding, but prior work on progressive routing has shown that adaptive routing policies like InvisiFlow's are still very effective [66].

Generality to other applications. InvisiFlow's abstraction to exploit network-wide under-utilized bandwidth is also applicable to other application amenable to low-priority transmission. As discussed above, InvisiFlow cannot (1) guarantee the delivery and (2) support too many distinct destinations. To address these problems, we can (1) ask the above application to handle delivery or add a layer above InvisiFlow to guarantee delivery as TCP and (2) group multiple destinations into one for simplicity, similar to how we group multiple IP addresses into an IP prefix. We leave the extension to user applications as future work.

8 Related Work

Applications of congestion gradients. Congestion gradients (*i.e.*, max-weight scheduling) have mainly been studied in a theoretical context [49, 50]. The mathematical optimality properties of congestion gradients have motivated experimental demonstrations of its use on wireless sensor networks [67, 68]. Some work [69] also applies it to urban traffic networks. However, these works focus on entirely different scenarios, so they need to consider many different factors (*e.g.*, lossy links in wireless networks) and implement the systems on different platforms (*e.g.*, CPU in sensors).

Per-packet multi-path routing. A commonly studied per-packet multi-path routing technique is packet spraying [51]. However, as shown in §6.2, the loss rate of low-priority UDP remains high even when utilizing packet spraying. This is because packet spraying assumes symmetry topology, while the available bandwidth for low-priority transmission is often asymmetric and varies largely over time. Furthermore, there are also some other per-packet multi-path routing protocols proposed [70, 71], primarily for wireless networks. As discussed above, they focus on entirely different scenarios and are mainly designed to achieve different objectives (*e.g.*, energy efficiency, security, and reliability).

Explore external memory in servers. Prior work has suggested system architectures that allow switches to utilize external memory on servers [52, 53]. The switches can directly access DRAM on servers via RDMA. However, similar to default-priority UDP, such a way is not friendly to achieve near-zero overhead on user traffic. In addition, [52, 53] need to pre-define the memory allocated in servers, while InvisiFlow can dynamically adjust the size of the memory needed.

9 Conclusion

Existing transmission methods for telemetry data often have to choose between overhead to user traffic and sustainable throughput of telemetry. To achieve both, we introduce InvisiFlow, a low-priority communication substrate utilizing opportunistic sending and congestion gradients. Our experimental results show that InvisiFlow can support high sustainable throughput with minimal impact on user traffic across multiple different applications and settings.

Acknowledgments

We thank the anonymous reviewers for their thorough comments and feedback. This work was funded in part by NSF grant CNS-1845749. The work was also partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

References

[1] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, David A. Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM 2015*, pages 139–152. ACM, 2015.

[2] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach (Guoyi) Chen, and Greg Mirsky. AM-PM: efficient network telemetry using alternate marking. *IEEE Netw.*, 33(4):155–161, 2019.

[3] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *OSDI 2016*, pages 233–248. USENIX Association, 2016.

[4] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Cherrypick: tracing packet trajectory in software-defined datacenter networks. In *SOSR '15*. ACM, 2015.

[5] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *SIGCOMM '20*, pages 76–89. ACM, 2020.

[6] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: scalable load balancing using programmable data planes. In *SOSR 2016*, page 10. ACM, 2016.

[7] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *NSDI '18)*, pages 1–16, 2018.

[8] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *SIGCOMM '20*, page 296–309. ACM, 2020.

[9] Yinda Zhang, Peiqing Chen, and Zaoxing Liu. Octosketch: Enabling real-time, continuous network monitoring over multiple cores. In *NSDI 2024*. USENIX Association, 2024.

[10] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *IMC '17*, page 78–85. Association for Computing Machinery, 2017.

[11] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *SIGCOMM 2018*, pages 402–416. ACM, 2018.

[12] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *SIGCOMM*, pages 225–238, 2017.

[13] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *ACM SIGCOMM 2021 Conference*, pages 207–222. ACM, 2021.

[14] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *SIGCOMM 2018*, pages 561–575. ACM, 2018.

[15] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. Heterosketch: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *NSDI 2022*, pages 719–741. USENIX Association, 2022.

[16] Haifeng Sun, Qun Huang, Jinbo Sun, Wei Wang, Jiaheng Li, Fuliang Li, Yungang Bao, Xin Yao, and Gong Zhang. Autosketch: Automatic sketch-oriented compiler for query-driven network telemetry. In *NSDI 2024*. USENIX Association, 2024.

[17] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *SIGCOMM 2019*, pages 44–58. ACM, 2019.

[18] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, Lawrence J Wobker, et al. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, pages 1–2, 2015.

[19] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: probabilistic in-band network telemetry. In *SIGCOMM '20*, pages 662–680. ACM, 2020.

[20] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E. Anderson. F10: A fault-tolerant engineered network. In *NSDI 2013*, pages 399–412. USENIX Association, 2013.

[21] Liangcheng Yu, John Sonchack, and Vincent Liu. Orbweaver: Using IDLE cycles in programmable networks for opportunistic coordination. In *NSDI 2022*, pages 1195–1212. USENIX Association, 2022.

[22] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 516–529, 2022.

[23] Snmp. https://datatracker.ietf.org/doc/html/rfc1157.

[24] Framework for in-situ flow information telemetry. https://datatracker.ietf.org/doc/draft-song-opsawg-ifit-framework/.

[25] Inband flow analyzer. https://datatracker.ietf.org/doc/html/draft-kumar-ippm-ifa.

[26] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI 2021*, pages 991–1010. USENIX Association, 2021.

[27] Jeff Rasley, Brent E. Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John B. Carter, and Rodrigo Fonseca. Planck: millisecond-scale monitoring and control for commodity networks. In *SIGCOMM'14*, pages 407–418. ACM, 2014.

[28] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert G. Greenberg, Guohan Lu, Ratul Mahajan, David A. Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM 2015,*, pages 479–491. ACM, 2015.

[29] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI 2014*, pages 71–85. USENIX Association, 2014.

[30] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *NSDI 2019*, pages 207–220. USENIX Association, 2019.

[31] Xiang Chen, Qun Huang, Dong Zhang, Haifeng Zhou, and Chunming Wu. Approsync: Approximate state synchronization for programmable networks. In *ICNP 2020*, pages 1–12. IEEE, 2020.

[32] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 2005.

[33] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *SIGCOMM '20*, pages 226–239. ACM, 2020.

[34] Nguyen Van Tu, Jonghwan Hyun, Ga Yeon Kim, Jae-Hyoung Yoo, and James Won-Ki Hong. Intcollector: A high-performance collector for in-band network telemetry. In *CNSM 2018*, pages 10–18. IEEE Computer Society, 2018.

[35] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *NSDI 2019*, pages 421–436. USENIX Association, 2019.

[36] Benoit Claise. Cisco systems netflow services export version 9. *RFC*, 3954:1–33, 2004.

[37] Generating netflow records. https://www.cisco.com/c/en/us/td/docs/dcn/nexus-data-broker/392/use-case/generate-netflow-records.html.

[38] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM 2016*, pages 101–114. ACM, 2016.

[39] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM 2018*, pages 576–590. ACM, 2018.

[40] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM 2015*, pages 123–137. ACM, 2015.

[41] Siyuan Sheng, Qun Huang, and Patrick P. C. Lee. Deltaint: Toward general in-band network telemetry with extremely low bandwidth overhead. In *ICNP 2021*, pages 1–11. IEEE, 2021.

[42] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. *RFC*, 3176:1–31, 2001.

[43] Jonatan Langlet, Ran Ben Basat, Gabriele Oliaro, Michael Mitzenmacher, Minlan Yu, and Gianni Antichi. Direct telemetry access. In *SIGCOMM 2023*, pages 832–849, 2023.

[44] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, nov 1984.

[45] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008*, pages 63–74. ACM, 2008.

[46] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *SIGCOMM 2013*, pages 3–14. ACM, 2013.

[47] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM 2013*, pages 15–26. ACM, 2013.

[48] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC 2010*, pages 267–280. ACM, 2010.

[49] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *29th IEEE Conference on Decision and Control*, pages 2130–2132. IEEE, 1990.

[50] Baruch Awerbuch and Frank Thomson Leighton. A simple local-control approximation algorithm for multicommodity flow. In *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993*, pages 459–468. IEEE Computer Society, 1993.

[51] Advait Abhay Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *INFOCOM 2013*, pages 2130–2138. IEEE, 2013.

[52] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: enabling state-intensive network functions on programmable switches. In *SIGCOMM '20*, pages 90–106. ACM, 2020.

[53] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostic, and Marco Chiesa. A high-speed stateful packet processing approach for tbps programmable switches. In *NSDI 2023*, pages 1237–1255. USENIX Association, 2023.

[54] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. DIBS: just-in-time congestion mitigation for data centers. In *EuroSys 2014*, pages 6:1–6:14. ACM, 2014.

[55] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo I. Seltzer. Parking packet payload with P4. In *CoNEXT '20*, pages 274–281. ACM, 2020.

[56] Network simulator 3. https://www.nsnam.org/.

[57] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa H. Ammar, Ellen W. Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and WAN traffic aggregates. In *SIGCOMM '20*, pages 735–749. ACM, 2020.

[58] Vamsi Addanki, Oliver Michel, and Stefan Schmid. Powertcp: Pushing the performance limits of datacenter networks. In *NSDI 2022*, pages 51–70. USENIX Association, 2022.

[59] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. ABM: active buffer management in datacenters. In *SIGCOMM '22*, pages 36–52. ACM, 2022.

[60] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/.

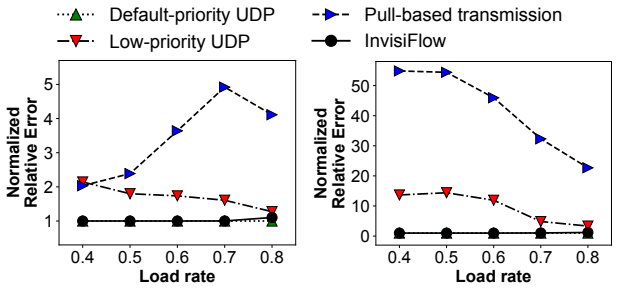
[61] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM 2010*, pages 63–74. ACM, 2010.

[62] Jie Li, Shi Zhou, and Vasileios Giotsas. Performance analysis of multipath BGP. In *INFOCOM Workshops 2021*, pages 1–6. IEEE, 2021.

[63] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *SIGCOMM'14*, pages 503–514. ACM, 2014.

[64] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. drmt: Disaggregated programmable switching. In *SIGCOMM 2017*, pages 1–14. ACM, 2017.

[65] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio: juniper networks' programmable chipset - for emerging in-network applications. In *SIGCOMM 2022*, pages 633–648. ACM, 2022.



(a) Default setting **(b) Asymmetric topology**

Figure 15: Flow error for heavy hitters.

[66] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., 2002.

[67] Avinash Sridharan, Scott Moeller, and Bhaskar Krishnamachari. Making distributed rate control using lyapunov drifts a reality in wireless sensor networks. In *WIOPT 2008*, pages 452–461. IEEE, 2008.

[68] Scott Moeller, Avinash Sridharan, Bhaskar Krishnamachari, and Omprakash Gnawali. Routing without routes: the backpressure collection protocol. In *IPSN 2010*, pages 279–290. ACM, 2010.

[69] Xingmin Wang, Yafeng Yin, Yiheng Feng, and Henry X Liu. Learning the max pressure control for urban traffic networks considering the phase switching loss. *Transportation Research Part C: Emerging Technologies*, 140:103670, 2022.

[70] Jenn-Yue Teo, Yajun Ha, and Chen-Khong Tham. Interference-minimized multipath routing with congestion control in wireless sensor network for high-rate streaming. *IEEE Trans. Mob. Comput.*, 7(9):1124–1137, 2008.

[71] Moufida Maimour. Maximally radio-disjoint multipath routing for wireless multimedia sensor networks. In *WMuNeP'07, 2008*, pages 26–31. ACM, 2008.

[72] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. Congestion control in machine learning clusters. In *HotNets 2022*, pages 235–242. ACM, 2022.

[73] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *OSDI 2020*, pages 463–479. USENIX Association, 2020.

A Background of OrbWeaver

OrbWeaver [21] is a framework designed for the opportunistic transmission of data within modern programmable networks. As detailed in its paper, OrbWeaver imposes a negligible impact on user traffic, the computational/state resources of participating switches, or their power consumption.

Instead of "measuring" the utilization of each port, OrbWeaver continually generates a stream of low-priority packets. Leveraging the behavior of priority queues, when there are no high-priority user packets in the queue, the low-priority packets produced by OrbWeaver can effectively utilize the available bandwidth and are transmitted to neighboring switches. Essentially, OrbWeaver fills these bandwidth gaps based on the characteristics of the priority queue.

OrbWeaver also supports the amplification (*i.e.*, multicast) of low-priority packets, as demonstrated in its paper. In InvisiFlow, we require only one multicast group to multicast pull requests to all neighboring switches. For Intel Tofino switches, the multicast group ID can be configured in the control plane, enabling the data plane to utilize this ID for multicasting packets.

B Workloads

Our evaluation workloads are configured as follows.

Hadoop workload (FB_Hadoop). By default, we adopt the FB_Hadoop workload [40]. In this workload, around 60% of the flow sizes are smaller than 1 KB, and the flows arrive according to a Poisson process as in previous work [17, 19].

Machine learning workload (ML). We emulate an ML workload to demonstrate InvisiFlow's performance under different traffic patterns and distributions. In this workload, all flows are large (1 MB each), and multiple ML jobs are running concurrently. We randomly select eight servers for each ML job and these servers periodically exchange data with each other, as described in [72, 73]. The duration of the period is calculated based on the load rate.

C Flow Error for Heavy Hitters

This section shows the normalized relative error of the flow size for heavy hitters with more than 10^3 packets.

The results are similar to the relative error of all flows shown in §6.2. Under the default setting, InvisiFlow at a 70% load exhibits a reduction of approximately 1.6× and 4.9× compared to low-priority UDP and pull-based baselines, respectively. In the asymmetric topology, the normalized relative error of all flows can reach up to 14.4 for low-priority UDP and 54.8 for pull-based baselines.

Unlocking ECMP Programmability for Precise Traffic Control

Yadong Liu[1*], Yunming Xiao[2*], Xuan Zhang[1], Weizhen Dang[1], Huihui Liu[1], Xiang Li[1], Zekun He[1], Jilong Wang[3], Aleksandar Kuzmanovic[4], Ang Chen[2], Congcong Miao[1]
[1] *Tencent,* [2] *University of Michigan,* [3] *Tsinghua University,* [4] *Northwestern University* *

Abstract

ECMP (equal-cost multi-path) has become a fundamental mechanism in data centers, which distributes flows along multiple equivalent paths based on their hash values. Randomized distribution optimizes for the aggregate case, spreading load across flows over time. However, there exists a class of important *Precise Traffic Control* (PTC) tasks that are at odds with ECMP randomness. For instance, if an end host perceives that its flows are traversing a problematic switch/link, it often needs to change their paths before a fix can be rolled out. With randomized hashing, existing solutions resort to modifying flow tuples; since hashing mechanisms are unknown and they vary across switches/vendors, it may take many trials before yielding a new path. Many other similar cases exist where precise and timely response is critical to the network.

We propose *programmable ECMP* (P-ECMP), a programming model, compiler, and runtime that provides precise traffic control. P-ECMP leverages an oft-ignored feature, ECMP groups, which allows for a constrained set of capabilities that are nonetheless sufficiently expressive for our tasks. An operator supplies high-level descriptions of their topology and policies, and our compiler generates PTC configurations for each switch. End hosts can reconfigure specific flows to use different PTC policies precisely and quickly, addressing a range of important use cases. We have evaluated P-ECMP using simulation at scale, and deployed one use case to a real-world data center that serves live user traffic.

1 Introduction

ECMP (equal-cost multipath) is foundational to modern data centers, for distributing traffic stochastically along multiple paths. ECMP is widely adopted not only because Fat-tree topologies [12, 26] naturally expose equivalent paths, but also because of its ease of implementation. The flow hashing mechanism at its core is efficiently realizable in ASICs and is widely available in commodity off-the-shelf switches.

Randomized flow hashing has turned out to be a simple yet effective mechanism. In data center networks, traffic patterns are hard to predict and they fluctuate over time. ECMP embraces this observation and optimizes for the aggregate. Individual flows might still fall victim to unfortunate choices of randomness (e.g., hash collisions between large flows); however, given enough flows, and when considering a long timespan, this randomness produces a good traffic spread and utilizes the underlying network efficiently. All production data center networks that we are aware of make use of ECMP.

However, strong aggregate performance does not remove the pitfalls of randomness in specific scenarios. In particular, we have identified a set of *Precise Traffic Control* (PTC) scenarios where rapid and precise response to network anomalies is key but randomness hinders it. Consider the case of switch/link failures that render an ECMP path unavailable or unstable [49, 56, 66]: it is then critical to quickly redirect network traffic off this path before we roll out a fix. In the presence of ECMP, this is no easy task: state-of-the-art solutions (e.g., Google's PLB [49] or PRR [56]) resort to random modification of flow tuples (e.g., by varying TCP ports), hoping that it would eventually yield a new path. This trial-and-error process could take minutes to complete [49]. Likewise, network monitoring systems [28, 31, 32] often need to quickly probe all paths to localize a fault; with randomness, we again need repeated flow modifications until all paths are covered. What these PTC scenarios have in common is the need to 1) exert precise control over traffic paths 2) against the backdrop of ECMP—the latter is equally important, as we cannot afford to *disable* ECMP, even if momentarily. In other words, the majority of network traffic should still be subjected to ECMP, while we seek a way to impose precise control *over* ECMP.

We propose to achieve this by carefully leveraging a commodity switch feature, ECMP groups, which is as simple, efficient, and widely available as basic ECMP itself. Basic ECMP maps a flow f to a preconfigured list of outgoing ports $l = [p_0, p_2, \cdots, p_n]$ based on the range that $hash(f)$ falls into. ECMP groups allow a flow to carry a selector s in its packet header, where each s maps to a its own port list l_s. Operators

*Equal contribution.

can configure the default ECMP group, say $\langle s = 0, l_{s=0} \rangle$, in conjunction with other $\langle s, l_s \rangle$ in an ECMP *matrix*. By doing so, operators could ensure, for instance, paths in l_s and $l_{s'}$ are always disjoint, and that precise control can be achieved by a single modification to the packet header field s. By leveraging this long-overlooked feature, we could instantiate ECMP matrices for each switch, so that traffic can enter/exit random modes of control by the end hosts toggling a packet header.

We call our key design *programmable ECMP* (P-ECMP), which does not require fanciful features (e.g., P4 programmability [29, 35, 39, 62]) but only exposes a restricted amount of reconfiguration for ECMP. The key challenge is to develop the 1) programming model, 2) compilation support, and 3) update mechanism for P-ECMP. While we could ask the operator to handcraft an ECMP matrix for each switch in her network, this is a tedious and error-prone undertaking. Instead, the operator describes her network in a topology matrix T, her PTC policies in control matrix C, and our compiler generates the ECMP matrices across the topology to realize precise control—while taking into account SRAM constraints across heterogeneous switches. Our consistent update mechanism further provides atomic transactions within and across switches, whether to repopulate the ECMP matrices or to dynamically change the selectors while traffic is still in flight.

We demonstrate a set of P-ECMP use cases that were previously difficult to achieve due to the lack of precise control:

- Network failover: End hosts can precisely re-path their flows to avoid perceived failures, whether silent packet drops or link flaps along an existing ECMP path.
- Failure localization: Network monitoring systems can precisely cover a set of underlying paths with their probes, e.g., to localize failures efficiently.
- Load imbalance: When skewed traffic patterns lead to performance degradation, end hosts can act upon congestion signals (e.g., ECN) to re-path select flows.
- Multipath protocols: With multipath protocols (e.g., MPTCP [50], MPRDMA [40]), we can ensure that each flow traverses a disjoint path for best performance.
- Packet spraying: Workloads that are resilient to network reordering [24] can spray packets across multiple paths for higher fairness and efficiency [21].
- Segment routing: Hosts can even precisely dictate all or parts of the forwarding paths for their flows, e.g., so that they traverse specific middleboxes.

The last use case, segment routing, points to the upperbound flexibility of P-ECMP in exerting precise control: over the entirety of a network path. The first use case, network failover, has been deployed and evaluated in a live data center network at scale. At the same time, all use cases are deployable in commodity data centers with legacy devices, and traffic that does not trigger PTC policies will continue to be handled by ECMP. Our evaluation with simulation and testbed experiments shows the effectiveness and low overhead of P-ECMP.

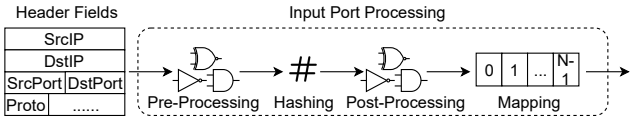


Figure 1: Four stages of ECMP processing.

2 Motivation

Many equal-cost paths exist in Clos topologies [12, 26], offering a natural opportunity for ECMP-based load balancing. When a packet arrives at an ingress port, relevant headers such as IP addresses and protocol types are parsed for further processing, including routing decisions and ECMP. The SAI (switch abstraction interface) [7] standard defines ECMP processing in four stages: pre-processing, hashing, post-processing, and mapping, as shown in Figure 1. During pre-processing, the switch performs bit-wise operations to obtain the flow ID, *e.g.,* the five-tuple <src_ip, dst_ip, src_port, dst_port, protocol>. The flow ID is then hashed by the hashing stage using ASIC functions (e.g., CRC) and reduced to a small integer. The post-processing stage also performs bit-wise operations, usually fixed in their logic, on the hash outputs. Finally, the mapping stage bins the hash output into one of the next hops $\{0, 1, \ldots, N-1\}$ for forwarding. All four stages are simple to realize in ASIC, so ECMP is widely available in commercial off-the-shelf switches.

2.1 The Pitfalls of Randomness

Since ECMP decisions are determined by the flow ID and the hash functions, across many flows the spreading pattern is essentially random. Furthermore, this randomness is difficult to predict or control, since switch vendors might employ different hash functions and their details are opaque to the network operator. While the pitfalls of ECMP regarding hash collisions resulting in (temporary) load imbalance are well-known, we have identified a more severe class of problems that, once they manifest, produce a more persistent issue inside the network.

Example: Network failover by re-pathing flows. Timely response to network failures is critical for high availability and service-level objectives (SLOs). Network failures can stem from a variety of issues, including hardware malfunctions in switches, software bugs, or configuration errors. While many of these failures can be addressed by the network control plane – for example, by using protocols like BFD to quickly detect failures and reroute traffic – others are more challenging. Failures such as silent packet drops, whether caused by misconfigured ACLs, routing blackholes, or errors in CRC computation, often affect only a subset of traffic and evade timely detection by traditional control-plane mechanisms.

In these cases, applications will perceive the failures earlier and the traditional wisdom is to produce a faster response from the end host side before the control plane fixes come in. Google [56] and Alibaba [66] have proposed using additional

header fields to modify the hash input used in network path selection, effectively tricking ECMP into recalculating the path. Applications can therefore re-path affected flows to bypass failures [56, 66]. Figure 2(a) shows an example where Flow B successfully avoids the failure encountered by Flow A on the original path.

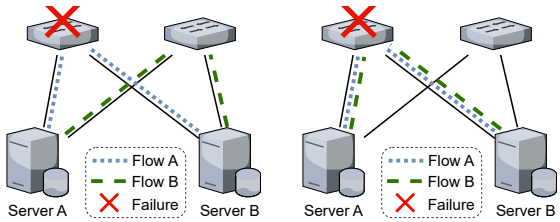
Problem: Lack of precise control. Since ECMP makes random decisions, and hash functions are opaque to operators, this re-pathing effort is essentially a trial-and-error process. It is common to encounter multiple failed re-pathing attempts due to hash collisions [56]. Figure 2(b)[1] illustrates a failed re-path attempt, where the re-pathed flow (Flow B) is mapped to the same path as the original flow (Flow A), encountering the same failure at Switch A.

Quantification of Failed Attempts. Quantitatively, the probability that either the "to" or "from" path of a new flow is mapped to a healthy path under ECMP is $\frac{N-1}{N}$, where N is the total number of equal-cost switches in the network tier where the failure occurs. The probability that the flow successfully circumvents the failure on its first re-path attempt, *i.e.,* both "to" and "from" paths avoid the failure point, is therefore $p = \frac{(N-1)^2}{N^2}$. If the flow fails to circumvent the failure on the first attempt, both "to" and "from" paths need to be re-selected randomly. Each re-path attempt is independent of the previous attempts and has the same probability of encountering the network failure as before, *i.e.,* $\frac{1}{N}$. Let T be the number of re-paths needed for successful failover. Then T has a geometric distribution with parameter p, where $P(T=k) = p(1-p)^{k-1}$. The expectation of T is given by:

$$\mathbb{E}[T] = \frac{1}{p} = \frac{N^2}{(N-1)^2}. \quad (1)$$

While the average number of new flows needed for a successful failover is relatively low, with T being at most 4 when $N = 2$ and decreasing when $N > 2$, its long-tail distribution can lead to extremely slow failovers for critical flows, resulting in SLA violations. For example, privious study [49] has shown that during a link failure, random re-pathing can cause certain flows to be disabled for 1-2 minutes due to unlucky failed re-pathing. This may further lead to unstable loads on other links for more than 20 minutes. Another proposal from Google is to re-path flows relying on TCP's RTO signal to indicates failures [56], where RTO's duration is influenced by RTT and subject to exponential backoff. Without a maximum limit on RTO and with a success probability $p \leq 0.5$, the average time for failure recovery using PRR cannot converge. Fortunately, RTO typically has an upper bound, and $p > 0.5$ when there are three or more switches in the ECMP group. Despite this, the recovery time in such setups is significantly longer compared to situations where failover is guaranteed to succeed on the initial attempt.

[1] For the sake of simplicity, examples in Figure 2 assume that the flow's "to" and "from" paths are identical. However, in reality, the "to" and "from" paths of the same flow may be different, as the flow identifier, *i.e.,* five-tuple header fields, is different when the source and destination are reversed.



(a) Succeeded case. (b) Failed case.

Figure 2: Demonstration of (a) succeeded and (b) failed failover by flow re-pathing or multi-path protocols.

2.2 Precise Traffic Control

Re-pathing is not an isolated case where the lack of precision causes operational challenges. In fact, there are many scenarios where precise traffic control (PTC) is critical.

Network failure localization. ECMP also hinders network failure localization, a crucial function for maintaining and ensuring high availability in modern data centers. Many data centers employ systems similar to Pingmesh [28], where servers regularly send ping (ICMP) packets to probe network availability. However, due to the randomness of ECMP, these probes are randomly distributed across available paths, without guaranteed coverage of all network hops (i.e., switches). As a result, achieving full path coverage requires an excessive number of probe packets, often exceeding the number of actual paths between any two servers. Further, the randomness in ECMP limits monitoring tools' ability to effectively identify localized network issues. While it can detect regional problems, it lacks the granularity needed to precisely pinpoint failures at the switch or link level [54].

Load imbalance. The goal of ECMP is to distribute flows based on their five-tuples, achieving an even spread. However, as the traffic loads are generally long tailed, the randomness in ECMP may lead to imbalances in real-world workloads, or even exacerbate congestion hotspots [13, 34]. To mitigate this, Google introduced a method called PLB, where applications re-path their flows upon detecting congestion signals – such as a few ECN-marked packets from switches – thereby improving load balancing [49]. However, addressing load imbalance in this way can lead to failed attempts. At certain cases, flows can be disrupted for 1-2 minutes due to unlucky attempts, destabilizing load distribution across other links for over 20 minutes, as reported by Google's PLB [49].

Multi-flow applications and multi-path protocols. Applications may initiate multiple flows for their transfer, and some transport protocol such as MPTCP [16] or MPRDMA [40] are inherently multipath. By exposing path diversity, we can leverage multiple paths for transmission, identify best paths, and achieve better balance in terms of path utilization – assuming that paths are independent. However, multi-flow applications and multi-path protocols do not interact well with ECMP. Its randomness may mean that paths could be overlapping. This could lead to (sub-)flows being routed along the same path,

diminishing the benefits of using multiple paths.

Packet spraying. Packet spraying is a technique that distributes packets across multiple available paths on a per-packet manner. This is particularly useful for flows that do not require strict packet ordering – such as UDP flows or networks with reordering-resilient network stack [24]. Previous results show that packet spraying can greatly improve network fairness and efficiency [21]. Unfortunately, the randomness in ECMP routing also undermines the effectiveness of packet spraying, as it prevents precise distribution of packets across paths, leading to potential hotspots and imbalance.

Segment routing. Finally, segment routing is a network function that enables the sender (host machines) to specify all or part of the forwarding path for packets. It is widely employed in wide-area networks (WANs) to support traffic engineering [44]. In data center networks, applications can benefit from segment routing such as diverting specific packets to designated middleboxes. However, segment routing is a network function that demands precise control, which is not compatible with the randomness of ECMP.

In all of these cases, we need a solution that enables the majority of the traffic to still use ECMP, while temporarily allowing certain flows or applications to exert precise control over ECMP. One solution is to implement these use cases in programmable switches, so that certain flows are processed differently when they traverse the network. However, this requires a universal deployment of programmable switches inside data center networks. Another solution is to use out-of-band mechanisms, as proposed in RePaC [66], which precomputes the mappings between inputs and outputs of the hash algorithm in ECMP and distributes these mappings to host machines. This allows host machines to exercise precise control of relative path (offset). However, this assumes uniform switch configurations within the data center network, whereas operators often source equipment from multiple vendors to avoid vendor monopoly and increase resilience against single-product flaws.

2.3 Idea: Leverage ECMP Groups

To support PTC use cases on commodity switches, our key observation is that ECMP has a certain amount of flexibility that we have not fully used. This is signaled by the fact that a significant portion of the SRAM memory resources reserved for ECMP routing remains underutilized. Indeed, the abundant ECMP table capacity is primarily designed to support ISP use cases, where asymmetric topologies and a large number of destination sets are common. However, this is not the case in DCNs which have symmetric topologies and thus require fewer ECMP groups at normal times. Our measurement of the commodity switches in a production DCN revealed that, while hardware providers (e.g., Trident5 ASIC) allocate space for 4 pipelines (e.g., uplink or downlink), with 8K ECMP groups and 64K ECMP members per pipeline, current switches use only up to 8 groups (0.1%) and 512 members (<1%) at most (spine switches). ToR and leaf switches use only one group (0.01%) and up to 64 members in uplinks (0.1%). Our idea is to extend the basic ECMP mechanism both *horizontally* and *vertically* leveraging *ECMP groups*, a common feature in ECMP-capable switches.

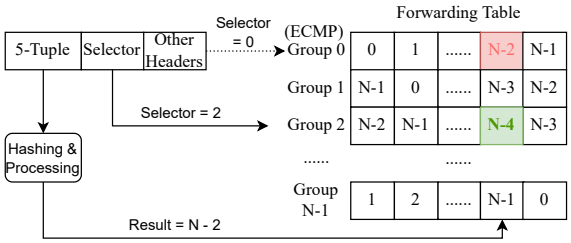


Figure 3: An example of PTC of path offset by P-ECMP where the selector is 2 and the outcome of the ECMP processing is $N-2$. By redirecting the ECMP group by 2, P-ECMP re-directs the forwarding port from $N-2$ to $N-4$.

In basic ECMP, the hash value of a flow tuple $Hash(f)$ is used as the index into an array to determine the output port. The ECMP group feature allows for multiple such arrays as depicted in Figure 3. These multiple (M) ECMP groups transform the forwarding table from a simple array (with only group 0) into a matrix C. The selection of an ECMP group can be driven by an additional field in the packet header, referred to as the selector s. As a result, the forwarding decision is based on the tuple $< s, Hash(f) >$, where s determines the row, and the hash of the flow's selected field $Hash(f)$ determines the column. That is, the final output port is given by $C[s, Hash(f)]$. This extended flexibility in the forwarding table enables us to tune the C matrix for precise control.

Horizontal: Let us first consider only one ECMP group, with C being a one-dimensional array $[0, 1, \ldots, N-1]$. This already allows for extending the forwarding behavior by inserting, deleting, or replacing items in this array. For instance, by adjusting the number of entries in the table according to specific rules, we can implement weighted ECMP, where the weight of each next hop reflects its proportion in the table. If the array contains only a single entry, this effectively achieves precise control, eliminating randomness so that packets are always forwarded to the same next hop.

Vertical: Considering multiple ECMP groups provides further flexibility, in two dimensions: (i) multiple ECMP policies can coexist simultaneously, allowing applications to select a policy on demand, and (ii) it introduces an additional layer of determinism on top of the inherent randomness of ECMP. Specifically, applications can embed a value in a custom header field to specify which row of the matrix to use. The row selection is deterministic, while the column selection within each row (*i.e.,* the ECMP routing) remains random.

2.4 P-ECMP contributions

The contributions we made in developing *programmable ECMP* (P-ECMP) are as follows.

Programming model. Our first contribution is a systematic study and exploration of the inherent flexibility in ECMP, upon which we distill the abstraction of ECMP *programmability*. Specifically, we categorize the programmability of ECMP into three sets of intent: (i) PTC of path offset, (ii) PTC of the exact next hops, and (iii) Weighted-Cost Multi-Path (WCMP). Since our approach to WCMP aligns with previous work [29], this paper focuses on the first two intents.

Compiler. We also develop a compiler that automates the process of generating ECMP forwarding tables for all switches in the network. The compiler takes the programmability intent, along with the network topology and resource constraints, as inputs. This streamlines the development and deployment of ECMP programmability, making it intuitive and easy for network operators to implement and manage.

Runtime updates. A challenge with ECMP programmability lies in ensuring consistency during deployment. Specifically, when deploying ECMP forwarding tables on commodity switches in production DCNs, issues can arise. For example, a host machine might have an outdated database and send a packet with a flag corresponding to an invalid ECMP policy, or in-flight packets might not be able to adjust their embedded flags and thus hit an invalid policy at next switches. This could lead to packet loss or out-of-order delivery. Given the constant high volume of traffic, an inconsistent update could have a significant impact. To address this, we propose a distributed transaction scheme that ensures consistency during updates, preventing packet loss or reordering issues.

Rich use cases. P-ECMP enables a range of PTC tasks, such as the ones motivated before. We evaluate these use cases and report our experience in testing the network failover use case in a production data center at scale.

3 System Design

3.1 Where Can We Program ECMP?

To better understand the potential programmability residing in commodity switches, let us first trace back to the architecture of commodity switches and its forwarding process. In the ECMP processing described above, three elements can be modified: the input fields, the hashing algorithm, and the mapping stage. Below, we explore whether modifying each of these components can alter the output port and enable programmability. Table 1 summarizes the analysis.

First, modifying the input fields in ECMP processing enables the generation of a different hash output, which in turn determines a different output port. This approach has been employed in several studies, such as PLB [49] for improved load balancing, and PRR [56] and RePaC [66] for network failover. These studies introduce an additional header field as an input to ECMP, beyond the standard five-tuple flow definition. This allows for changing the forwarding output port by adjusting the value of the extra header field without disrupting ongoing flow sessions.

Table 1: Programmability potential of commodity switch at different forwarding stages.

Stage		PTC of Offset	PTC of Hop	Compatibility	Practicality
	Input	Yes*	Yes*	Yes	No
	Hashing	No	No	No	No
Mapping	Member Redirect	No	Yes	No	Yes
	Group Redirect	**Yes**	**Yes**	**Yes**	**Yes**

However, if the hashing algorithm used by the switches is unknown, this modification may result in random outcomes, thereby limiting programmability. When the hashing algorithm is known in advance, such as what was assumed in RePaC [66], it is possible to deterministically adjust the value of the extra header field to achieve the desired output port. Specifically, let f represent the selected flow fields for hashing, and $Hash(f)$ denote the hashing result. RePaC deterministically alters the output port by modifying the input fields with a difference $\Delta \in \{0x0...01, 0x0...10, \ldots, 0x1...00\}$, *i.e.,* the output becomes $Hash(f+\Delta)$. Leveraging the linearity of hashing algorithms, we have $Hash(f+\Delta) = Hash(f) + Hash(\Delta)$. This gives that $Hash(\Delta)$ can be used to control the output port offset. However, programmability can still be constrained by the significant heterogeneity of switches. For example, in heterogeneous environments where ToR, leaf, and spine switches employ different hashing algorithms [61], finding a suitable value for the additional header field may become infeasible due to the diversity of hashing functions across switches. For further details, please refer to Appendix A. In summary, RePaC only enables PTC of offsets in limited scenarios. Thus, it is not feasible for production deployment.

A second approach involves changing the hashing algorithm to generate different outputs for different input packets. However, this requires specialized switch chips capable of selecting the hashing algorithm based on a specific header field value in incoming packets. In our data center, only a small fraction of switches support such modifications, making this approach impractical. Additionally, changing the hashing algorithm does not necessarily guarantee that a different output port will be selected, as different algorithms may still produce the same output port for certain inputs (see Appendix B). Even when successful, this method demands extensive computation of all possible input-to-output mappings to achieve the desired network functions by selecting appropriate selectors. Moreover, the range of possible functions is constrained by the limited number of available hash functions.

The final approach involves modifying the mapping stage, which can be done in two ways: (i) redirecting the packet to a different member within the same ECMP group or (ii)

redirecting it to a different ECMP group while keeping the member fixed. However, the first approach faces compatibility issues. Many switches do not support performing arbitrary mathematical operations on the post-processing results; the only feasible modification is to replace the post-processed result with a predefined value (such as selector s). This limitation reduces programmability, allowing only PTC of hops but preventing the use of the hashing process to support PTC of offset. As a result, we discard this approach.

On the other hand, the second approach of redirecting the packet to a different ECMP group is more feasible and aligns well with our goals. Therefore, we choose this method as the foundation for P-ECMP.

3.2 Programming Model and Use Cases

Let $l = [p_0, p_1, \cdots, p_{N-1}]$ represent an ECMP group, where each p_i ($i \in [0, N-1]$) corresponds to an output port, and N is the total number of available ports. For each ECMP group, the port selection for a flow f is determined by a hashing function, $Hash(f)$. P-ECMP generalizes this by using multiple (M) ECMP groups instead of just one, transforming the forwarding table into an ECMP matrix, referred to as the control matrix C, whose dimension is $M \times N$. We assume that users can select a specific ECMP group by embedding a selector s in the packet header, i.e., l_s. If the selector value exceeds the total number of ECMP groups ($s > M$), it will be taken modulo M, i.e., $s \bmod M$. As a result, the port selection for flow f becomes an item selection in the matrix, $C[s \bmod M, Hash(f)]$.

Next, we demonstrate how PTC can be implemented. We categorize two types of PTCs, each associated with a different control matrix C. For simplicity, we will describe each type individually before addressing how to support them concurrently in § 3.3.

Precise traffic control of path offsets. A useful form of PTC involves adding an offset to the existing randomness introduced by ECMP. Specifically, let us assume a default selector s_0 and a custom selector s. The desired property is that the output port selection shifts by a constant offset corresponding to the difference between s_0 and s for a given flow. Formally, this means $F[s, Hash(f)] - F[s_0, Hash(f)] = G(s) - G(s_0)$, where G is a linear function. Taken $G(x) = x$, we have $F[s, Hash(f)] - F[s_0, Hash(f)] = s - s_0$. In this way, the randomness of ECMP is maintained, *i.e.,* with $Hash(f)$ used for column selection, while PTC of path offset is achieved through the selector $s - s_0$.

The control matrix that satisfies this property can be constructed by recursively copying the last available ECMP group and shifting each subsequent group by one position, either to the right or left. For example, if the first ECMP group is $l_0 = [p_0, p_1, \ldots, p_{N-1}]$, the next group would be $l_1 = [p_{N-1}, p_0, \ldots, p_{N-2}]$. This process continues until all N ECMP groups are generated, with the final group being $l_{N-1} = [p_1, p_2, \ldots, p_{N-1}, p_0]$, as illustrated in Figure 3.

The PTC of offset supports various use cases:

- Network failover. Google has proposed randomly re-routing flows upon detecting a network failure signal, such as an RTO [56], by adding a random value in the IPv6 header to alter the ECMP hash outcome. However, this approach is prone to re-pathing failures caused by hash collisions, where the re-pathed flow may still be mapped to the same failed path.

  In contrast, the ability to perform packet forwarding with precise offset control enables deterministic re-pathing, ensuring successful failover. For example, assume the default selector is $s_0 = 0$. When a flow needs to be re-pathed, it can select any s that is not equal to the default selector and does not result in the same path, i.e., $s - s_0 \not\equiv 0 \bmod M$.

  In practice, the value of M (the number of ECMP groups) may not be knowledgeable to applications, and there could be multiple values of M corresponding to different layers of switches in the network. A robust solution is to adopt a sufficiently large prime number for M, which works across any network topology. In most modern data centers, a prime number greater than 16 is typically adequate.

  Another concern is that re-pathed flows should ideally remain load-balanced. We provide a detailed discussion on selector selection to realize this in Appendix C.

- Load imbalance. Similar to network failover, Google proposed PLB [49], which randomly re-path flows to avoid congestion points. In contrast, PTC of offsets allows applications to re-path flows with a specific offset that ensures they will not be mapped to the original path, guaranteeing successful flow re-pathing. The selector value used is the same as in the network failover scenarios.

- Multipath protocols. PTC of offset can also enhance multipath protocols by ensuring that no sub-flows are mapped to the same path, thereby improving resilience against network failures or congestion.

  Unlike flow re-pathing, multipath protocols using P-ECMP should assign increasing integers starting from 0 as the selector s[2]. For instance, if an MPTCP flow creates 4 sub-flows, the selectors s should be $0, 1, 2, 3$, respectively. When all network tiers have at least 4 equal-cost paths, each sub-flow will be mapped to a distinct path with no overlap. However, if a network tier has fewer than 4 available paths, two sub-flows may overlap at one hop, reducing MPTCP's effectiveness. This limitation is not due to P-ECMP but is an inherent consequence of the pigeonhole principle. Determining the optimal number of sub-flows requires analysis of the network topology and workloads, which is beyond the scope of this study.

[2] A native implementation of MPTCP uses random port numbers for its sub-flows, which makes assigning increasing integers starting from 0 as the selector s ineffective. Instead, we have implemented a custom MPTCP protocol where sub-flows are distinguished based on the selector s.

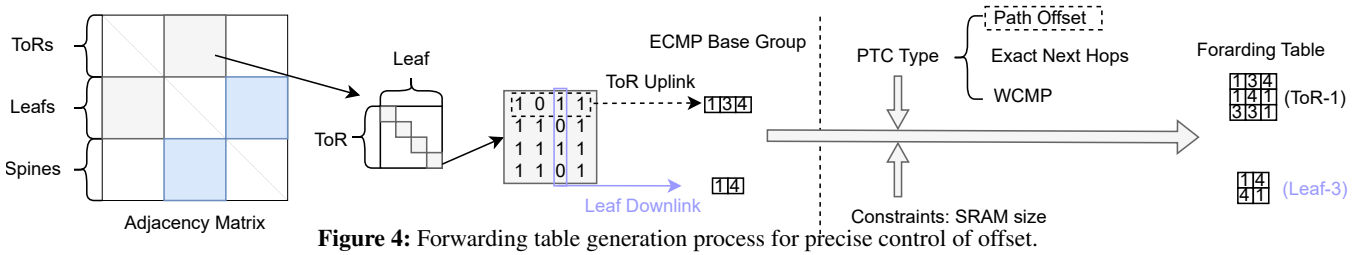


Figure 4: Forwarding table generation process for precise control of offset.

Precise traffic control of exact next hops. P-ECMP can achieve greater precision by removing the hashing process. This can be done by either setting all output ports in an ECMP group to the same port or reducing the number of output ports to just one (i.e., making $l_s = [p_s]$), allowing the output port to be determined entirely by the selector s. At different network tiers, commodity switches can use different packet headers or different bits of the same header field as the selector. As a result, multiple selectors can be configured by the sender, enabling precise control over the entire network path.

This also supports various use cases:

- Failure localization. Existing network failure localization tools, such as Pingmesh [28], need to generate an excessive number of probe packets to cover all paths due to the inherent randomness in ECMP routing. To overcome this limitation, systems like NetBouncer [54] employ IP-in-IP encapsulation to systematically probe all possible paths, providing more accurate and fine-grained failure localization at the cost of incurring additional overhead due to the encapsulation technique. In contrast, P-ECMP offers an optimized alternative for network failure localization, allowing for precise control over all paths without the overhead associated with IP-in-IP encapsulation.
- Packet spraying. Previous approaches to packet spraying have typically followed a random manner [21], relying on ECMP routing. In contrast, P-ECMP enables deterministic packet spraying, enhancing efficiency and fairness beyond what can be achieved through random packet distribution.
- Segment routing. Commodity switches used in today's data centers do not genuinely support segment routing natively, which contributes to its limited adoption. As a workaround, segment routing is often achieved using techniques like IP-in-IP encapsulation [47], though this comes with the tradeoff of increased header size.

  By leveraging the ability to specify the exact output ports with P-ECMP, segment routing can be implemented without the need for IP-in-IP encapsulation. In this approach, the selector s serves as an indicator of the intended path, and together with coordinated forwarding tables across different network tiers realizes the segment routing.

3.3 Compilation Support

To deploy the aforementioned PTC, operators must offload the control matrix as forwarding tables into every switch in the data center network. Manually performing this task would be highly labor-intensive and prone to errors. Instead, we propose automating this process using a compiler. At a high level, the compiler takes as input the type of PTC (as described in § 3.2), the network topology, and the SRAM constraints of the heterogeneous switches. The output of the compiler is the forwarding tables for all switches in the network.

Figure 4 shows the network's adjacency matrix for a topology with switches organized into three tiers: top-of-rack (ToR), leaf, and spine. While P-ECMP can adapt to any topology, here we assume a fat-tree topology with partial connectivity for simplicity. The adjacency matrix is sparse due to the absence of interconnectivity within the same tier (e.g., between ToR switches). We omit sections of the matrix where all values are zero and focus on the portion where ToR switches connect to leaf switches. By scanning each row, we can identify the ECMP base group, representing the available next hops for each ToR switch. In Figure 4, the first ToR switch connects to all leaf switches except the second, resulting in an ECMP base group of $[1,3,4]$ for its uplinks. Similarly, scanning each column reveals the ECMP base group for the leaf switches' downlinks. Note that these numbers represent the global switch IDs across the entire data center. During forwarding table deployment, translating the global switch IDs to the corresponding output ports on each switch is needed.

After determining the ECMP base group for each switch, the next step is to incorporate additional inputs, such as the desired PTC type and the SRAM constraints of each switch, to generate the forwarding table. The process for generating the forwarding table for each PTC type is straightforward, as outlined in § 3.2. The control matrices for different PTC types can be concatenated, with the matrix for precise control of offset appended after the base ECMP group being preferred. This is because the default ECMP group is also part of the control matrix for precise control of offset. We discuss the design of the selector bits in Appendix E.

Note that the PTC of offsets does not necessarily require knowledge of the network topology, whereas the PTC of hops does. When both types of PTC are supported concurrently, servers need to be aware of the network topology and the mappings between selectors and specific PTC policies.

3.4 Update Mechanism

The dynamic nature of data center networks, such as topology changes, requires updates to the forwarding tables in switches.

However, these updates can lead to inconsistencies between the PTC policies in the switches and the host machines, potentially causing packet drops or unintended forwarding behavior for a short period. Given the high bandwidth in data centers – often in the tens or hundreds of Gbps – even brief periods of inconsistency, or no delay but with packets already in transit, can significantly impact data. For instance, at 100 Gbps, a 10 ms disruption may affect 125 MB of data per link. The impact will be significantly larger considering the entire data center network.

To mitigate this issue, we propose a transactional runtime update mechanism that ensures global consistency inspired by previous work [45, 60, 64]. Specifically, we treat the forwarding tables as versioned entities, where each version corresponds to a specific range of selectors. When an update is imposed at runtime, instead of modifying the current version of the forwarding table during an update, a new version is created. This ensures that the existing forwarding table remains unchanged during runtime. Once the new version of the forwarding table is ready, servers update their selectors to align with the new version, guaranteeing a smooth runtime update without forwarding glitches.

In practice, we divide the SRAM space at switches into two symmetric halves, with each half occupying equal space. One half stores the current version of the forwarding table, while the other holds the new version during updates. For instance, if the SRAM can accommodate up to M ECMP groups, ECMP groups 0 to $\lfloor \frac{M-1}{2} \rfloor$ would represent the current version, while groups $\lfloor \frac{M-1}{2} \rfloor + 1$ to $M-1$ would represent the new version. When the new table is in place, servers update their selectors by adding $\lfloor \frac{M}{2} \rfloor$ to adapt to the new version. Once all servers have transitioned, the new version becomes the active one, freeing the 0 to $\lfloor \frac{M-1}{2} \rfloor$ range for the next update cycle.

It is important to note that updates to PTC policies are only necessary for addressing stationary topology changes, such as network expansion, reconstruction, or long-lasting failures and maintenance. These updates are not required for transient failures, such as temporary congestion or brief hardware/software glitches, which are short-lived and self-resolving. As a result, compiling a new version of ECMP groups is not a time-critical task. Nevertheless, our system is capable of compiling the new configurations and offloading them to all switches in the production network within a few milliseconds.

4 Implementation

Switch. We have implemented the aforementioned functionalities across all switches used in our production data centers, primarily including various series of Trident and Tomahawk. Our implementation is built on SONiC [9], where we ensure each operation at a switch – such as adding, replacing, or deleting ECMP groups and their members – is atomic by using locks. Additionally, when making network-wide changes, we utilize the transactional runtime update mechanism (§ 3.4) to maintain consistency across the entire network.

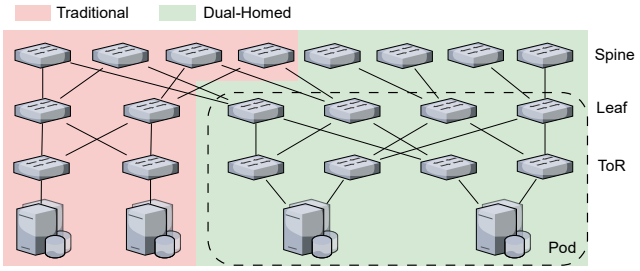


Figure 5: Comparison of traditional and dual-homed data center network architectures.

Host machine. When a host machine needs to invoke the various network functions supported by P-ECMP, such as re-pathing flows after detecting a failure, it populates the selector s in the packet header with the appropriate value corresponding to the desired network function.

Several prior works have explored different methods for encoding the selector s within the packet header. For example, FlowBlender [34] uses the VLAN tag, while PLB [49] and PRR [56] leverage the IPv6 Flow Label. Both of these fields are compatible with P-ECMP. However, in our implementation within the production network (§ 5.3), we avoid both options because (*i*) VLAN tags has been used for other purposes in our production network, and (*ii*) IPv6 Flow Labels are not useful for IPv4 traffic, which is still prevalent on the Internet and in our data centers. Instead, we select the Differentiated Services Code Point (DSCP) in the IP header to encode the selector s value. This is reasonable as DSCP is intended to be used for classifying and managing network traffic, and providing quality of service (QoS) in modern IP networks [27]. it also avoids conflicts with other packet or flow-related operations in our data center. Note that DSCP, VLAN, and Flow Label are not the only viable options – data center operators should select the appropriate header field for carrying the selector s based on their specific environment and requirements.

There are various approaches for modifying the DSCP value. For instance, one can modify the DSCP at the kernel level using eBPF [3] or by introducing a loadable kernel module [4]. Alternatively, one can modify the DSCP at the application level by calling `SetSocketOpt` [8], a function provided by the Unix socket interface, to set the desired value. We adopt the last approach in our implementation.

Note that the focus of this paper is on exploring the programmability of ECMP forwarding on commodity switches, rather than addressing when and how specific network functions should and can be applied, such as how to detect silent packet drops or congestion. For these aspects, we simply rely on state-of-the-art approaches. For instance, in network failover using deterministic re-pathing, we follow PRR [56] by using TCP RTOs as indicators of silent packet drops. For load

balancing, we adopt PLB's approach [49], using a fixed number of packets with ECN marks to detect network congestion. Additionally, our implementation supports application-level signals that can be customized based on user requirements.

Multi-homed server. As a leading cloud provider, we have built and been managing multiple large data centers. Nevertheless, we adopted a different approach when constructing them. One critical revision is that we adopt a dual-homed server bonding architecture where each server connects to two ToR switches. As illustrated in the green section of Figure 5, the numbers of ToR, leaf, and spine switches are hence doubled compared to the standard fat-tree topology. The dual-homed server bonding provides several benefits, including improved network resiliency and reduced downtime. It has also been adopted by other providers recently [48].

Given our dual-homed setup, it is essential for host machines to support similar functionalities provided by P-ECMP-enabled switches across different network interface cards (NICs) as well. A common and convenient practice is to bond multiple NICs to a single logically bonded interface, which allows flows to be distributed across different NICs and paths [5]. However, bonding often relies on hashing algorithms, which introduce randomness as ECMP.

Instead, we adopt a design approach similar to our switch-based solution. Specifically, we add the selector value to the output of the hashing algorithm and perform modulo calculations afterwards. This ensures deterministic path selection and avoids any randomness or the need to modify the hashing algorithm itself. Our approach can be implemented by a hot patch to the NIC bonding driver, making the support for multi-homed networks easy and light.

5 Evaluation

We begin by evaluating the compilation and update mechanism of P-ECMP (§ 5.1). Next, we demonstrate the effectiveness of utilizing P-ECMP to address various critical network corner cases (§ 5.2). Most experiments in § 5.1 and § 5.2 are done using the NS3 simulator [6]. Finally, we share our experience deploying network failover enabled by P-ECMP's PTC of path offset capability in our production network (§ 5.3).

5.1 P-ECMP Compilation and Update

We first present the resource requirements for deploying P-ECMP after compilation. We evaluate multiple topologies, as listed in Table 2, including whether dual-homed architecture is adopted, the number of leaf and spine switches (determined by the product of the number of spines per plane and the number of planes, where the number of planes equals the number of leaf switches per pod), and the resulting maximum number of paths between two servers. Among them, the largest topology #12 is the same as our production network.

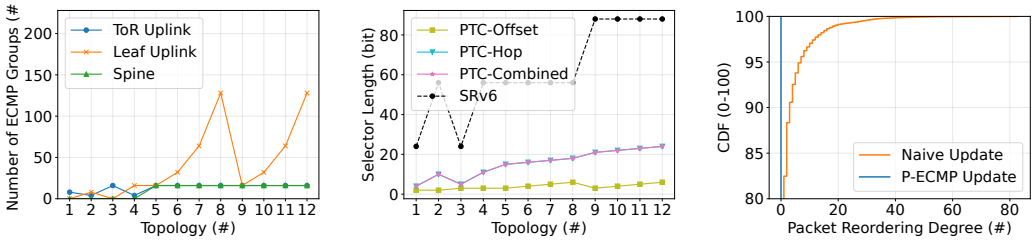
Table 2: Topologies for evaluation. OSR stands for oversubscription ratio.

Topo	Dual-Homed	# Leaf Per Pod	# Spine	OSR	Max # Paths Between Servers
#1	N	4	N/A	N/A	4
#2	Y	4	N/A	N/A	8
#3	N	8	N/A	N/A	8
#4	Y	8	N/A	N/A	16
#5	N	8	8×8	8:1	64
#6	N	8	16×8	4:1	128
#7	N	8	32×8	2:1	256
#8	N	8	64×8	1:1	512
#9	Y	8	8×8	8:1	128
#10	Y	8	16×8	4:1	256
#11	Y	8	32×8	2:1	512
#12	Y	8	64×8	1:1	1,024

Figure 6(a) shows the number of ECMP groups required for P-ECMP across different switches under various topologies, with both PTCs enabled. At the ToR uplink, between 4 and 16 ECMP groups are needed. For leaf switches (uplink), the SRAM resources range from 4 to 128 ECMP groups, except in cases where spine switches are not available, requiring 0 groups. For both ToR and leaf switches, the available SRAM resources are more than sufficient to accommodate P-ECMP. At spine switches, the SRAM resources required range from 4 to 16 ECMP groups per pod. However, it is important to note that this number must be multiplied by $(x-1)$ for a network with x pods. Given that the Trident 5 ASIC provides 8K ECMP groups and accounting for our transactional update mechanism – which halves the available ECMP space – P-ECMP can support networks with up to 512 pods based on ECMP considerations. Nevertheless, the actual number is capped at 128 due to the spine switch's 128-port limit. In other words, P-ECMP's consumption of SRAM is not the bottleneck. Notably, when using either the PTC of offset or hop, the number of required ECMP groups is halved (see Figure 11 in Appendix D).

We move on to the bit length of selector s. As shown in Figure 6(b), when only the PTC of offset is required, the selector ranges from 2 to 6 bits, making a 6-bit DSCP sufficient. If the PTC of path is needed, whether combined with the offset PTC or not, the selector will require more bits. For single-homed topologies (up to topology #8), fewer than 20 bits are needed, making the 20-bit IPv6 FlowLabel a suitable option for carrying the selector. For larger and dual-homed topologies, up to 24 bits may be required, necessitating the use of multiple header fields or other headers not covered in § 4. Nevertheless, compared to SRv6 or IP-in-IP (which incurs even higher overhead than SRv6 but is not depicted in Figure 6(b) for clarity), P-ECMP reduces the header overhead by 3x to 6x.

Next, we evaluate our transactional runtime update mechanism. At each switch, updating ECMP groups typically takes less than 2ms. While updating a single switch is atomic, updating all switches without our transactional update mechanism (§ 3.4) can result in packet reordering due to inconsistencies



(a) Number of ECMP groups. (b) Bit length of selector. (c) Packet reordering degree.

Figure 6: Evaluation of P-ECMP compilation and update mechanism.

across switches and in-flight packets using outdated P-ECMP selectors. We generate a Hadoop workload [10, 51] on topology #4 to assess the packet reordering degree at the receiver, defined as the number of packets arriving earlier or later than their expected sequence position. As illustrated in Figure 6(c), without our transactional update mechanism, the reordering degree can reach up to 83, with a 99th percentile of 19. Depending on the receiver's tolerance for packet reordering, such high levels of reordering may lead to performance degradation, or even packet loss in severe cases.

5.2 Network Functions

Next, we evaluate the support of various network functions by P-ECMP. We use topology #12 for all experiments below.

Network failover. We begin by evaluating the failure recovery of P-ECMP with a single flow. Specifically, we initiate a flow at full rate and introduce a network failure later. We follow PRR [56] to use TCP RTO as the indicator for network failures. We also change the RTO calculation following PRR to better adapt to data center networking scenarios, where RTO $\approx$ RTT + 5ms and has a cap of 1 second. It is noteworthy that the indicator for network failure and the trigger for flow re-path operation is up to user customization, which can react faster or slower than in our experiment.

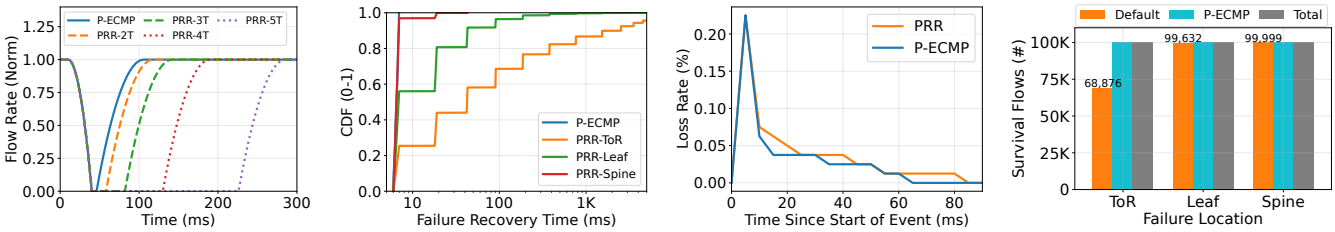
Figure 7(a) compares the performance of P-ECMP to PRR as a representative of random re-pathing. The failure occurs at 10ms. After that, the host takes roughly 6ms to detect the failure and trigger flow re-path. If the first re-path fails, the flow will pause and wait for another RTO, which grows exponentially, before attempting another re-path. In this experiment, P-ECMP succeeds in the first re-path attempt, resulting in full flow recovery at 106ms. However, PRR or other random re-pathing may require two (PRR-2T), three (PRR-3T), four (PRR-4T), or more attempts, resulting in flow recovery at 118ms, 142ms, 190ms, or even later.

We repeat the experiments for 10,000 times. Figure 7(b) presents the CDF of the failover duration for P-ECMP and PRR under different failure scenarios: at the ToR switch (PRR-ToR), at the leaf switch (PRR-Leaf), or at the spine switch (PRR-Spine). The results demonstrate the long-tail distribution of flow recovery duration by PRR (note that X-axis is in log scale). Failures at the ToR switch have the most significant impact on flows, requiring a median recovery duration of 42ms and 95th percentile recovery duration of notably over 4.5s. Failures at the leaf switch have an intermediate impact, with a median recovery duration of 7ms and a 95th percentile recovery duration of 91ms. Note that in practice, failures indeed mostly occur at ToR or leaf layers. Finally, failures at the spine switch have the smallest impact, with only a slight chance ($\sim$3%) of failure during the first re-path attempt. In contrast, P-ECMP always succeeds in re-pathing the flow in the first attempt regardless of the failure location, and hence realizing a stable flow recovery duration of 6ms.

We further present a case study of link failure between a ToR and a leaf switch. Prior to the failure, we initiated data flows of varying sizes mimicking web search workloads [10, 15] from each host machine to randomly selected destination machines. Figure 7(c) illustrates the packet loss rates over time, starting from the onset of the failure. Both PRR and P-ECMP rapidly decrease loss rates, with PRR reducing the loss from over 0.2% to 0% within 85ms. Notably, P-ECMP surpasses PRR in reducing loss rates, achieving failure mitigation within 65ms – a 24% improvement over PRR. Although our current data limitations prevent a similar emulation to more outage events presented in PRR report [56], we expect that P-ECMP will consistently outperform PRR across various scenarios, owing to its PTC capability.

Load imbalance. Next, we evaluate P-ECMP's performance against load imbalance. For each host machine, a destination is randomly chosen, and we generate flows with sizes following web search and Hadoop workload [10, 14, 15, 51]. We compare P-ECMP to PLB [49], a leading load balancing scheme leveraging random flow re-pathing. We incorporate PTC of offset by P-ECMP into the re-pathing phase of PLB [49], and employ the same parameters as PLB.

Table 3 presents the total number of flow re-paths and normalized flow completion time (FCT) results. Our analysis shows that P-ECMP requires fewer re-paths in half of the cases, while in the other half, it incurs a marginally higher number of re-paths (1% to 3%) compared to PLB. This does not indicate that P-ECMP fails to re-path flows on the first attempt; rather, the timing of successful re-paths (e.g., P-ECMP succeeding in the first attempt while PLB requires multiple attempts) can alter the workload distribution across remaining paths and influence the overall network dynamics. What is more important is that P-ECMP achieves significant improvements across various metrics, including p50 FCT, p99



(a) Single-flow recover example. (b) Single-flow recovery duration. (c) Failover from a link failure event. (d) MPTCP enhancement.

Figure 7: Use case evaluation of P-ECMP.

Table 3: Normalized flow completion time with load balancers. LF stands for last flow (*i.e.,* 100th percentile).

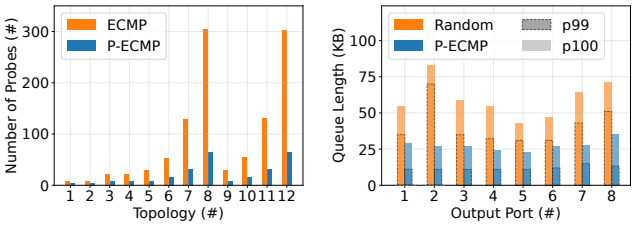
Trace	LB Scheme	30% Load				50% Load				80% Load			
		p50	p99	LF	#RePath	p50	p99	LF	#RePath	p50	p99	LF	#RePath
Web Search	PLB	1	21.5	33.0	1x	1	**22.5**	214.3	1x	1	54.9	366.7	**1x**
	P-ECMP	**0.90**	**8.1**	**13.6**	**0.96x**	**0.91**	23.1	**182.4**	**0.93x**	**0.92**	**45.0**	**189.0**	1.01x
Hadoop	PLB	1	147.2	334.0	**1x**	**1**	212.4	4,700.0	1x	**1**	41.9	1,278.3	**1x**
	P-ECMP	**0.92**	**135.0**	**296.7**	1.02x	1.01	**188.3**	**2,246.3**	**0.97x**	1.01	**40.7**	**769.1**	1.03x

FCT, and last flow (LF) completion time, in most scenarios. Notably, under high network load, P-ECMP nearly halves the last flow completion time, highlighting its effectiveness in addressing the randomness and delays associated with PLB.

Multi-path protocol. We then evaluate P-ECMP's enhancement to the robustness of multi-path protocols. We have implemented a custom MPTCP based on P-ECMP, where subflows are distinguished using the P-ECMP selector. We generate 100K MPTCP flows between two servers, with each MPTCP flow consisting of four subflows, resulting in a total of 400K subflows. A subflow fails if either direction (uplink or downlink) is affected by a switch failure, but an MPTCP flow fails only when all its subflows are lost. Our results show that spine failure barely affects the results. When a leaf switch fails, less than 400 MPTCP flows fail because all their sub-flows are mapped to the same failed leaf switch out of four available leaf switches. When a ToR switch fails, more than half of the MPTCP flows fail because there are only two equal-cost ToR switches. These results match the theoretical expectation[3] and demonstrate the vulnerabilities of multi-path protocols to network failures, especially at the ToR level. In contrast, when P-ECMP is enabled, all MPTCP flows survive regardless of where the failure occurred, demonstrating the effectiveness of P-ECMP in enhancing the robustness of multi-path protocols.

Failure localization. Figure 8(a) illustrates the number of probes required to cover all available paths between a pair of host machines across various topologies. Previous ECMP-based mechanisms (such as Pingmesh [28]) unfortunately require 2x to 5x more probes compared to P-ECMP which enables the PTC of next hops. There are also proposals to use techniques like IP-in-IP to achieve similar functionality.

[3]Assume there are N switches at the network layer where failure occurs. That leaves $N-1$ healthy hops. As mentioned in § 2.1, the probability that a subflow avoids the failure is $\frac{(N-1)^2}{N^2}$, so its failure rate is $1-\frac{(N-1)^2}{N^2}=\frac{2N-1}{N^2}$. Assume there are K subflows. Given that an MPTCP flow fails only if all subflows fail, the survival rate of an MPTCP flow is $1-\frac{(2N-1)^K}{N^{2K}}$.



(a) Failure localization. (b) Packet spray.

Figure 8: More use case evaluation of P-ECMP.

However, these approaches incur significantly higher packet header overhead compared to P-ECMP (see Figure 6(b)).

Packet spraying. We evaluate packet spraying using a 30% Hadoop workload, monitoring ToR switch output queue lengths. Figure 8(b) reveals that random spraying causes uneven packet distribution at a microscopic level, with 99th percentile queue lengths ranging from 31KB to 70KB, likely degrading performance in low-latency environments. This imbalance worsens at the 100th percentile. In contrast, P-ECMP's PTC capability ensures a more balanced queue length distribution across ports, with most queues measuring 11KB at the 99th percentile and a maximum of only 15KB. This demonstrates superior load balancing. Further results confirm that P-ECMP consistently outperforms random spraying in terms of FCT, as detailed in Appendix D.

Segment routing. Finally, we address segment routing. Previous approaches to implementing segment routing on commodity switches typically rely on IP-in-IP or SRv6 protocols, both of which introduce significantly higher (3x-6x) header overhead compared to P-ECMP, as shown in Figure 6(b).

5.3 In Production

Over the past year, we have incrementally deployed P-ECMP's PTC with offset across multiple data centers. The rollout was seamless, with P-ECMP coexisting with existing workloads without disrupting or interfering with live traffic.

Application Recovery. We focus on Cloud Block Storage

Table 4: CBS application performance with P-ECMP (numbers before slashes) and without P-ECMP (numbers after slashes) under different network failures and configurations. #MDS refers to the number of Metadata Service daemons.

Failure Type	#MDS	IO Jitters				IO Hang			
		Occurrence (%)	Min (s)	Med (s)	Max (s)	Occurrence (%)	Min (s)	Med (s)	Max (s)
Leaf Switch Unidirectional Black Hole	0	**100** / 100	**1** / 5	**7** / 7	**9** / 9	**56** / 72	**1** / 1	**2** / 3	**3** / 6
	28	**100** / 100	**6** / 7	**7** / 8	**8** / 11	**60** / 68	**1** / 1	**2** / 3	4 / **3**
	36	**100** / 100	**6** / 9	**7** / 11	**11** / 12	**68** / 70	**1** / 1	**1** / 4	**4** / 6
Leaf Switch Bidirectional Black Hole	28	**100** / 100	**3** / 3	**4** / 4	**6** / 10	**52** / 56	**1** / 1	**3** / 2	**3** / 4
	36	**100** / 100	**3** / 3	**3** / 4	**6** / 8	**44** / 54	**1** / 1	**2** / 2	**4** / 4
ToR Switch Unidirectional Black Hole	28	**100** / 100	**7** / 7	**10** / 15	**15** / 25	**90** / 98	**1** / 1	**5** / 8	**12** / 18

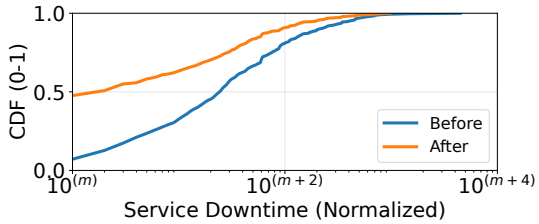


Figure 9: The distribution of service downtime by network failures in our production networks, before and after deploying P-ECMP.

(CBS) to illustrate the effectiveness of P-ECMP. CBS is a high-performance application that transmits large chunks of data across data center storage nodes. CBS may spawn multiple flows at the same time, and the completion of a CBS IO task usually incorporates a series of transport-layer flows. When a network failure occurs, CBS may encounter IO jitters or IO hang, *i.e.,* all the ongoing flows are lost, multiple times through its lifetime as the new flows spawned may be unfortunately mapped to the path with failure.

Before P-ECMP is introduced, the network failure recovery for CBS relies on FullPingmesh, a custom service similar to Pingmesh [28], and a custom socket-level failure detection and reconnection mechanism. FullPingmesh is able to detect and locate network failures within 20s. In addition, our custom socket-level failure detection for CBS can detect potential network failures by sending heartbeat packets and monitoring the socket buffer. When CBS considers that a network failure occurs and starts flow re-pathing by creating new TCP flows with different port numbers. According to our production log, the failure detection takes on average 1s.

Table 4 presents the CBS application performance with and without P-ECMP under different network failures and configurations, *i.e.,* different numbers of Metadata Server (MDS) daemons [1], where more MDS daemons generally help to serve larger workloads [2]. For each configuration and failure type, we repeat the experiments for 50 times. The results show that P-ECMP improves the application performance in most scenarios. Specifically, P-ECMP reduces the IO jitters minimum, median, and maximum durations by up to 80%, 36%, and 40%, respectively. P-ECMP also reduces the occurrence of IO hang by up to 16%, and the recovery duration for IO hang as well. The results demonstrate the effectiveness of P-ECMP in enhancing service availability.

Overall Impact. Our data centers have implemented and integrated multiple standalone solutions to detect and localize network failures, including active and passive probing, as well as data plane telemetry. Figure 9 shows the distribution of service downtime caused by network failures in our data centers. Many network failures can be mitigated within one minute. However, some failures such as hardware failures, take orders of magnitude more – *e.g.,* tens of minutes or even hours – to recover, which falls short of our needs. For example, the switches may encounter optical module failures. This requires the on-site operator to manually inspect and replace the optical module. This may take hours or even days depending on the availability and schedules of the on-site operators. In the worst case, the monthly occurrence of such failures could surpass ten times per pod, lasting for more than 20% of the total operational time.

After deploying P-ECMP in our data centers, the overall downtime due to network failures significantly decreased, as shown in Figure 9. Most downtimes were reduced to within one minute, and tail downtimes were also significantly improved. It is noteworthy that the figure still shows some long downtimes after P-ECMP is deployed. This is because these failures are multi-point or global network failures, *e.g.,* campus electricity outages, and are out of the capacity of P-ECMP or any flow re-pathing signals. Overall, P-ECMP is effective in production and meets our expectations.

6 Discussion

Dual-homed server bonding architecture. We adopted a dual-homed server bonding architecture as in § 4. It is noteworthy that this is not a strict requirement for P-ECMP, as P-ECMP can work with any topology. Instead, the dual-homed server architecture guarantees that a failure-free path is always available in case of any single-point failures in the network, and thus allows obtaining full benefits from P-ECMP. Indeed, when there is only one ToR connected to the host machines, P-ECMP cannot help in the presence of a ToR failure.

PTC policy with asymmetric topologies. When only the PTC of path offset is enabled, P-ECMP functions effectively in any topology, as long as there is more than one equal-cost output port available at each switch (otherwise P-ECMP is not useful). Appendix C details how to select the appropriate selector value for arbitrary topologies. However, enabling PTC of next hops introduces additional complexity. So far, our

discussion is restricted to tree-like topologies. P-ECMP can handle tree-like topologies with varying numbers of switches in different tiers. However, for non-tree-like topologies, such as mesh networks, hypercube topologies, or random graphs, where the concept of "network tiers" may not be as clear, the PTC policy requires more careful design. We leave further exploration of these scenarios to future work.

Interaction between P-ECMP and switch's local failover. For network failover issues, we note that commodity switches are often equipped with built-in local failover mechanisms, such as rapid link failure detection and fast reroute (FRR) protocols [17]. These mechanisms enable switches to quickly reroute traffic to alternative paths when a link or port failure is detected, often within milliseconds. As a result, not all failures require intervention from P-ECMP, which operates as an application-layer failover mechanism. However, certain types of errors fall outside the capabilities of commodity switches' local failover (though programmable switches can resolve some of these issues [23, 55]). Examples include silent packet drops and gray failures [31, 32], which are difficult to detect and mitigate using traditional failover mechanisms. Furthermore, P-ECMP addresses a broader range of scenarios beyond failover, such as congestion avoidance and load balancing. By operating at the application layer, P-ECMP complements the switch's local failover and provides a more comprehensive solution for ensuring network reliability and performance.

7 Related Work

ECMP and host-based path control. To defend against the randomness of ECMP and enable host-based path control, previous studies have mainly focused on removing ECMP, typically by redesigning or using different routing protocols [30, 33, 54]. For instance, XPath [30] identifies all end-to-end paths, compresses them, and pre-installs them into the IP TCAM tables of commodity switches. However, this approach requires a large number of forwarding table entries, which becomes impractical at the scale of data centers. More recently, programmable switches provides another solution for explicit host control. Nevertheless, their deployment in production networks remains limited so far.

Other studies attempt to work within the constraints of ECMP. For example, Volur [65] uses a centralized controller to collect routing information from switches, predict network behaviors, and send these predictions to servers. However, this approach faces two major issues: limited scalability and inaccurate predictions. RePaC [66] leverages the linearity of hashing functions to achieve relative path control. Unfortunately, this method assumes uniform switch deployment, making it impractical in production environments with diverse switches and hashing functions.

Our approach, P-ECMP, aligns more closely with the second group of studies. However, we leverage a different feature – additional ECMP groups – to extend ECMP's capabilities, allowing for precise traffic control. P-ECMP requires acceptable resources at switches and adapts to heterogeneous switch environments. Notably, P-ECMP has been successfully deployed in production.

Data center network failure. Existing proposals for network failure detection are mostly based on active or passive probing [11, 19, 20, 28, 42, 63], or data plane telemetry monitoring and analysis [18, 22, 25, 38, 41, 46, 52, 68, 69]. After detecting and locating the network failures, data center operators then need to mitigate the failures. Various automated failover solutions have been proposed based on SDN [36, 37, 43, 53, 58, 59, 67].

Nevertheless, the first-detection-then-mitigation fashion of failover is inherently slow. To speed up this process, another line of work proposes to mitigate the failures before accurately locating them. For instance, NetPilot [57] proposes to rely on the data center redundancy and blindly apply mitigation methods. Fast failovers can also be done by the host machines alone. For example, MPTCP [16] or MPRDMA [40] can be employed to address the network failures in data centers [50] by actively adjusting its sub-flows to circumvent network failures. In single flow scenarios, PRR circumvents the failure by re-pathing the flow leveraging the ECMP capability of data center networks [56]. Nevertheless, all the above solutions are vulnerable to hashing collisions. P-ECMP enhances fast failover, multipath protocols, and many other network functions by enabling precise traffic control.

8 Conclusion

This paper introduces programmable ECMP (P-ECMP), a programming model that achieves precise traffic control (PTC) by reconfiguring ECMP groups on commodity switches. P-ECMP enables PTC for both path offset and exact next hops, extending its support to multiple use cases, from network failover to segment routing, without disrupting the majority of traffic flowing through standard ECMP forwarding. Our evaluations demonstrate that P-ECMP requires minimal resources at switches and outperforms several existing approaches across various use cases. P-ECMP has been deployed in production for network failover, where it has proven effective in significantly reducing failover duration.

Acknowledgements

We sincerely thank our shepherd Kaihui Gao and anonymous reviewers for their insightful comments. We also thank software engineers at Tencent for their support in deploying P-ECMP in production. Jilong Wang was supported by the National Key Research and Development Program of China under Grant No. 2020YFE0200500. Congcong Miao is the corresponding author.

References

[1] ceph-mds – ceph metadata server daemon. https://docs.ceph.com/en/latest/man/8/ceph-mds/.

[2] Chapter 2. Configuring Metadata Server Daemons. https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/3/html/ceph_file_system_guide/configuring-metadata-server-daemons.

[3] eBPF - Introduction, Tutorials & Community Resources. https://ebpf.io/.

[4] modprobe(8) - Linux man page. https://linux.die.net/man/8/modprobe.

[5] networking:bonding. The Linux Foundation DokuWiki. https://wiki.linuxfoundation.org/networking/bonding.

[6] ns-3 | a discrete-event network simulator for internet systems. https://www.nsnam.org/.

[7] opencomputeproject/SAI. https://github.com/opencomputeproject/SAI.

[8] setsockopt(3) - Linux man page. https://linux.die.net/man/3/setsockopt.

[9] Software for Open Networking in the Cloud (SONiC). https://github.com/anish-n/SONiC/.

[10] Traffic Generator. HPCC-PINT/traffic_gen. https://github.com/ProbabilisticINT/HPCC-PINT/tree/master/traffic_gen.

[11] A. Adams, P. Lapukhov, and J. H. Zeng. Netnorad: Troubleshooting networks via end-to-end probing. *Facebook White Paper*, 2016.

[12] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM'08, Seattle, WA, USA, August 17-22, 2008*, pages 63–74. ACM, 2008.

[13] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *Proceedings of ACM SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 503–514. ACM, 2014.

[14] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of ACM SIGCOMM'10, New Delhi, India, August 30 -September 3, 2010*, pages 63–74. ACM, 2010.

[15] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: probabilistic in-band network telemetry. In *Proceedings of ACM SIGCOMM'20, Virtual Event, USA, August 10-14, 2020*, pages 662–680. ACM, 2020.

[16] O. Bonaventure, C. Paasch, and G. Detal. Use cases and operational experience with multipath TCP. *RFC*, 8041:1–30, 2017.

[17] Cisco. IPv4 Loop-Free Alternate Fast Reroute. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_pi/configuration/15-s/iri-15-s-book/iri-ip-lfa-frr.pdf.

[18] B. Claise. Cisco systems netflow services export version 9. *RFC*, 3954:1–33, 2004.

[19] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. J. Luckie, R. K. P. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy. Inferring persistent interdomain congestion. In *Proceedings of ACM SIGCOMM'18, Budapest, Hungary, August 20-25, 2018*, pages 1–15. ACM, 2018.

[20] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Netdiagnoser: troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2007, New York, NY, USA, December 10-13, 2007*, page 18. ACM, 2007.

[21] A. A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pages 2130–2138. IEEE, 2013.

[22] B. Eriksson, P. Barford, R. A. Bowden, N. G. Duffield, J. Sommers, and M. Roughan. Basisdetect: a model-based network event detection framework. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 451–464. ACM, 2010.

[23] K. Gao, S. Wang, K. Qian, D. Li, R. Miao, B. Li, Y. Zhou, E. Zhai, C. Sun, J. Gao, D. Zhang, B. Fu, F. Kelly, D. Cai, H. H. Liu, Y. Li, H. Yang, and T. Sun. Dependable virtualized fabric on programmable data plane. *IEEE/ACM Trans. Netw.*, 31(4):1748–1764, 2023.

[24] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. Juggler: a practical reordering resilient network stack for datacenters. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 20:1–20:16. ACM, 2016.

[25] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM'11, Toronto, ON, Canada, August 15-19, 2011*, pages 350–361. ACM, 2011.

[26] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of ACM SIGCOMM'09, Barcelona, Spain, August 16-21, 2009*, pages 51–62. ACM, 2009.

[27] D. Grossman. New terminology and clarifications for diffserv. *RFC*, 3260:1–10, 2002.

[28] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of ACM SIGCOMM'15, London, United Kingdom, August 17-21, 2015*, pages 139–152. ACM, 2015.

[29] K. Hsu, P. Tammana, R. Beckett, A. Chen, J. Rexford, and D. Walker. Adaptive weighted traffic splitting in programmable data planes. In *SOSR '20: Symposium on SDN Research, San Jose, CA, USA, March 3, 2020*, pages 103–109. ACM, 2020.

[30] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo. Explicit path control in commodity data centers: Design and applications. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 15–28. USENIX Association, 2015.

[31] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 1–16. USENIX Association, 2018.

[32] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 150–155. ACM, 2017.

[33] S. A. Jyothi, M. Dong, and B. Godfrey. Towards a flexible data center fabric with source routing. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, pages 10:1–10:8. ACM, 2015.

[34] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, pages 149–160. ACM, 2014.

[35] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo. Programmable in-network security for context-aware BYOD policies. In *USENIX Security 2020, August 12-14, 2020*, pages 595–612. USENIX Association, 2020.

[36] J. Li, J. Hyun, J. Yoo, S. Baik, and J. W. Hong. Scalable failover method for data center networks using openflow. In *2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014*, pages 1–6. IEEE, 2014.

[37] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zupdate: updating data center networks with zero loss. In *Proceedings of ACM SIGCOMM'13, Hong Kong, August 12-16, 2013*, pages 411–422. ACM, 2013.

[38] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 399–412. USENIX Association, 2013.

[39] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 101–114. ACM, 2016.

[40] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda. Multi-path transport for RDMA in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 357–371. USENIX Association, 2018.

[41] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM'11, Toronto, ON, Canada, August 15-19, 2011*, pages 290–301. ACM, 2011.

[42] P. Marchetta, A. Botta, E. Katz-Bassett, and A. Pescapè. Dissecting round trip time on the slow path with a single packet. In *Passive and Active Measurement - 15th International Conference, PAM 2014, Los Angeles, CA, USA,*

March 10-11, 2014, Proceedings, volume 8362 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 2014.

[43] C. Miao, Y. Xiao, M. Canini, R. Dai, S. Zheng, J. Wang, J. Bu, A. Kuzmanovic, and Y. Wang. TENSOR: lightweight BGP non-stop routing. In *Proceedings of ACM SIGCOMM'23, New York, NY, USA, 10-14 September 2023*, pages 108–121. ACM, 2023.

[44] C. Miao, Z. Zhong, Y. Xiao, F. Yang, S. Zhang, Y. Jiang, Z. Bai, C. Lu, J. Geng, Z. He, Y. Wang, X. Zou, and C. Yang. Megate: Extending WAN traffic engineering to millions of endpoints in virtualized cloud. In *Proceedings of ACM SIGCOMM'24, Sydney, NSW, Australia, August 4-8, 2024*, pages 103–116. ACM, 2024.

[45] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of ACM SIGCOMM'17, Los Angeles, CA, USA, August 21-25, 2017*, pages 15–28. ACM, 2017.

[46] T. Mizrahi, G. Navon, G. Fioccola, M. Cociglio, M. G. Chen, and G. Mirsky. AM-PM: efficient network telemetry using alternate marking. *IEEE Netw.*, 33(4):155–161, 2019.

[47] C. E. Perkins. IP encapsulation within IP. *RFC*, 2003:1–14, 1996.

[48] K. Qian, Y. Xi, J. Cao, J. Gao, Y. Xu, Y. Guan, B. Fu, X. Shi, F. Zhu, R. Miao, C. Wang, P. Wang, P. Zhang, X. Zeng, E. Ruan, Z. Yao, E. Zhai, and D. Cai. Alibaba HPN: A data center network for large language model training. In *Proceedings of ACM SIGCOMM'24, Sydney, NSW, Australia, August 4-8, 2024*, pages 691–706. ACM, 2024.

[49] M. A. Qureshi, Y. Cheng, Q. Yin, Q. Fu, G. Kumar, M. Moshref, J. Yan, V. Jacobson, D. Wetherall, and A. Kabbani. PLB: congestion signals are simple and effective for network load balancing. In *Proceedings of ACM SIGCOMM'22, Amsterdam, The Netherlands, August 22 - 26, 2022*, pages 207–218. ACM, 2022.

[50] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM'11, Toronto, ON, Canada, August 15-19, 2011*, pages 266–277. ACM, 2011.

[51] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of ACM SIGCOMM'15, London, United Kingdom, August 17-21, 2015*, pages 123–137. ACM, 2015.

[52] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. csamp: A system for network-wide flow monitoring. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 233–246. USENIX Association, 2008.

[53] B. E. Stephens and A. L. Cox. Deadlock-free local fast failover for arbitrary data center networks. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9. IEEE, 2016.

[54] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 599–614. USENIX Association, 2019.

[55] S. Wang, K. Gao, K. Qian, D. Li, R. Miao, B. Li, Y. Zhou, E. Zhai, C. Sun, J. Gao, D. Zhang, B. Fu, F. Kelly, D. Cai, H. H. Liu, and M. Zhang. Predictable vfabric on informative data plane. In *Proceedings of ACM SIGCOMM'22, Amsterdam, The Netherlands, August 22 - 26, 2022*, pages 615–632. ACM, 2022.

[56] D. Wetherall, A. Kabbani, V. Jacobson, J. Winget, Y. Cheng, C. B. M. III, U. Moravapalle, P. Gill, S. Knight, and A. Vahdat. Improving network availability with protective reroute. In *Proceedings of ACM SIGCOMM'23, New York, NY, USA, 10-14 September 2023*, pages 684–695. ACM, 2023.

[57] X. Wu, D. Turner, C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of ACM SIGCOMM'12, Helsinki, Finland - August 13 - 17, 2012*, pages 419–430. ACM, 2012.

[58] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks, Philadelphia, PA, USA, November 16 - 17, 2015*, pages 26:1–26:7. ACM, 2015.

[59] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 719–733. USENIX Association, 2017.

[60] J. Xing, K. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked*

Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022, pages 651–665. USENIX Association, 2022.

[61] Y. Xu, K. He, R. Wang, M. Yu, N. Duffield, H. M. G. Wassel, S. Zhang, L. Poutievski, J. Zhou, and A. Vahdat. Hashing design in modern networks: Challenges and mitigation techniques. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 805–818. USENIX Association, 2022.

[62] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of ACM SIGCOMM'20, Virtual Event, USA, August 10-14, 2020*, pages 126–138. ACM, 2020.

[63] H. Zeng, R. Mahajan, N. McKeown, G. Varghese, L. Yuan, and M. Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. *Mountain Safety Res., Seattle, WA, USA, Rep. MSR-TR-2015-55*, 2015.

[64] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1187–1204. USENIX Association, 2020.

[65] Q. Zhang, D. Zhuo, V. Liu, P. Lapukhov, S. Peter, A. Krishnamurthy, and T. E. Anderson. Volur: Concurrent edge/core route control in data center networks. *CoRR*, abs/1804.06945, 2018.

[66] Z. Zhang, H. Zheng, J. Hu, X. Yu, C. Qi, X. Shi, and G. Wang. Hashing linearity enables relative path control in data centers. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 855–862. USENIX Association, 2021.

[67] W. Zhou, J. Croft, B. Liu, and M. Caesar. Neat: Network error auto-correct. In *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, pages 157–163. ACM, 2017.

[68] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *Proceedings of ACM SIGCOMM'20, Virtual Event, USA, August 10-14, 2020*, pages 76–89. ACM, 2020.

[69] Y. Zhu, N. Kang, J. Cao, A. G. Greenberg, G. Lu, R. Mahajan, D. A. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of ACM SIGCOMM'15, London, United Kingdom, August 17-21, 2015*, pages 479–491. ACM, 2015.

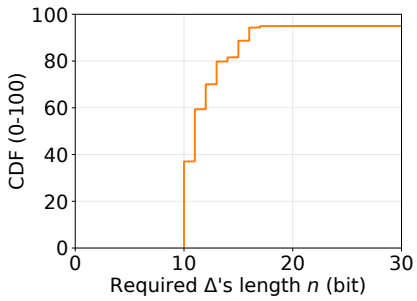


Figure 10: Required Δ's length to cover 64 paths.

A Evaluation of RePaC In Production

RePaC [66] introduces modifications to header fields by XORing a n-bit Δ. This generates a *pathmap* of 2^n entries where each possible Δ corresponds to an offset $Hash(\Delta)$. The output offset is subject to the number of available next hops. Assume the number of next hops is 2^m, then $Hash(\Delta)$ has a length of m bits. Let O be the matrix of all the outputs with n-bit Δ as input, where each $O_{i,j}$ represents the value of jth index of $Hash(\Delta_i)$ for $i \in [1, 2^n]$. To control the offset across all possible alternative paths, it requires $\text{rank}(O) = m$, meaning the hash results of all n-bit Δ must cover all 2^m possible paths. When there are multiple K layers of switches in the network with $2^{m_1}, 2^{m_2}, \ldots, 2^{m_k}$ switches per layer, then it requires $\text{rank}(O) = \sum_{i=1}^{K} m_i$, ensuring coverage of all possible path offsets across all switch layers.

We evaluated this requirement in a heterogeneous environment using all commonly used hash algorithms in commodity switches, including 8 variants of CRC-8, 8 variants of CRC-16, and 8 variants of CRC-32. We used topology #12, which mirrors our production network, with the same type of switches within each layer (e.g., ToR, leaf, or spine) but different switches across layers. We tested all permutations of hash algorithms across layers, totaling permutation(24,3)=12,144 settings, with different hashing algorithms assigned to different network layers.

Figure 10 shows the CDF of the required Δ's bit length n to cover all available paths across three network layers. The minimum bit length of Δ is 10 (1 bit needed for 2 ToR switches per server, 3 bits for 8 leaf switches per pod, and 6 bits for 64 spine switches). While 20 bits suffice in most settings, in 608 out of 12,144 (roughly 5%) switch permutations, no suitable Δ exists to cover all paths (tested with up to 1024-bit inputs, *i.e.,* $n = 1024$). In other words, RePaC fails if the deployed switches use hashing algorithms from these 608 settings. If switches are purchased randomly, there is a 5% chance that RePaC will not work. Moreover, if heterogeneity across pods

Table 5: Hashing collision rate between different hashing algorithms. Three numbers in each cell represent collision rates when there are 2, 4, and 8 equal-cost next hops, respectively.

Collison Rate (%)	Pearson	CRC	XOR	Random
Random	50.00 / 25.00 / 12.50	49.99 / 25.00 / 12.49	50.01 / 24.98 / 12.50	-
XOR	50.00 / 25.00 / 12.50	50.00 / 25.00 / 12.50	-	
CRC	50.01 / 24.97 / 12.50	-		
Pearson	-			

Table 6: The obtained maximum path load using selected selector set and the differences between it and the optimal (D.O.) or P-ECMP-allowable optimal (D.D.O) loads for an arbitrary network with a maximum of 4 equal-cost next hops.

$n\backslash p$	5	7	Maximum Path Load (%) Obtained	D.O.	D.D.O.
2	1	1	50	0	0
3	2	1	66.67	0	0
4	1	3	66.67	8.33	0

is considered, it will require longer Δ and increase the number of switch settings where no suitable inputs exist.

B Alternative of Changing Hashing Algorithm

We investigate the four most common hashing algorithms used in ECMP [66]: random-based, XOR, CRC, and Pearson hashing. The input to the hashing algorithms in ECMP is usually the five-tuple <Src_IP, Dst_IP, Src_Port, Dst_Port, Proto>, which includes 104 bits. However, given the huge input space, *i.e.,* 2^{96} possibilities for 104-bit inputs, we only select a random set of four-tuple inputs. We then feed them into the hashing algorithms respectively, and check whether the outputs would be the same. If two hashing algorithms produce identical outputs for the same input, switching between them would not alter path selection, rendering it incapable of achieving PTC of offset.

Table 5 illustrates the probabilities that two different hashing algorithms produce the same outputs for a random set of five-tuple inputs, *i.e.,* the re-pathing failure rates. The results show that the re-pathing failure rates are substantially high – 50%, 25%, and 12.5% when there are 2, 4, and 8 equal-cost next hops, respectively. This means that the PTC of offset cannot be guaranteed by swamping among any of these common hashing algorithms.

C Load balance After Flow Re-Path

Here, we discuss how to maintain load balancing for flows after they are re-pathed for failover, leveraging P-ECMP's PTC

Table 7: The obtained maximum path load using selected selector set and the differences between it and the optimal (D.O.) or P-ECMP-allowable optimal (D.D.O) loads for an arbitrary network with a maximum of 8 equal-cost next hops.

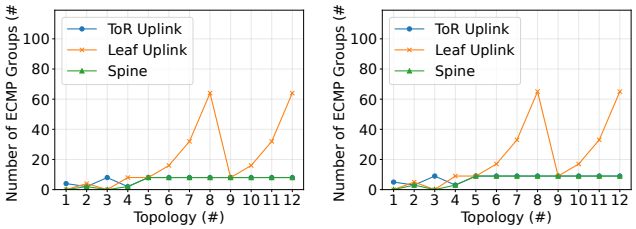
$n\backslash p$	11	13	17	19	23	29	Maximum Path Load (%) Obtained	D.O.	D.D.O.
2	1	1	1	1	1	1	50	0	0
3	2	1	2	1	2	2	60	6.67	6.67
4	3	1	1	3	3	1	66.67	8.33	0
5	1	3	2	4	3	4	75	5	5
6	5	1	5	1	5	5	60	23.33	6.67
7	4	6	3	5	2	1	85.71	0	0
8	3	5	1	3	7	5	75	12.5	5

of offset capability. We assume that the PTC of next hops is not simultaneously supported, and that servers have limited knowledge of the network topology. Otherwise, servers could simply select from the remaining available paths evenly, ensuring that all re-pathed flows are distributed uniformly across the paths.

First, if the bit efficiency of the selector s is crucial and the network topology follows a symmetric Clos-like structure – which is adopted by most data centers –, the servers should select selector s from the set of positive odd numbers less than the largest ECMP group size in the network. For instance, if the spine tier has 8 switches, then the selector s should be selected from the set $\{1,3,5,7\}$ with equal probability. In this case, selector s needs only three bits, providing good bit efficiency. It also guarantees that the load will be balanced across half of the remaining available switches in each network tier. It is important to only select odd numbers because the other network tiers may contain switches of even numbers, *e.g.,* 2 or 4, and selecting an even number could result in failed re-pathing at some network tiers.

Second, if the network topology is asymmetric/less structured, the host should select selector s from at most $N-1$ prime numbers greater than N, where N is the largest ECMP group size in the network. For example, if $N=8$, then the selector s should be selected from the set $\{11,13,17,19,23,29\}$ with equal probability. In this case, selector s occupies 5 bits, providing the capability to avoid any single-point failures and *good* load-balancing performance on most network tiers with a size equal to or less than N. We detail more analysis below.

We assume an arbitrary ECMP network where the largest ECMP group size is N. If the ability to circumvent single-point network failure is desired, then it means that there might be network tiers with the ECMP group size n to range from 2 to N, *i.e.,* $n \in [2,N]$. In such a network, our goal is to find a set of selector S that satisfies two conditions: (i) the re-pathing succeeds circumventing the network failure on the old path, *i.e.,* $\forall s \in S, p \bmod n \neq 0$; and (ii) good load balance is achieved after the flow re-pathing. Note that the minimum ECMP group size is 2 because we assume a dual-homed architecture (§ 4). If a traditional fat-tree topology is adopted,



(a) PTC of offset only. (b) PTC of hop only.

Figure 11: Number of ECMP groups with either PTC type.

the minimum ECMP group size is 1 at the ToR switch tier, meaning that flow re-pathability cannot ensure a successful failover from a ToR switch failure or link failure between the server and the ToR switch.

To quantify the second condition, let $D_n = \{s \bmod n, \forall s \in S\}, \forall n \in [2,N]$. A perfect load balance after flow re-pathing based on S means that $\forall n \in [2,N], D_n$ needs to be perfectly balanced, *i.e.,* every possible element has the same frequency. This might involve a large selector set S. In fact, the minimum size of S to achieve perfect load balance is the least common multiple of all prime numbers ranging from 2 to N. For example, with $N = 8$, the minimum $|S| = 2 \times 3 \times 5 \times 7 = 210$. To achieve a perfect balance an ECMP group of size n requires the number of selectors $|S|$ to be the product of its group size and any positive integer, *i.e., kn* where $k \in \mathbb{Z}^+$. If n is not a prime number, then it is the product of several prime numbers smaller than it. As long as $|S|$ can achieve perfect balance for its prime factors, $|S|$ can achieve perfect balance for n.

To find the set S for perfect load balance, one should start with the set of smallest prime numbers that are greater than N, *e.g.,* $\{11,13,17,\ldots\}$ if $N = 8$. This set should contain $|S|$ prime numbers. It follows that $\forall n \in [2,N], D_n$ can be calculated accordingly. If D_n is not balanced, then one should adjust the prime numbers in the set. The above process should be repeated until S is found. Some intuition might be helpful to optimize the prime number adjusting, but we do not go into details here as we do not deem the perfect load balance after flow re-pathing to be necessary given its high cost.

Rather, it may be more practical to target a smaller selector set to ensure *satisfactory* post-re-pathing load balance, rather than striving for perfection. This trade-off allows for a "good enough" load balance outcome while mitigating the high costs associated with the above prime number searching.

To achieve this objective, we propose a straightforward approach: simply select the subsequent $N_1 - 1$ prime numbers that are greater than N for the selector set, where N_1 represents the largest prime number less than or equal to N. For instance, when N is 8, the highest prime number fitting this criterion is $N_1 = 7$. Consequently, we arrive at $S = \{11,13,17,19,23,29\}$ where $|S| = 6$.

Assuming that load balance is established prior to the occurrence of any failure — meaning that all paths have identical loads — we define the degree of post-re-pathing load balance by examining the maximum load on each path. Employing the example provided, $D_4 = \{3,1,1,3,3,1\}$. In the scenario where one path becomes unavailable, its load is evenly distributed across two other paths. This results in the maximum load on each path reaching 66.67%.

For a state of perfect load balance, implying that the load on the failed path can be evenly redistributed across all other paths, the maximum load on each path can be up to 75%. Consequently, the disparity between our solution and the optimal path load equates to 8.33%.

However, it is crucial to acknowledge that the value 2 can never be attained via modulo 4 with any prime numbers. Consequently, we introduce the concept of P-ECMP-allowable degree of load balance by eliminating paths that cannot be derived through the modulo operation involving prime numbers. Returning to the aforementioned case, two paths can be re-pathed using P-ECMP. Hence, the difference between our solution and the optimal P-ECMP-allowable path load is 0%.

Table 6 and 7 illustrate the post-re-pathing load balance where N is 4 and 8, respectively. Overall, we find that the above straightforward approach demonstrates a good enough post-re-pathing load balance that meets our use case requirements adequately, with the disparity between the obtained and the optimal P-ECMP-allowable maximum path load remaining under 6.67%.

It is noteworthy that unlike RePaC which requires a central server to calculate the comprehensive path map and subsequently distribute it across all data center servers, the algorithm outlined above for ascertaining the optimal selector set S can be executed on any local server. Furthermore, each local server retains the freedom to determine its own desired properties, such as pursuing a state of perfect load balance or settling for a reasonably good load balance. More crucially, RePaC struggles to address scenarios involving an asymmetric network topology. In contrast, P-ECMP demonstrates the capacity to handle arbitrary network topologies and offers varying levels of load balance that can be fine-tuned against the size of the selector set.

D Additional Evaluation of P-ECMP

Figure 11 presents the number of ECMP groups with either PTC type.

Table 8 shows the normalized FCT with random packet spray and with P-ECMP.

E Selector Length

We examine the length of the selector when supporting either PTC type or both concurrently. For the sake of simplicity, we assume a small topology with one ToR switch connected to each server, four leaf switches, and two spine switches. Figure 12 shows the ECMP groups and members when different PTCs are supported.

Table 8: Normalized flow completion time with packet spray. LF stands for last flow (*i.e.,* 100th percentile).

Trace	LB Scheme	30% Load			50% Load			80% Load		
		p50	p99	LF	p50	p99	LF	p50	p99	LF
Web Search	Random Packet Spray	1	11.4	45.7	1	57.8	412.8	1	54.8	1,155.4
	P-ECMP	**0.98**	**10.0**	**45.1**	**0.84**	**56.0**	**396.7**	**0.98**	**47.5**	**1,071.8**
Hadoop	Random Packet Spray	1	57.8	412.8	1	166.5	2,062.1	1	27.8	475.2
	P-ECMP	**0.84**	**56.0**	**396.7**	**0.81**	**151.2**	**2,049.0**	**0.99**	**27.1**	**340.5**

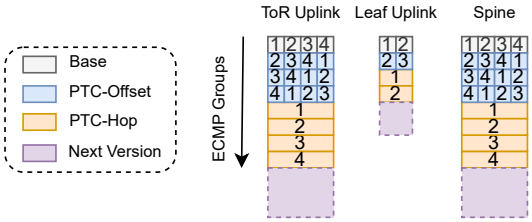


Figure 12: Forwarding table design.

When only the PTC of offset is required, the selector can range from 0 to 3 (for more details, see Appendix C). This value is effective across all network hops. For example, when $s = 1$, the flow will be re-pathed through a different leaf, spine, and another leaf switch in a different pod. As a result, the selector length is 2 bits. Our transactional update mechanism (§ 3.3) doubles the required space, increasing the selector length to 3 bits.

When only the PTC of next hops is needed, the situation changes, as different selector values are required at different switches. Without ECMP groups for PTC of offset, we find that 5, 3, and 5 selector values are required at the ToR, leaf, and spine switches, respectively, to specify the exact next hops. Therefore, the selector length is $\lceil \log(5) \rceil + \lceil \log(3) \rceil + \lceil \log(5) \rceil = 8$ bits. Including the transactional update, the selector length increases to 9 bits.

When both PTC types are supported, we again evaluate each switch individually. This results in 8, 4, and 8 selector values needed at the ToR, leaf, and spine switches, respectively. Thus, the selector length is $\lceil \log(8) \rceil + \lceil \log(4) \rceil + \lceil \log(8) \rceil = 8$ bits. Accounting for the transactional update, the selector length remains 9 bits, the same as when only the PTC of next hops is supported.

Enabling Portable and High-Performance SmartNIC Programs with Alkali

Jiaxin Lin* UT Austin
Zhiyuan Guo* UCSD
Mihir Shah† NVIDIA
Tao Ji† Microsoft
Yiying Zhang UCSD
Daehyeok Kim UT Austin
Aditya Akella UT Austin

Abstract

Trends indicate that emerging SmartNICs, either from different vendors or generations from the same vendor, exhibit substantial differences in hardware parallelism and memory interconnects. These variations make porting programs across NICs highly complex and time-consuming, requiring programmers to significantly refactor code for performance based on each target NIC's hardware characteristics.

We argue that an ideal SmartNIC compilation framework should allow developers to write target-independent programs, with the compiler automatically managing cross-NIC porting and performance optimization. We present such a framework, Alkali, that achieves this by (1) proposing a new intermediate representation for building flexible compiler infrastructure for multiple NIC targets and (2) developing a new iterative parallelism optimization algorithm that automatically ports and parallelizes the input programs based on the target NIC's hardware characteristics.

Experiments across a wide range of NIC applications demonstrate that Alkali enables developers to easily write portable, high-performance NIC programs. Our compiler optimization passes can automatically port these programs and make them run efficiently across all targets, achieving performance within 9.8% of hand-tuned expert implementations.

1 Introduction

SmartNICs [3, 11, 15, 33, 34, 36] have become a popular platform for hosting various tasks in datacenters [5, 10]. They enhance traditional NICs by incorporating programmable compute units, a memory hierarchy, and (in some cases) domain-specific accelerators (*e.g.,* crypto engine). These additional resources enable the NICs to serve various functionalities, spanning network functions [10, 26], transport offloads [4, 50, 59], and non-network infrastructure processing [19, 25]. Many works have also offloaded application-specific computation onto the NICs (*e.g.,* data analytics [29, 31], RPC orchestration [14, 28, 58] and serverless [7]).

However, despite growing popularity, effectively programming SmartNICs remains challenging for two key reasons. First, NIC architectures vary significantly, so each vendor offers a custom language or SDKs tailored to its hardware architecture. They only provide architecture-specific low-level programming primitives, such as customized instruction sets to access memory and accelerators. At the same time, NIC programs often involve complex processing logic, *e.g.,* intricate control flow and stateful computations. Implementing such complex logic using vendor-provided low-level programming primitives is challenging—it essentially requires developers to have expertise with each vendor's hardware.

Second, even if one can implement them, optimizing program performance is challenging. The performance depends on numerous factors, such as identifying ideal parallelism and effectively using heterogeneous memory specific to a target NIC. Developers must manually tweak programs based on their understanding of detailed target-specific hardware characteristics which is arduous and error-prone.

Trends indicate that SmartNICs from different vendors or generations from the same vendor exhibit significant variations in hardware parallelism and memory interconnects to cater to the diverse needs of cloud infrastructures, applications, and workloads. Thus, as datacenter operators upgrade or change their NIC hardware the above challenges become even more pronounced requiring not only (re)writing programs but also (re)tuning performance for each new SmartNIC iteration. Existing NIC programming frameworks, including P4 extensions and recent proposals [6, 26, 44, 56] while extremely valuable, are narrowly built for a single type of architecture – thus, they do not offer the much-needed portability.

We present Alkali, a compilation framework enabling portable and high-performance NIC programs. With Alkali, developers use high-level languages (*e.g.,* a subset of C) to write a single-threaded, run-to-completion NIC program. Then, our compiling toolchain transforms the program into an optimized, pipeline- and data-parallel program aligned with

*Both authors contributed equally to the paper.
†Author's contributions were made while at UT Austin.

the target NIC's hardware characteristics.

Our paper makes the following contributions to realize Alkali and showcase its benefits:

• **An IR targeted at SmartNICs:** Inspired by existing compiler frameworks like LLVM [22] that recognize the importance of an intermediate representation (IR) for supporting multiple targets, we design an IR called α*IR* aimed at SmartNICs, their architectural nuances and SmartNIC programs' common structures. α*IR* generalizes and captures how the inherent parallelism in NIC programs' computations and salient aspects of their memory access patterns map across diverse SmartNICs. To this end, α*IR* uses the *stateful parallel network handler graph* abstraction (§4).

• **A two-phase iterative algorithm for parallelism optimizations:** Translating the α*IR* into parallel code that optimally leverages the underlying hardware is challenging. This is due to the large space of potential parallelization – *e.g.,* the many ways of splitting NIC programs into pipeline stages, replication options per stage, possible mappings between replicas/stages and compute units, and placements of different replicas'/stages' state onto the target's memory hierarchy. To efficiently navigate the space and identify good parallelization plans, we model the problem as a two-phase iterative optimization (§5). The first identifies suitable pipeline stages via a novel weighted min-cut formulation to determine the places to partition programs (in α*IR*) into pipeline stages. The second employs an SMT solver alongside a simple NIC performance model to determine the number of replicas of each stage and the placement of replicas and state (onto compute cores and memory layers, respectively) to maximize performance.

• **Open-source prototype:** We present a full end-to-end implementation of the Alkali compiler toolchain including a C frontend, a mid-end, and multiple backends that support four types of NIC targets–on-path SoC NICs (Agilio [36]), off-path SoC NICs (BlueField-2 [34]), FPGA NICs [3], and specialized ASIC SmartNICs (PANIC [30]). We open-source the prototype [2].

Our evaluation using a wide range of NIC applications demonstrates that our framework enables developers to easily write portable, high-performance NIC programs. The compiled programs run efficiently across different NIC targets - their performance is competitive with experts' implementations, showing less than a 9.8% gap.

2 Background and Motivation

SmartNICs can be viewed generally as comprising a *subset* of three types of components: (1) programmable compute units, (2) a memory hierarchy, and (3) reconfigurable match-action table (RMT) pipelines. Of these, RMT pipelines have received substantial attention from a programming point of view, with P4 [55] being a framework ideally suited to it, with important P4-based NIC programs already used in many production settings today [24, 27, 42, 57].

The architecture of the remaining "non-RMT" components has significant heterogeneity (Figure 1): (1) *Programmable compute units* have substantial differences in micro-architecture and degrees of parallelism. For instance, the BlueField-2 DPU has 8 ARM cores, while Agilio boasts 48 micro-engines, Pensando's Elba contains 256 MPUs, and FPGAs can have hundreds of hardware modules running in parallel; (2) *Memory hierarchies* have distinct sharing scopes, sizes, and access latency. For example, Agilio has CAM, CLS, CTM, and EMEM [37], Elba has SRAM and TCAM [11], while FPGAs have SRAM, BRAM, and DRAM [3]. They are software-managed, non-coherent, and require careful selection for placing stateful program objects.

2.1 NIC Programming Challenges

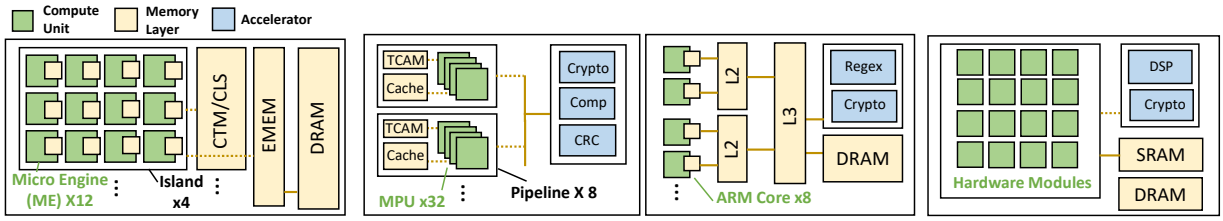
Programmability is a key driving factor of the broader adoption of SmartNICs. However, programming today's SmartNICs remains arduous and time-consuming. First, the NICs' hardware complexity and low-level programming interfaces create a steep learning curve. Second, hardware heterogeneity exacerbates the challenge, making programming and learning efforts non-portable across different NICs.

2.1.1 Target-specific Primitives

NIC vendors today provide a language and toolchain tailored to their NIC. For example, developers use Verilog for FPGA NICs, DOCA SDK for BlueField-2 [38], Micro-C (a subset of C++ with the vendor-provided library) for Agilio [37], and a mixture of customized P4 with hand-coded assembly for Elba [11]. They offer low-level target-specific primitives that are tightly coupled with the NIC architecture. For example, when writing the network transport offload (Flextoe [50]) on Agilio (the program logic shown in Figure 2), developers need to break down a sequence of packet processing tasks into fine-grained micro threads and determine how to replicate and map these micro threads onto 48 micro engines across 4 compute islands [37, 50]. Then, since the transport offload maintains a flow table, developers need to explicitly define which memory layer this table's content is placed in using the customized C++ macro (*e.g.,* `__declspec(emem)`). A similar tedious programming process is needed for FPGA NICs [26] and Elba NICs [5].

2.1.2 Lack of Portability

Portability between different NIC targets is essential for reducing vendor lock-in and allowing network operators to switch between NIC targets as workloads evolve. However, today, developers must follow a two-step transformation process to adapt architecture-dependent low-level programs across different NICs, making portability challenging.



(a) **Netronome Agilio [36]** (b) **Pensando Elba [11]** (c) **NVIDIA BlueField-2 [34]** (d) **AMD Alveo [3]**

Figure 1: **An abstract view of different SmartNICs' architecture.**

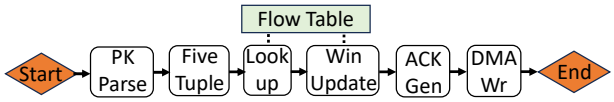
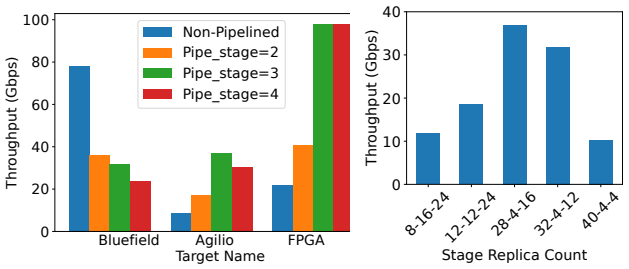


Figure 2: **FlexTOE's packet processing flow graph [50].**



(a) **Impact of pipeline parallelism.** (b) **Per-stage replica count.**

Figure 3: **Different parallelism influences FlexTOE [50]'s performance on different arch. Using 16 connections with 256B packets.**

Step 1: Translate primitives. Developers must resolve the impedance mismatch between different NICs' low-level hardware primitives. For example, the Figure 2's packet processing logic will be implemented as a thread on an ARM core on BlueField-2, whereas on FPGAs, it will be implemented using combinational and sequential logic on LUTs. Similarly, the flow table can be implemented in CAM on Agilio but will be implemented as a hash table on BlueField-2 due to the lack of CAM hardware support. When performing primitive translation, developers need to find primitives that provide the same or equivalent functionality. Such translation may not always exist because of target-specific functional constraints (*e.g.,* supported accelerators and hardware interfaces).

Step 2: Refactoring. Developers also need to significantly refactor and re-optimize their code to make it run fast based on the target NIC's hardware characteristics:

(1) Change code parallelism. SmartNICs are designed with extensive hardware parallelism, making them ideal for highly parallel network tasks. NIC programs usually leverage two forms of parallelism: data parallelism, where the same code block runs across multiple compute units with packets or flows distributed via RSS [35], and pipeline parallelism, where the code is divided into stages that communicate in a pipelined fashion. The optimal parallelism strategy varies across NICs, requiring developers to adjust the code's parallelism when porting. This involves determining (1) the number of pipeline stages and (2) how many replicas to use per stage.

To illustrate the need for such refactoring, we implemented FlexToe [50] in Figure 2 on three platforms - BlueField-2, Agilio, and FPGA - using two approaches: (1) a non-pipelined data parallel-only version, where the entire program is replicated across all compute units and packets are steered to compute units using flow steering, and (2) pipelined versions, where the program is divided into stages, and each stage is replicated independently.

Figure 3a shows that the optimal number of pipeline stages varies by architecture. On Agilio and FPGA, three pipeline stages yield the best performance, while on BlueField-2, the non-pipelined version performs better. These differences arise because Agilio and FPGA both (1) have more compute units, allowing pipeline stages to fully utilize available units when the active flow number limits data parallelism; (2) have specialized hardware queues to reduce the communication overhead between pipeline stages, and (3) are more sensitive to data locality - memory access latencies vary more across their memory layers, and pipelining helps improve locality.

Figure 3b shows that with pipeline parallelism, developers must also optimize the number of replicas per stage. For example, on Agilio, replicating a three-stage pipeline to optimally occupy its 48 micro-engines boosts performance up to 3.7×.

(2) Change state mapping. Many NICs [3, 11, 36] come with a memory hierarchy, and developers must select the appropriate memory level for storing states. This includes analyzing each program state's size and access patterns to place frequently accessed states in faster, lower-capacity memory layers. For example, our experiments showed that placing Figure 2's flow table into the fastest memory layer (CAM) resulted in an over 8× throughput improvement compared with placement in the slowest layer (EMEM).

2.2 Limitation of Existing Frameworks

An ideal SmartNIC programming framework should allow developers to write target-independent programs and let the compiler automatically port these programs across various NICs. However, existing frameworks [26, 44, 45, 56] fail to achieve this for the following reasons:

(1) Non-reusable compiler infrastructure and optimizations: Existing frameworks are usually only narrowly built

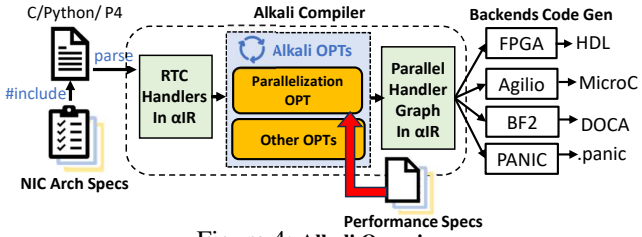


Figure 4: **Alkali Overview.**

for a specific NIC. For instance, Floem [44] is built for on-path SoC NICs; ClickNP, and Flowblaze [26,45] target FPGA NICs; P4 and its backend compilers [41,56] target RMT-like SmartNICs. Compiler optimizations in these frameworks are often not reusable across different NICs. For example, P4 backend compilers' optimizations focus on packing resources into constrained, fixed, line-rate RMT-like pipelines but lack performance-oriented optimizations such as pipeline stage replication and state mapping, which are crucial for SoC and FPGA NICs as shown in Figure 3.

(2) Lack of automatic parallelization: These compilers still require developers to manually define the degree of parallelism and cannot automatically change the program's pipeline/data parallelism degree when compiling programs to different NICs. For example, Floem requires developers to determine the program's pipeline stages and replication counts and rewrite the program when porting to another SoC NIC with different core counts.

These limitations stem from the tight coupling of programs and optimizations to specific NIC targets, which restricts portability across different architectures.

3 Alkali Overview

Overcoming the above limitations requires decoupling NIC programs (written in various languages) from a specific NIC architecture. Drawing inspiration from the role of intermediate representations (IR) in conventional software compilers, which bridge high-level code and architecture-specific implementations, we introduce α*IR*, a new IR tailored for NIC programming. α*IR* serves as a bridge between target-agnostic NIC programs and target-specific code generation.

Building on α*IR*, we develop Alkali, a multi-target SmartNIC compilation framework that (1) allows developers to write NIC programs in high-level languages, (2) uses α*IR* to translate between languages and vendor-specific code seamlessly, and (3) employs novel compiler optimization techniques to automatically port programs for high performance across various targets.

Figure 4 illustrates the workflow of Alkali. First, developers import the target NIC's architecture specification provided by NIC vendors, which specifies high-level architectural functionality constraints of the target NIC. They then write single-threaded, run-to-completion NIC programs (*e.g.,* Figure 6a) using high-level languages (*e.g.,* a subset of C). This interface frees programmers from having to manage hardware parallelism or memory hierarchies, enabling simpler and more portable code. These input programs are parsed and translated into α*IR* where the compiler performs parallelization optimizations that transform the single-threaded program into a highly optimized, pipeline- and data-parallel program based on the target NIC's characteristics. The parallelization optimization is guided by a simple performance specification provided by the NIC vendor. Finally, the backend takes the optimized α*IR* program and generates target-specific code.

3.1 Design Challenges and Ideas

Here, we describe the main design challenges in Alkali:

C1: IR design. An IR for NICs must preserve the parallelism inherent in NIC programs while also abstracting the details of their execution on diverse NIC hardware. Directly reusing a general-purpose IR like LLVM IR [22] will not work well since it is too low-level to suitably capture/express the semantics of SmartNICs' parallel execution patterns and memory hierarchies. This makes developing NIC-specific optimization passes hard. Creating an IR specific to a single type of NIC (*e.g.,* FIRRTL IR for FPGA [16], P4 IR for RMT NICs [55,56]) fails to represent other architectures adequately.

C2: Identifying parallelization plans. Transforming single-threaded NIC programs into pipeline-/data-parallel programs is challenging because there are numerous ways to divide a program into pipeline stages and replicate them. Parameters include the location of stage boundaries, the number of stages, and the stages' replication counts. Also, NIC programs are usually stateful, and careless parallelization can compromise state consistency and correctness. Alkali must efficiently navigate this expansive design space to find optimal solutions while maintaining the program's semantics.

To solve these challenges, we employ the following ideas:

I1: Stateful handler graph IR (§4). We observe that although NICs come with huge compute and memory heterogeneity (Figure 2), their architectures and program execution typically involve two main types of parallelism (pipeline and data) and three kinds of state access (local value, persistent table, and context state). Based on this, we propose the *stateful handler graph IR* that enables flexible compiler optimizations.

I2: Iterative mapping-then-cut search (§5). To manage the vast search space of parallelization plans, Alkali's optimizer mimics a developer's fine-tuning process: starting with a data-parallel, non-pipelined program, developers iteratively identify bottlenecks and apply pipeline cuts to increase stages and boost performance. Similarly, Alkali's optimizer operates in an iterative loop between two modules: (1) the *mapping engine* (§5.1), which determines optimal data parallelism strategies for the current pipeline and identifies bottleneck stages, and (2) the *cut engine* (§5.2), which splits the bottleneck into additional pipeline stages to improve throughput. The updated

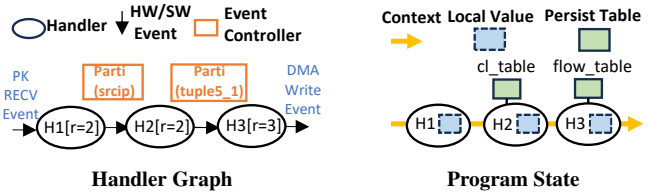


Figure 5: **Visualised αIR structure for Figure 6b.**

pipeline is then returned to the mapping engine for further refinement. Throughout this process, both modules carefully analyze program states to ensure that all cut and replication plans preserve program correctness.

4 Alkali IR

We design αIR with two goals: (1) Providing a general representation of the program's execution on heterogeneous architecture, capable of abstracting various types of computational parallelism and state access, and (2) Enabling the compiler to realize a broad set of architecture-agnostic transformations/optimizations. This section explains how αIR achieves them with an example input C function that implements connection limiter and transport [50] detailed in Figure 6a.

4.1 Abstracting Parallelism with Handlers

Observation. SmartNICs have a massive set of parallel compute units. Each compute unit can be viewed as an inseparable hardware block that logically executes code in sequence, such as Agilio's micro-engines, BlueField-2's ARM cores, and FPGA's hardware modules. NIC programs typically leverage two types of parallelism on these units: data parallelism and pipeline parallelism.

IR design. We propose to abstract a NIC program as a *handler graph* to capture parallelisms, in which each handler abstracts a code block running inside a single compute unit. In Figure 6b, the IR contains three handlers: H1 (line 4), H2 (line 15), and H3 (line 23), and Figure 5a visualizes the handler graph. Each handler can be replicated, and a number of replicas is specified in the IR (*e.g.,* [r=2] in line 4). Handlers use *events* to trigger another handler's execution (*e.g.,* lines 9 and 18 in Handlers H1 and H2). The start node of the graph is triggered by hardware events (*e.g.,* packet receive, host MMIO doorbell, etc.), and the end node generates hardware events (*e.g.,* network packet send, DMA operations; see Line 31 in Handler H3). Each handler has an event controller (line 14) that defines how incoming events are ordered and partitioned between a handler's replicas. The handler graph captures both data and pipeline parallelism: events spreading across replicas represent data parallelism, while events flowing through handler chains represent pipeline parallelism.

Enabled optimizations. This representation facilitates parallelism optimizations, allowing the compiler to adapt and transform the graph for different hardware. As discussed in §5, the compiler can split a handler into two pipelined handlers to enhance pipeline parallelism or adjust the number of replicas per handler to increase data parallelism.

4.2 State-based Memory Abstraction

Observation. SmartNICs have multiple memory layers, each with different sharing scopes across compute units. Memories shared by fewer units typically offer better performance but have smaller capacity. Optimal memory placement for program states and variables depends on the state's sharing scope, lifetime, and access pattern.

IR design. We abstract NIC memory accesses into three types of state: local value, persistent table, and context state (Figure 5b), each with distinct sharing scopes, lifetimes, and access patterns. Local values are temporary variables accessed by a single handler replica and live only during a single event's execution. Persistent table state, like the connection limiter's table in Figure 6b's line 1, persists across events and is accessed via lookup/update operations (lines 16 and 18). Context memory is used to pass data along the handler chain, passing data by reference across handlers, unlike events copied by value between handlers. For example, in Figure 6b's line 9, the `payload` value is stored in the context memory, and the reference (`ctx_1`) is passed to H3.

Enabled optimizations. This representation enables various memory optimizations, such as partitioning tables between handler replicas to avoid synchronization and improve data locality (§5.1) and optimizing data flow between handlers using context memory to avoid copies (§5.4).

5 Parallelization Optimizations

Initially, the input run-to-completion, single theaded C function is converted into a single handler without pipeline or data parallelism. As shown in Figure 7, the goal of the parallelization optimization is to transform this single handler into replicated, pipelined handlers based on the target NIC's hardware characteristics.

The total search space for all possible pipeline and data parallelization plans is enormous,[1] and exploring it exhaustively is NP-hard. To manage this complexity, Alkali reduces the search space while ensuring that the generated plan is both correct and near-optimal. To achieve this, Alkali decomposes the search problem into two modules: a mapping engine, which determines the handler replication plans, and a cut engine, which splits handlers to increase pipeline stages. Alkali uses an iterative "Mapping-Then-Cut" loop between these two modules to guide the search.

[1] A program with just a hundred lines of code can have over a million possible plans.

```c
#include <alkali.h> // Alkali library
#include <agilio_spec.h> // Arch Spec

// Connection limiter table, key=src ip
ak_TABLE(64,int,int) cl_table;
// Transport flow state table, key=5 tuple
ak_TABLE(64,five_tuple_t,
        flow_state_t) flow_table;

// _net_recv event defined by agilio_spec
void _net_recv(hdr_t hdr, buf_t payload) {
    /* Generate 5-tuple */
    five_tuple_t tuple5;
    tuple5.srcip = hdr.srcip;
    // ... assign dst ip, ports and proto

    /* Connect Limiter: count pk per srcip */
    int i = ak_tb_lookup(cl_table, hdr.srcip);
    i += 1;
    ak_tb_update(cl_table, hdr.srcip, i);
    // ... check i, drop packet if necessary

    /* Transport: flow state update */
    flow_state_t fs = ak_tb_lookup(flow_table,
                                   tuple5);
    fs.dma_pos += hdr.len;
    // ... update window, OoO detection
    ak_tb_update(flow_table, tuple5, fs);

    // _dma_write defined by agilio_spec
    _dma_write(payload, fs.dma_pos);
}
```

(a) **Run-to-completion (RTC) C function that runs a connection limiter and an on-NIC transport [50]. All red functions are library functions imported from alkali.h.**

```
cl_table = "ak.init"(64){type=persist}
flow_table = "ak.init"(64){type=persist}

"ak.handler"[r=2] H1(hdr, payload){          /*H1, 2 replica*/
 tuple5_0 ="ak.init"()                  /* Generate 5-tuple */
 ctx_0  ="ak.init"()
 sip ="ak.struct_access"(hdr){field=srcip}
 tuple5_1 ="ak.struct_update"(tuple5_0, sip){field=srcip}
 ctx_1 ="ak.ctx_store"(ctx_0, payload){field=p} //pass use ctx
 "ak.genevent"(ctx_1,sip,hdr,tuple5_1){target=H2} //gen event
}

// Ctrl partitions events to H2's replicas by srcip
"ak.ctrl"(H2){field=sip, rule=ordered, parti_by}
"ak.handler"[r=2] H2(ctx_1,sip,hdr,tuple5_1){/*H2, 2 replica*/
  i1 ="ak.tb_lookup"(cl_table, sip)        /* Conn Limiter */
  i2 ="ak.add"(i1,1)
  "ak.tb_update"(cl_table, sip, i2)
  "ak.genevent"(ctx_1,hdr,tuple5_1){target=H3}
}

"ak.ctrl"(H3){field=tuple5_1, rule=ordered, parti_by}
"ak.handler"[r=3] H3(ctx_1,hdr,tuple5_1){    /*H3, 3 replica*/
  fs_0 ="ak.tb_lookup"(flow_table, tuple5_1)   /* Transport */
  dma_pos_0 ="ak.struct_access"(fs){field=dma_pos}
  dma_pos_1 ="ak.add"(dma_pos_0,1)
  len ="ak.struct_access"(hdr){field=len}
  fs_1 ="ak.struct_update"(fs_0, dma_pos_1){field=dma_pos}
  "ak.tb_update"(flow_table, tuple5_1, fs_1)
  payload = "ak.ctx_load"(ctx_1){field=p}
  "ak.genevent"(payload,dma_pos_1){target=dma_send}
}
```

(b) **Optimized αIR for the C program. Red texts are αIR operators. αIR is static single-assignment (SSA) based, thus each value is assigned once. As shown in the IR, after optimizations, the C program is transformed into three replicated handlers.**

Figure 6: **From Alkali source program to optimized αIR.**

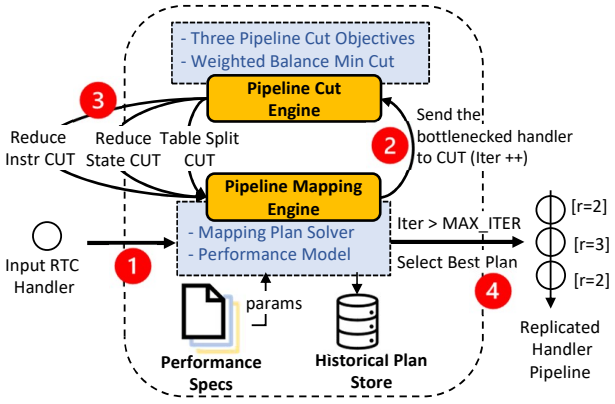


Figure 7: **Parallelization Optimization Overview**

1. **The Pipeline Mapping Engine** takes a non-replicated handler pipeline and uses the SMT solver to search for a replication plan that maximizes the pipeline's throughput within the target NIC's hardware constraints. The solver compares different searched plans' throughput using a cross-target performance model, whose parameters are customized based on the vendor-provided performance specifications. Once the optimized replication plan for a pipeline is found, it is stored in the plan store, and the bottleneck handler in the pipeline is sent to the cut engine (Step 2 in Figure 7).

2. **The Pipeline Cut Engine** improves the bottleneck handler's throughput by splitting the handler into two pipeline stages. It uses a Weighted Balanced Min-Cut algorithm with three criteria: (1) reducing the handler's state size to improve state locality, (2) lowering the handler's instruction count to increase throughput, or (3) splitting persistent tables into different stages to enable a handler's replication. Newly produced increased-staged pipelines are sent back to the mapper (Step 3 in Figure 7).

The loop continues until the predefined maximum number of iterations is reached, which is 100 in our implementation.

5.1 Pipeline Mapping Engine

The pipeline mapping engine takes a non-replicated handler pipeline as input and tries to find a mapping plan – *i.e.,* the number of replicas per handler and the memory placement for each program state – that maximizes its throughput on a given NIC. Our main innovation lies in formulating mapping as an SMT problem. Using a solver and a *cross-target performance model*, our engine searches for a mapping that, given the *hardware resources* and *handler replication constraints*, achieves the highest model-estimated throughput.

The solver retrieves parameters such as the number of compute units, memory hierarchy, and sizes from a "performance

specification" (Table 1). Alkali requires NIC vendors to provide these (simple and brief) specifications.

Mapping plan. The solver searches for two parts of a plan:

The *computational mapping plan* defines the assignment of handlers to compute units on the target NIC. Let the NIC have n compute units, and the handler pipeline have K handlers. The plan $M_1, \ldots, M_n$ ($0 \leq M_i \leq K$) specifies the assignment of handlers to compute units. $M_i = 0$ indicates that no handler is assigned to compute unit i. $M_i = M_j = X$ means that the handler X is replicated on both compute units i and j.

The *state mapping plan* specifies the memory layer to which each persistent table or context memory state is mapped. Let the program have states $S_1, \ldots, S_m$ and the target NIC have X memory layers. The state plan $L_1, \ldots, L_m$ ($1 \leq L_i \leq X$) defines the assignment of a memory layer to each state.

Resource constraints. Mapping plans must account for:

1. **Memory capacity**: For all states mapped to the same memory layer, the sum of the sizes of these states must be smaller than the capacity of memory layer.
2. **Memory sharing scope**: If the handler H is mapped to compute unit C, and if H's state S is mapped to memory layer L, then C must be able to access L.

Replication constraints. Care must be taken when replicating stateful handlers in NIC programs (as noted in [18] in the network functions context). Crucially, replicating a handler with a persistent state may have correctness issues because sharing such states among parallel handler replicas can lead to race conditions. For example, in Figure 6b, when handler H2's replicas run concurrently, a race between the connection limiter table's lookup and update operations can cause the state to become inconsistent.

To avoid race conditions, we can either synchronize handler replicas or disable replication for handlers, but both approaches can significantly and needlessly degrade performance. Fortunately, because tables are indexed by keys, we can partition table state across handlers as long as we ensure that all events accessing a specific key are steered to the same replica [18, 43]. For example, the connection limiter table in H2 (Figure 6b) can be partitioned between replicas when events are steered based on the source IP by the event controller (line 14).

Thus, we add the following constraints during mapping:

1. A handler without any persistent tables or with read-only persistent tables can be replicated.
2. A handler with non-read-only persistent tables can be replicated only if: (1) All lookup and update operations on non-read-only tables use the same key (which can be identified through data flow analysis) and (2) the key is present in the input event parameters, which allows the event controller to use it to steer events (*e.g.,* in Figure 6b, the source IP is H2's input event parameter and is used by the event controller for steering in line 14).

Cross-target performance model. The performance model estimates each pipeline stage's performance using the handler's throughput multiplied by its replication count, and the overall pipeline performance is determined by the slowest pipeline stage. It uses a general formula whose parameters are customized based on performance characterization metrics – also included in the NIC performance specification (Table 1).

Parameter	Description	Usage
$InstrTimeFn(\{\text{op}\})$	A function that returns the instruction execution time for a set of IR operators.	Perf Model
$MemTimeFn(l, b)$	A function that returns memory access latency for layer l when b bytes are accessed.	Perf Model
$CommTimeFn(i, j)$	A function that returns interconnect communication time from compute unit i to j.	Perf Model
CU Count	Number of compute units on the NIC.	Map Constraints
Mem Layers	Total memory layers and how these memories are shared between compute units.	Map Constraints
Mem Sizes	The size of each memory layer.	Map Constraints

Table 1: **Parameters provided by the performance specification. Performance parameters can be derived from offline microbenchmarks.**

The throughput of a handler H's replica is calculated as:

$$T(H) = \frac{1}{Exec_{Time}} = \frac{1}{Time_{Instr} + Time_{Mem} + Time_{Comm}}$$

That is, $T(H)$ depends on:

1. $Time_{Instr}$**:** The handler's instruction execution time is estimated by *InstrTimeFn*, as defined in Table 1. Each backend can provide its own implementation of *InstrTimeFn* through Alkali's interface. The current implementation uses a simple model that sums up the cycle latencies of individual IR operators. In future work, the *InstrTimeFn* interface could be extended to integrate with more detailed or ML-based models for improved accuracy.
2. $Time_{Mem}$: The total memory access time for the handler's states, calculated by summing the memory access latencies of all mapped states. The latency for accessing each state is calculated based on the $MemTimeFn$ in the performance specification.
3. $Time_{Comm}$: The time of sending events to the next pipeline stages through on-chip interconnects. If this handler has M replicas and sends the event message to the handler on compute unit j, the communication time is $\sum_{i \in M_Replicas} CommTimeFn(i, j)$, which models the message incast overhead that scales linearly with concurrent senders. This term penalizes mappings with excessive handler replications that cause significant communication overhead when pushing data to the same destination.

It is important to note that precise mapping-performance estimates are not critical. The primary objective is to comparatively rank and assess different mapping strategies. This design philosophy allows our performance model to remain general and cross-target, accommodating various hardware configurations.

Lock-free execution. Alkali's replication algorithm produces lock-free programs by avoiding synchronization among compute units (*e.g.,* Alkali replicates a handler only when

its state can be partitioned). This design is driven by two factors. First, locking overhead is typically prohibitive in high-throughput NIC data paths, and many NIC programs can operate without synchronization when sufficient hardware parallelism is available. Second, accurately estimating locking and synchronization overhead in the performance specification is challenging.

5.2 Handler Cut Engine

The handler cut engine aims to improve the throughput of a bottleneck handler by splitting its instructions into two chained pipeline handlers. Similar to prior work [9], the basis of our approach is to construct a flow network based on the data dependencies of handler instructions which allows us to model the problem of partitioning instructions into pipeline stages as selecting cuts in the flow network. Our innovations lie in devising a Weighted Balanced Min-Cut formulation showing how different performance objectives map to this modeling and ensuring and proving the correctness of the cut when the handler has persistent NIC program states.

Construct flow network. Figure 8a shows the flow network graph constructed for the H1 handler in Figure 6b. Each gray node (L5–L10) represents an IR statement from lines 5 to 10 of Figure 6b, while each white node corresponds to the value defined by a statement. For every statement S that defines a value V, we add a *definition edge* (S,V) with a weight equal to V's byte count, representing the cost of transmitting V between pipeline stages. Additionally, for each statement S that uses V, we add a *usage edge* (V,S) with infinite weight.[2]

As shown in the figure, by finding a cut that divides the nodes into disjoint subsets, we can split the H1 handler into two pipelined handlers. The figure shows a possible cut where the first pipeline stage contains statements L5 and L6, while the second stage includes statements L7–L10. The communication cost between stages is equal to the sum of the weights of the cut edges (79B in this example).

Uncuttable region for persistent table. Cutting access to persistent tables across pipeline stages can lead to correctness issues due to inconsistent operations. For instance, Figure 8b illustrates the flow network for the H2 handler in Figure 6b, showing a cut that separates the table's lookup operation ($L16$) from its update operations ($L17$ and $L18$) into two pipeline stages. In this scenario, when two packets enter the pipeline sequentially, with the first packet in stage 2 and the second packet in stage 1, the second packet's table lookup may occur before the first packet's table update, which can be incorrect.

We prevent such cuts as follows: we first add a UNCUT node (Figure 8b) for each persistent table and add bidirectional infinite-weight edges from the lookup (L16) and update (L17) statement nodes to the UNCUT node. This ensures that all statements involved in reading or writing the same persistent table remain within the same pipeline stage.

Weighted balanced min cut. We now consider where to cut the flow graph to most effectively improve the handler's performance. We develop a weighted balanced min-cut algorithm that extends the traditional balanced min-cut approach [9, 61] with specialized weight functions tailored to our performance objectives. The balanced min-cut algorithm starts by finding a min-cut of the graph and then iteratively applying the min-cut process to the larger side of the flow graph until it identifies a balanced cut that satisfies $|W - \overline{W}| \leq \beta$, where W and $\overline{W}$ represent the sum of node weights on the left and right sides of the cut, respectively. β is the imbalance tolerance threshold.

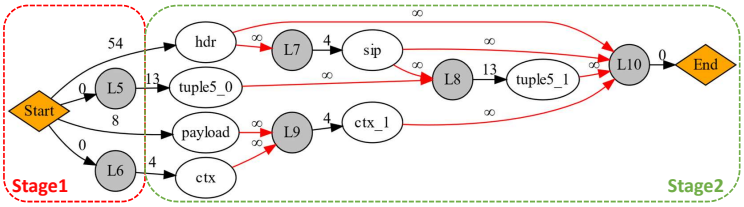
We note three main sources of speedup from pipelining a handler, which guide our node weight functions' design:

1. **Reducing handler state**: By partitioning the handler state equally into two stages, the state size within each handler is decreased. This allows the handler state to be allocated in a local, faster but smaller memory layer in the next mapping iteration. If the handler's performance is memory-bound, throughput can be improved. *Weight function:* Based on this, we assign weights equal to the byte count to all value nodes (white nodes in Figure 8a) and assign a weight of 0 to all other types of nodes.
2. **Reducing instruction counts**: Equally partitioning the handler's instructions into two stages decreases the instruction count in each handler. If the handler's performance is compute-bound, this can improve throughput. *Weight function:* Thus, we assign a weight of 1 to all statement nodes (gray nodes in Figure 8a) and assign a weight of 0 to all other node types.
3. **Splitting tables to enable handler replication**: As discussed in §5.1, a handler cannot be replicated if it contains multiple persistent tables with different keys. By splitting tables with different keys into different pipeline stages, each handler becomes replicable, allowing the program to utilize more compute units in the next mapping iterations. *Weight function:* First, we group tables with identical keys and add a UNCUT node for each "table group" to prevent splitting same-key tables across different stages. Then we assign a weight of 1 to each table group's UNCUT node and 0 to all other node types. This weight assignment encourages the weighted balanced min-cut algorithm to split tables with different keys into different stages while keeping the same key tables in the same stage.

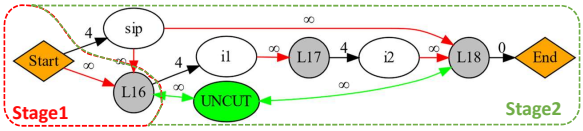
5.3 End-to-end Example and Properties

Figure 9 shows a simple example that puts these components all together. In iteration 1, the input RTC handler A is mapped by the mapper. Since it has multiple tables with different keys, A can only have a single replication. Next (iteration 2), A is sent to the cut engine, which generates three 2-stage han-

[2]The infinite weight ensures that cuts happen only on *definition edges* – this is because the weights of the cut edges should reflect a value's transmission cost, which depends on its defined size rather than the number of uses.



(a) **Constructed flow network graph of Figure 6b's H1 handler. It shows a cut that splits the handler into two pipelined handlers, with L5 and L6 in the first pipeline stage, and L7-L10 in the second pipeline stage.**



(b) **Flow network graph of Figure 6b's H2 handler. This cut shown in the figure is incorrect since it split the table lookup operation (L16) with the update operation (L18) into two stages. By adding the UNCUT node we can prevent this cut.**

Figure 8: **Constructed flow network graph.**

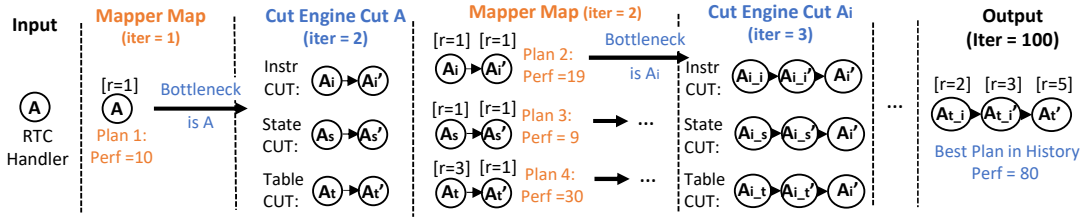


Figure 9: **End-to-end example.**

dler pipelines based on three performance objectives. These pipelines are mapped, which calculates the replication number for each handler within each pipeline. Compared to iteration 1, the program's performance may increase or decrease after the cut. In subsequent iterations, the mapper identifies the bottleneck handler for these three pipelines and sends it to the cut engine to generate 3-stage pipelines. This loop continues until the predefined max. iterations is reached, after which the optimizer selects the plan with the best performance from the history plans.

Search optimality vs. efficiency. It is important to note that Alkali does not guarantee optimality; in rare cases, the search algorithm may overlook better parallelization plans. However, Alkali is not designed to find the absolute best plan. Instead, its goal is to efficiently generate correct, competitive plans that closely approximate expert implementations. By employing domain-specific heuristics, Alkali filters out suboptimal plans, which narrows the search space and reduces reliance on an exact performance model.

Correctness. In §A.2, we show that after optimization, the replicated handler pipeline P_a is semantically equivalent to the original RTC handler P_b; that is, *after executing the same input event sequence, P_a and P_b will generate the same output event sequence and end in the same persistent state.*

5.4 Other Optimizations

After the parallel optimizations, Alkali performs a pass of four other optimizations:

1. Traditional optimizations. These include peephole optimizations [1], control flow reduction, and elimination of common expressions [8] and dead code [20].

2. Event controller generation. After replicating handlers, the compiler generates event controllers to dispatch events between replicas of handlers based on the table keys.

3. Context memory conversion. α*IR* supports two methods to pass variables along the handler chain: *Pass-by-event* and *Pass-by-context* (§4). Context memory conversion optimization determines when to use pass-by-context based on the variable's size and lifetime. If a variable is large or needs to persist across multiple handlers, Alkali opts for pass-by-context to avoid redundant data copying.

4. Saving context memory. This optimization further minimizes context memory allocations. Each handler pipeline analyzes the temporal overlap in the lifetimes of data stored in the context. It then tries to reuse memory for data with non-overlaspping lifetimes (detailed example relegated to §A.1) thereby saving overall context memory.

6 Programming Frontend, Architecture Spec

Our prototype supports programs written in C, using a subset of the language that is expressive enough for a wide range of NIC programs (§8.1). For example, developers can use conditional branches (if-else) and bounded loops, but pointer operations and unbounded loops are not currently supported. Since Alkali programs are written as single-threaded executions, concurrency primitives such as mutex locks are also not supported. Additionally, developers must include `alkali.h`, which provides built-in data structures (*e.g.,* tables, packet buffers) and library functions for common operations such as table lookup/update and buffer extract/emit. Looking ahead, we plan to extend Alkali to support additional languages, including Python and P4.

Because network traffic distribution affects program performance, developers can annotate their code to make Alkali's compiler optimizations workload-aware. For instance, they can limit the maximum number of table replications based on active flow counts, allowing the SMT solver to use this as a replication constraint. Additionally, developers can specify branch probabilities to enhance the performance model's accuracy in estimating handler execution efficiency. Below are examples:

```
1  annot_MAX_REPLICA(3)    ak_TABLE(64,int,int) ftable;
2  annot_BRANCH_PROB(0.7)  if(a){...};
```

Each NIC vendor provides an "architecture specification", which is included as a header file in the Alkali program (*e.g.,* line 2 in Figure 6a). The specification defines the high-level interfaces for supported hardware events on the target NIC, enabling developers to write functions to process and generate these events. For example, Figure 6a's C function processes `net_recv` events and generates `dma_write` events and is defined by the specification as follows:

```
1  // Event processed by programmer
2  void _net_recv(hdr_t hdr,buf_t data){}
3  // Event processed by hardware
4  void _dma_write(buf_t data,long addr){return;}
```

This event-based interface hides NIC micro-architectural details, enhancing program portability: a single Alkali program can be compiled for NICs with the same event types.

7 Backends and Compiler Implementation

We implement four distinct NIC backends, each representing a different type of SmartNIC architecture: Agilio [36] (on-path SoC NICs), BlueField-2 [34] (off-path SoC NICs), Alveo [3] (FPGA NICs), and PANIC [30] (a prototype of an ASIC NIC).

Each backend takes the optimized αIR and generates target-specific executables. The Agilio backend produces a NFP Micro-C code [37], while the BlueField-2 backend converts the αIR into LLVM IR, generating ARM binaries linked to the DPDK runtime. The Alveo backend outputs Verilog code, and the PANIC backend produces bare-metal RISC-V binaries with scheduler configurations.

The backend translates the IR objects into architecture-specific components (Table 5). For instance, in the Agilio NIC, handlers run as micro-engine threads, while on FPGAs, handlers are implemented as hardware modules with parallel, pipelined LUTs (FPGA backend details are in Appendix B).

Because handler replicas running on parallel compute units can reorder events due to variable processing times (*e.g.,* smaller packets finishing before larger ones), the αIR event controller ensures proper event ordering by default. It reorders events from the previous stage's handler replicas before dispatching them to the next stage. Each backend generates different code to handle event sequencing (*e.g.,* the FPGA generates a reorder buffer, while Agilio generates code that invokes a hardware sequencer). For programs that tolerate out-of-order processing [7, 28], programmers could add annotations to disable event ordering, allowing backends to use FIFO queues, not sequencers.

We implement the C frontend, Alkali compiler, and backends using the MLIR framework [23], totaling 20K lines of C++. We also developed runtime libraries for each NIC target, totaling 4K lines of code, providing features such as dynamic memory allocation, event steering, and inter-compute unit communication queues. Since Alkali centralizes most optimization and mapping in a common compiler mid-end, adding support for new backends is streamlined. *e.g.,* the BlueField-2 backend was developed in a week using 1107 lines of code.

8 Evaluation

We evaluate Alkali by answering the following questions:

1. Does it help easily write portable NIC applications (§8.1)?
2. Does Alkali programs perform well across targets (§8.2)?
3. Does the mapping-then-cut loop find good plans (§8.3)?
4. Does the mapping engine find optimal computational and state mapping plans (§8.4)?
5. How do other optimizations improve performance (§8.4)?

Testbed: Our testbed comprises five commodity servers with two Intel Xeon Gold 6258R 28-core CPUs and 256GB of RAM. We use one server equipped with a Mellanox ConnectX-6 DX NIC to generate DPDK traffic to four SmartNIC servers. Each SmartNIC server is equipped with one of these four SmartNICs: (1) a 100GbE AMD Alveo U280 [3], atop which runs the generated FPGA program at 250MHz; (2) PANIC's FPGA prototype [30] (configured to have 32×250MHz RISC-V cores); (3) a 40GbE Netronome Agilio CX [36] (equipped with 48×800MHz micro engines); and (4) a 100GbE BlueField-2 DPU (8×2.5GHz ARM A72 cores and 16GB DRAM).

8.1 NIC Applications Implemented with Alkali

Table 2 lists five NIC applications implemented using Alkali. For the L2 forwarding [40], JSQ RSS [17], and FlexTOE [47], we re-implemented in Alkali-C based on their open-source implementations to ensure functional equivalence. For the NF chain and RPC message reassembly applications, where no open-source implementations were available, we asked a graduate student with SmartNIC expertise to write them using BlueField-2's DOCA API as our baseline.

Table 2 compares the lines of code (LoC) between Alkali programs and their baseline implementations. The Alkali implementations are 5–10× smaller, as Alkali allows developers to focus on core logic while abstracting away target-specific details. All Alkali-C programs can be compiled for different NICs with no modification, demonstrating Alkali's portability.

Application	Description	Alkali LoC	Original Impl LoC (Target NIC)
L2 Forward [40]	Swaps the Ethernet source and destination addresses and echoes packets back to the network.	28	128 (Agilio)
Flextoe Transport [47]	Flextoe's RX pipeline (same as Figure 2), it validates the packet header, performs OoO detection, and looks up and updates the **flow table**. It then generates ACKs and DMA payloads to the host.	224	1264 (Agilio)
NF Chain	A chain of three network functions to offload a stateful firewall, a connection limiter, and a L4 load balancer. The chain has **six tables**: four use the 5-tuple as the key, while two use src and dst IP addresses as the key.	289	780 (BlueField-2)
RPC Message Reassembly	It drops OoO packets, looks up the **flow assemble buffer table** to get the assemble buffer for this flow, and appends the **incoming packet's payload** to this buffer and update table. It sends assembled msg to host.	120	467 (BlueField-2)
JSQ RSS [17]	Implements Ringleader's request steering [28]. NIC uses a **worker load table** to track the inflight request count on 32 host cores and steers incoming requests to the least-loaded core.	126	1312 (FPGA)

Table 2: **Application description. The last two columns compare LoC when using Alkali and vendor-provided SDK.**

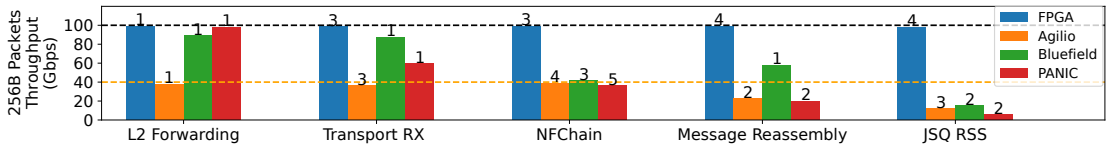


Figure 10: **Performance of Alkali programs on NIC targets. The number over the bar shows the number of pipeline stages in the generated code.**

	Throughput (Gbps)		Latency	LUTs
	64B Packet	128B Packet	(ns)	Usage
Ringleader	64 Gbps	100 Gbps	64ns	208K(1.6%)
Alkali	64 Gbps	100 Gbps	84ns	247K(1.9%)

Table 3: **Alkali's JSQ RSS performance compared to Ringleader's original implementation on FPGA.**

8.2 Performance on Heterogeneous Targets

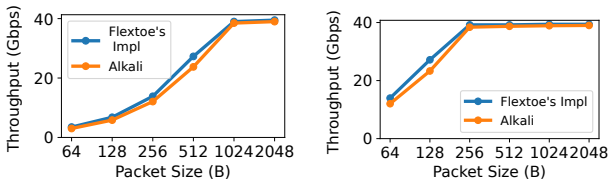
We evaluate the above applications' performance on different targets and compare them with experts' implementations.

Alkali program performance. Figure 10 shows the end-to-end throughput of five applications across four targets, measured with 256B packets. The L2 forwarding application achieves the line rate on all NICs. FlexTOE also reaches line-rate performance on all NICs except PANIC, where the low-frequency RISC-V cores become a bottleneck. FlexTOE's flow table satisfies replication constraints and can be partitioned across multiple handler replicas, enabling data and pipeline parallelism to sustain high throughput.

The NF chain application, consisting of six tables, achieves the line rate on the FPGA and Agilio NICs. Alkali's cutting algorithm efficiently splits the NF chain's tables into pipelined handlers, ensuring that tables in each handler share the same key and can be replicated. On BlueField-2, performance is limited to 50 Gbps due to high inter-core communication overhead, which degrades pipeline efficiency.

Message reassembly operates below line rate on Agilio (22 Gbps), BlueField-2 (57 Gbps), and PANIC (19.73 Gbps) due to the costly buffer aggregation operation, which involves copying the entire packet payload to the reassembly buffer. This overhead significantly reduces throughput on SoC NICs.

JSQ RSS has limited throughput on Agilio, BlueField, and PANIC. It uses a small load table to track in-flight packet counts per core. Upon receiving a packet, it looks up each core's load, selects the least loaded one, and updates its count. Since the table lookup and update operations use different keys, handler replication is impossible, creating a bottleneck.



(a) **Single Flow Throughput** (b) **16 Flows Throughput**

Figure 11: **Alkali compiled FlexTOE's performance compared to its original implementation on Agilio.**

The FPGA achieves line-rate performance because the table is mapped to local registers, allowing both operations to complete within two cycles and ensuring high throughput.

Comparison with experts' implementations. We evaluated and compared Alkali generated program performance with FlexTOE [50]'s open-source Agilio implementation and Ringleader [28]'s FPGA implementation. We also compared the message reassembly and JSQ RSS application with NIC expert's Bluefield implementations.

Figure 11 shows that Alkali closely matches FlexTOE with a 10% performance gap. Both achieve line rate for packet sizes larger than 1KB with a single flow or for packets larger than 256B with 16 flows. The gap is mainly due to Alkali's mapping engine searched replication count for a pipeline stage being not optimal when running on real hardware.

Alkali's JSQ RSS performance matches Ringleader's implementation. The generated Verilog achieves comparable throughput to an FPGA expert's design since the compiler automatically breaks the program into pipeline stages. Alkali has 30% higher latency and 18% higher resource utilization due to the backend inserting registers between dependent operations to avoid timing issues. A future backend could remove unnecessary registers when timing allows.

We then asked a NIC expert to port this JSQ RSS program and implement the message reassembly program on Bluefield. The result in Appendix C shows that Alkali's performance lags 0.6% to 9.8% behind the expert implementations. Notably, the expert spent 14 hours implementing the fine-tuned

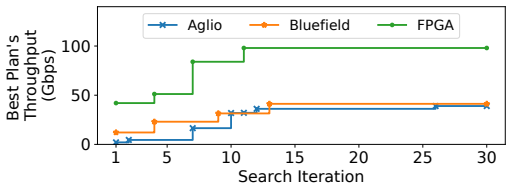


Figure 12: **The best-searched parallelization plan's throughput under different iteration counts.**

applications, whereas Alkali compiled from unmodified code.

8.3 Effectiveness of Mapping-Then-Cut

We use the NF chain application to demonstrate how Alkali's mapping-then-cut loop searches for an optimal parallelization plan. Figure 12 shows that as the number of search loop's iterations increases, the performance of the historical best plan improves. In different iterations, Alkali tries different cutting methods, gradually dividing the program into multi-stage pipelines, which improves performance. Eventually, performance plateaus as finer-grained parallelism no longer adds benefits. The performance saturation point varies by architecture. For Bluefield, the optimal plan is found with a 3-stage pipeline (at iteration 13), while for Agilio, it is found with a 4-stage pipeline (at iteration 26).

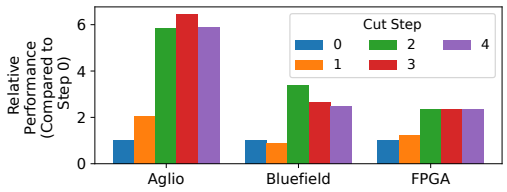
We provide a walk-through of how the program is cut into pipelined handlers. For each NIC, we present the cut history for the 5-stage pipelines that perform best. Figure 13's bar graph shows throughput changes after each cut step; the table below details the cut type, the replication count per handler, and the identified bottleneck. The results show: (1) different cut methods improve performance in different steps. For example, the table cut improves performance in Agilio's Steps 1 and 2, while in Step 3, the state cut does. This highlights the importance of Alkali trying all three cut methods per iteration; (2) the bottleneck handler shifts after each cut, emphasizing the need to iteratively identify bottlenecks before applying further cuts; and 3) a cut does not always guarantee improved performance. For example, Bluefield's performance decreases after the first cut, but improves after the second. This shows the importance of Alkali maintaining the plan history and continuing the search to avoid getting stuck in the local optima.

8.4 Effectiveness of the Mapping Engine

We now evaluate whether the mapping solver can identify good handler replication and state mapping plans.

Handler replication plan. We evaluate whether Alkali can effectively determine handler replications. We did an exhaustive manual search of all possible replication plans[3] for the

[3]Since the NF chain achieves line-rate on Agilio, we restricted the application to a single island (12 micro-engines) to observe throughput variations across different mapping plans.



Cut Step	[Cut method], replication count of each stage		
0	1	1	1
1	[table] 21	[table] 31	[table] 41
2	[table] 722	[table] 322	[table] 212
3	[state] 3222	[state] 2221	[instr] 1212
4	[instr] 32222	[state] 12221	[instr] 12121

Figure 13: **Cut traces of the best performing 5-stage pipelines on three NICs. Use the NF chain application. In the table, the number sequence represents the per-stage replication count. *e.g.*, 22 indicates a 2-stage pipeline with 2 replicas per stage. The bottleneck stage is shown in red.**

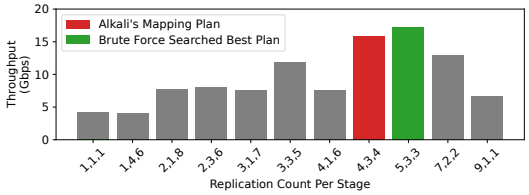


Figure 14: **Alkali mapping plan's performance compared with the brute-force searched plans. Measured on Agilio using NF chain.**

State Placement	FlexTOE Perf	NF Chain Perf
Place into EMEM	1.05 Mpps	0.86 Mpps
Place into CLS	2.06 Mpps	3.34 Mpps
Alkali solver's plan	6.28 Mpps	4.43 Mpps

Table 4: **Comparison of the solver's state placement plan's performance with the other two naive plans.**

NF chain application's 3-stage pipeline and measured the performance of these plans on the Agilio NIC. We compared these plans' performance with the one generated by Alkali's mapping solver (Figure 14). The results show that Alkali finds the second best plan in the search space, whose performance is 8.4% below the best replication plan. Alkali did not find the best plan because of the simplicity of the performance model, which estimates computational cost by summing the execution latencies of individual IR operations. Nonetheless, the results suggest that this model is adequate for automatically finding practical mappings.

Alkali's solver can identify the optimal replication plan for both FPGA and BlueField-2. For FPGA, Alkali's performance model can accurately estimate the handler's throughput, as the execution cost (initial interval) of each IR operation type is predictable when running on the hardware.

State placement. Alkali's mapping solver strategically places program states in the fastest available memory tier based on the state's sharing scope and size. For FlexTOE and NF chain, we compare Alkali's state placement plan with two other naive mapping strategies on Agilio: (1) placing all states into the global memory hierarchy (EMEM) and (2) placing all states into the cluster local scratch memory (CLS).

Table 4 shows that, compared to the naive placement strategies, Alkali's state placement yields a 1.32× to 6× improvement in throughput. Alkali achieves better performance by utilizing all of Agilio's memory layers (CAM, fast local memory (LMEM), CLS, and EMEM). Alkali places frequently accessed, small states in CAM and LMEM, while storing infrequent, larger states in CLS and EMEM.

Performance model sensitivity. We found that Alkali's performance model is robust to parameters in performance specification. Even with parameter errors up to 50% of the accurate value, the mapping result performs only 1.3% lower than when using accurate values.

Effectiveness of other optimizations. We studied the benefits of other compiler optimizations (Section 5.4) and measured the throughput of the FlexToE program on BlueField-2, Agilio NICs, and PANIC. Our results show that traditional compiler optimizations yield performance improvements of 5%–20%. Context conversion results in gains of 6%–113%, while context reduction provides improvements of 6%–137%.

9 Limitations and Future Work

Handling dynamic workloads. Alkali currently depends on program annotations to enable workload-aware compile-time optimizations (see §6). However, this approach has limitations, as it does not account for workload shifts occurring at runtime. In the future, integrating Alkali with a dynamic runtime workload profiler could enable automatic recompilation and adaptation in response to changing workloads.

Detailed performance specifications. Alkali's performance model relies on functions defined in the performance specification. Our prototype utilizes simple functions (*e.g.,* summing instruction cycles for *InstrTimeFn*) to estimate performance. While this approach achieves good results, as demonstrated in our evaluation, future work will incorporate more detailed, potentially ML-based, per-hardware models. These enhanced models will account for additional factors such as memory contention, cache hierarchies, and out-of-order execution, further improving the accuracy of parallel optimization.

Support for locking. As mentioned in §5.1, Alkali currently does not support locking due to the challenges of accurate model contention overhead. In the future, with a more detailed performance specification that models locking costs, Alkali could enable handler replication for non-partitionable shared states by automatically generating locks to safeguard critical sections where the shared state is read-then-modified.

10 Related Work

Compilers for programmable switches. Programmable switch compilers such as Domino [52] partition C code into pipeline stages, while Gallium [62] and Lyra [12] distribute programs across switches and servers. Unlike Alkali, these compilers are designed for switches with fixed line-rate pipelines, focusing on pipeline partitioning and resource packing. In contrast, NICs support pipelining and data parallelism without a fixed rate, which requires Alkali to explore multiple dimensions of parallelism with extensive performance optimizations.

IR for accelerators. Prior work has developed domain-specific IRs for hardware accelerators, including GPUs [22, 54] and FPGAs [32, 49]. However, α*IR* is distinct in that it is specifically designed to abstract the unique architectural features of NICs, such as parallel event processing and memory hierarchies. It also supports key functionalities commonly used in NIC programs, including persistent tables, flow steering, and context states.

SmartNIC performance prediction. Many tools have been developed to predict the performance of NIC programs [13, 39, 46, 51]. Clara [46] transforms NIC programs into LLVM IR and uses ML models to predict the program's performance on various SoC NICs. Pipeleon [60] adopts a profiling-based approach to predict the performance of the P4 program. HLSPredict [39] and LEAPER [51] forecast FPGA program throughput and latency. Alkali can integrate these models for parameter estimation in the performance specification. For example, Clara could take α*IR* as input and estimate its execution latency on a given target, which can be integrated with Alkali's performance model (*e.g.,* through the *InstrTimeFn* interface).

Network programming languages and IRs. Many existing works have developed high-level programming languages for network hardware devices, such as Lucid [53], Click [21, 26], Floem [44], and λ-NIC [7]. Alkali is complementary to these efforts and can be extended to support these programming languages. NetASM [48] provides an IR for network programs using an assembly instruction set. However, it lacks automatic parallelization optimizations for heterogeneous targets.

11 Conclusions

We present Alkali, a multi-target NIC compilation framework that introduces a novel intermediate representation to abstract the compute parallelism and state access of NIC programs. Alkali uses an iterative parallelism optimization algorithm that automatically transforms single-threaded programs into highly optimized ones. Our prototype effectively compiles five NIC applications across four distinct NIC architectures. The compiled programs run efficiently on all targets, exhibiting a performance gap of less than 9.8% compared to expert implementations.

12 Acknowledgements

We thank our shepherd, Jiaqi Gao, and the anonymous NSDI reviewers for their feedback that significantly improved the paper. Jiaxin Lin is supported by a Google PhD Fellowship.

References

[1] Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[2] Alkali. Alkali Repo. https://github.com/utnslab/Alkali.

[3] AMD Xilinx. Adaptable Accelerator Cards for Data Center Workloads. https://www.xilinx.com/products/boards-and-kits/alveo.html.

[4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *NSDI*, volume 20, pages 93–109, 2020.

[5] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, et al. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, 2023.

[6] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. In *OSDI*, volume 20, pages 973–990, 2020.

[7] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. λ-nic: Interactive serverless compute on programmable smartnics. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 67–77. IEEE, 2020.

[8] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[9] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. Automatically partitioning packet processing applications for pipelined architectures. *ACM SIGPLAN Notices*, 40(6):237–248, 2005.

[10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66, 2018.

[11] Michael Galles and Francis Matus. Pensando distributed services architecture. *IEEE Micro*, 41(2):43–49, 2021.

[12] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 435–450, 2020.

[13] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. Lognic: A high-level performance model for smartnics. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 916–929, 2023.

[14] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 60–68, 2019.

[15] Intel. Intel® Infrastructure Processing Unit (Intel® IPU) — intel.com. https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html. [Accessed 02-02-2024].

[16] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216. IEEE, 2017.

[17] Jiaxin Lin. Ringleader Public Repo. https://github.com/utnslab/RingleaderNIC/tree/main.

[18] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 501–516, 2019.

[19] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 756–771, 2021.

[20] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM Sigplan Notices*, 29(6):147–158, 1994.

[21] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular

router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[22] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[24] Nikita Lazarev, Tao Ji, Anuj Kalia, Daehyeok Kim, Ilias Marinos, Francis Y. Yan, Christina Delimitrou, Zhiru Zhang, and Aditya Akella. Resilient baseband processing in virtualized rans with slingshot. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 654–667, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[26] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[27] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 371–385, Renton, WA, April 2022. USENIX Association.

[28] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. {RingLeader}: Efficiently offloading {Intra-Server} orchestration to {NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, 2023.

[29] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. Towards accelerating data intensive application's shuffle process using smartnics. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(2):1–23, 2023.

[30] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. Panic: A high-performance programmable nic for multi-tenant networks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 243–259, 2020.

[31] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[32] Kingshuk Majumder and Uday Bondhugula. Hir: An mlir-based intermediate representation for hardware accelerator description. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 189–201, 2023.

[33] Marvell Technology. Marvell® OCTEON 10 DPU Platform. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf. [Accessed 02-02-2024].

[34] Mellanox Technologies. NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[35] Microsoft. Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[36] Netronome. Netronome DPU. https://www.netronome.com/products/smartnic/overview/.

[37] Netronome. Programming Netronome Agilio® SmartNICs. https://www.netronome.com/media/documents/WP_NFP_Programming_Model.pdf.

[38] NVIDIA. NVIDIA DOCA Software Framework. https://developer.nvidia.com/networking/doca.

[39] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. Hlspredict: Cross platform performance prediction for fpga high-level synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–8, 2018.

[40] OpenNFP. OpenNFP Packet Processing Repo. https://github.com/open-nfpsw/c_packetprocessing/tree/master.

[41] P4lang. p4-dpdk-target. https://github.com/p4lang/p4-dpdk-target.

[42] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Francisco Pereira, Fernando MV Ramos, and Luis Pedrosa. Automatic parallelization of software network functions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1531–1550, 2024.

[44] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas E Anderson. Floem: A programming system for nic-accelerated network applications. In *OSDI*, volume 18, pages 663–679, 2018.

[45] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. {FlowBlaze}: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[46] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 772–787, 2021.

[47] Rajath Shashidhara. Flextoe Public Repo. https://github.com/tcp-acceleration-service/FlexTOE/tree/main.

[48] Muhammad Shahbaz and Nick Feamster. The case for an intermediate representation for programmable data planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–6, 2015.

[49] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 940–953, 2019.

[50] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, 2022.

[51] Gagandeep Singha, Dionysios Diamantopoulosb, Juan Gómez-Lunaa, Sander Stuijkc, Henk Corporaalc, and Onur Mutlua. Leaper: Fast and accurate fpga-based system performance prediction via transfer learning. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 499–508. IEEE, 2022.

[52] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[53] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 731–747, 2021.

[54] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85. IEEE, 2017.

[55] The P4 Language Consortium. P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[56] The P4.org Architecture Working Group. P4 Portable NIC Architecture (PNA). https://p4.org/p4-spec/docs/PNA.html.

[57] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 17–32, New York, NY, USA, 2021. Association for Computing Machinery.

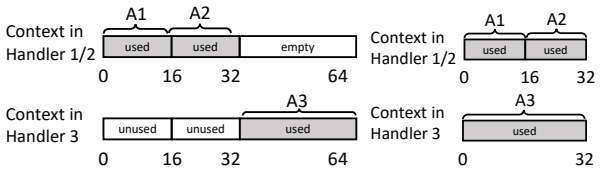
[58] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–131, 2020.

[59] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, 2023.

[60] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, TS Eugene Ng, et al. Unleashing smartnic packet processing performance in p4. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1028–1042, 2023.

[61] Hannah Honghua Yang and DF Wong. Balanced partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1533–1540, 1996.

[62] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 283–295, 2020.



(a) **Without Opt** (b) **With Opt**

Figure 15: **Context memory optimization.**

A Additional Design Details

A.1 Context Memory Reduction Example

Figure 15 shows how context memory optimizations reduce context memory allocations. Consider a program with a call chain handler 1 $\rightarrow$ handler 2 $\rightarrow$ handler 3. The context state passed through the chain has three fields: A1 (16 bits), A2 (16 bits), and A3 (32 bits). Figure 15a shows the allocation of context memory without the optimization. It allocates 64b to store A1, A2, and A3. However, from the figure, we can see that A3 has a non-overlapping lifetime with A1 and A2, as A1 and A2 are used only by handlers 1 and 2, while A3 is used only in handler 3. With context memory optimization, for each handler call chain, the compiler analyzes the lifetimes of context fields and attempts to reuse buffer slots for context fields with non-overlapping lifetimes. As such, the compiler can allocate only a 32b context memory, allowing A3 to reuse the memory slots of A1 and A2 (Figure 15b).

A.2 Proof of Parallelization Optimizations Correctness

We prove that the transformed pipelined, parallelized program is fully equivalent to the original run to completion (RTC) program. We define the equivalence using as following:

Theorem 1. *With the same starting persistent state, after executing the same input packet sequence, a run-to-completion program P_a and a transformed pipelined, parallelized program P_b would generate the same sequence of external events and ends up in the same persistent state.*

Proof. The proof is based on the following property of αIR transformation implementation:

Prerequisite 1. *αIR transformation does not break data dependency.*

Prerequisite 2. *αIR transformation does not reorder dependent table and event generation operations, as we convert the ordering constraint into data dependency in αIR.*

We first prove that the program state and generated event values are equivalent after transformation after execution of a single packet. In case of single packet execution, the order of generate events is Thanks to the αIR's simplified memory model and operations, the only program state is table value. We need to prove

Lemma 1. *After executing a single packet, in P_a and P_b, for any table T and any key k, the value $T[k]$ will be the same; for any generate command, the content of the generated event is the same.*

Proof. We reason about the value of operations. There are two types of operations in αIR. First, all operations except table access operations in αIR are pure operations, whose value is only decided by its parameter values, and not related to any system state; Second, Only table operations $lookup, update$ and event $generate$ operations are non-pure operations. which table operations will change system state and event generation will change the external state.

Based on that, We further prove all operation output values remain unchanged after the transformation by reduction. Based on **pre-assumption 1**, All of our transformations reserve the data dependency for non-pure operations: which means that for all pure operations we do not change the input parameters, thus the output values are the same. For the values produced by $non-pure$ operation, we use reduction to prove they are not changing. We prove the following For any table T and key k, if we perform the same sequence of operations, we will end up in the same system state. We prove by reduction on **sequence length**. For the first-ever non-pure operation in the data dependency graph (in topological order), all its input parameters must be pure values, it holds in this case. At any sequence l, as all system state and values are all the same, the parameter for input $l+1$ is also the same, and we will get same output value and system states. □

After understanding the case for single packet, We could easily get the following conclusion based on the proof: 1. Reschedule of pure operations with both pre-assumption hold will not change the output value of them. 2. P_a and P_b is equivalent when a sequence of packets are executed in order without parallelism.

Next, we reason about the case when multiple packets could be executed in pipeline- or data-parallelism, based on the following properties:

Prerequisite 3. *A packet will be processed by only one single handler out of multiple replicated handlers.*

Prerequisite 4. ***Key Partition***. *For a partition P and parallel handlers H_i after the partition, if any table operation with key k happens in H_k, all table operations with key k must happen in H_k.*

Prerequisite 5. ***Event Reorder***. *The order of generated events in parallel handler for any input packet sequence is the same as if the sequence is executed without parallelism. This is achieved with reorder buffers.*

	Event	Handler	Controller	State		
				Local	Context	Persistent Table
Agilio	Inter-ME msg	A hardware thread on ME	HW Sequencer	ME local registers	Mem pool in CLS/EMEM	Table in TCAM/CLS/..
Bluefield	Inter-ARM core msg	A thread on a ARM core	Reorder queues	Stack var in DRAM	Mem pool in DRAM	Table in DRAM
FPGA	Inter-module msg	Pipelined LUTs	Reorder Buffer	Intra-module datastream	Inter-module datastream	Table in BRAM/TCAM
PANIC	Inter-RISCV core NoC msg	Instructions on RISCV core	Centralized Sche queues	RISCV core local mem	NoC msg metadata	Local State Only

Table 5: **How αIR's concept maps to different hardware targets.**

We now reason about the case that P_a and P_b is the same with multiple packets executed in parallel. We first prove that the value of table and generated events is equivalent:

Theorem 2. *For a partition P and parallel handlers H_i. For any sequence of packets, the table state change (For any table T, P_a and P_b, for any key k, the value $T[k]$ is the same for parallel (multiple RTC in parallel on different H_i) and single packet execution.*

Proof. Considering each different k is executed independently in the same sequence, leveraging Pre-assumption 3. Similarly, for a multi-partition-stage pipeline, we can prove by reduction if the **input event sequence** is the same for each partition stage, the program is identical to the original program. □

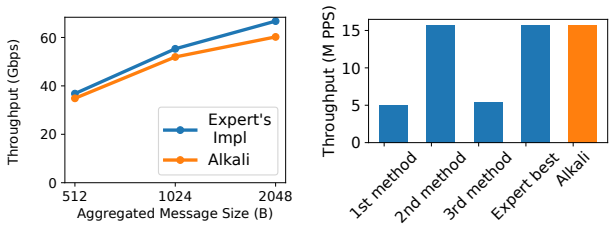
Also, based on Pre-assumption 5, the order of generated event is identical. P_a and P_b generate same event sequences, as the order and value of each event in the sequence is identical. So for any input event sequence, the transformation keeps **generated events** and **program state** unchanged. □

However, in real scenarios, we often do not want the program to be fully equivalent: for example, sometime the order of generated events do not matter to both system state and external state, and leveraging this information could lead to further performance optimizations. Thus, we leave this choice to the user: if the user believes that there is any system state (i.e, a table for counting the number of packets) that does not need to be updated in order or there are external *generate* events that do not need to happen in order, user could annotate the corresponding table and generating operations and Alkali will leverage the information to optimize the generated code.

B Additional Implementation Details

Table 5 shows how different Alkali concepts maps to heterogeneous hardware backends.

FPGA Backend. Compared to CPU architectures, FPGAs exhibit unique computational characteristics. They support massive parallelism and can be thought of as having thousands



(a) **RPC message ressaambly** (b) **JSQ RSS porting process**

Figure 16: **Compare to expert's implementation on Bluefield.**

of micro-cores operating concurrently. To fully leverage this parallelism, the backend exploits both coarse-grained parallelism at the handler level and fine-grained parallelism within each handler.

The backend first exploits coarse-grained parallelism where each handler and its replicas are mapped to an independent hardware module, ensuring that handlers can execute in parallel. Within each handler, the backend exploits fine-grained parallelism by mapping each IR operator to a dedicated hardware execution unit (*e.g.,* subtractor, adder, CAM) and converting IR's SSA values into pipelined FIFO queues that interconnect these execution units. This design maximizes parallel execution by enabling continuous data flow between IR operators.

The FIFO queues between IR operators ensure that the generated hardware design meets timing constraints. While this approach may introduce some redundant pipelining structures, we observe that the additional resource usage remains minimal compared to the available FPGA resources. Future work will focus on optimizing resource utilization to further improve efficiency.

FPGA mapping constraints (Table 1) differ from those of SoC-based NICs. We do not impose a limit on the number of compute units in the FPGA. For memory layers and sizes, we set the constraint based on the total available BRAM and DRAM on the FPGA.

C Additional Evaluation Results

Compare with a BlueField expert's implementation. We compare Alkali's message reassembly and JSQ RSS performance with a BlueField expert's implementation. The expert takes 14 hours to implement these two programs and tuning performance as much as possible. Alkali adopts the unmodified source as for other backends. As shown in Figure 16, Alkali's message reassembly performance is slightly lower than the expert's implementation (5.4% to 9.8%), this small gap is because the expert smartly chooses to reuse the buffer for the first packet in a message to store the aggregated data, avoiding allocation overhead. This buffer reuse is a BlueField backend-specific optimization that could be supported in the future. For the JSQ RSS, the NIC experts tried multiple versions of the program to test different mappings and pipelining methods, while Alkali generated code automatically generates the pipeline and achieves a performance comparable to the expert's best implementation.

Scaling IP Lookup to Large Databases using the CRAM Lens

Robert Chang[1] Pradeep Dogga[1] Andy Fingerhut[2] Victor Rios[1] George Varghese[1]

[1]*University of California, Los Angeles* [2]*Cisco Systems*

Abstract

Wide-area scaling trends require new approaches to Internet Protocol (IP) lookup, enabled by modern networking chips such as Intel Tofino [35], AMD Pensando [2], and Nvidia BlueField [55], which provide substantial ternary content-addressable memory (TCAM) and static random-access memory (SRAM). However, designing and evaluating scalable algorithms for these chips is challenging due to hardware-level constraints. To address this, we introduce the CRAM (CAM+RAM) lens, a framework that combines a formal model for evaluating algorithms on modern network processors with a set of optimization idioms. We demonstrate the effectiveness of CRAM by designing and evaluating three new IP lookup schemes: RESAIL, BSIC, and MASHUP. RESAIL enables Tofino-2 to scale to 2.25 million IPv4 prefixes—likely sufficient for the next decade—while a pure TCAM approach supports only 250k prefixes, just 27% of the current global IPv4 routing table. Similarly, BSIC scales to 390k IPv6 prefixes on Tofino-2, supporting 3.2 times as many prefixes as a pure TCAM implementation. In contrast, existing state-of-the-art algorithms, SAIL [83] for IPv4 and HI-BST [65] for IPv6, scale to considerably smaller sizes on Tofino-2.

1 Introduction

For many, Internet Protocol (IP) lookup is considered a challenge of the past. With over 40 years of research and hundreds of papers (e.g., [7, 19, 21, 22, 29, 45, 65, 83, 89]) focused on supporting IP lookup at scale, numerous schemes have been developed—some of which have been in practical use for two decades. However, these classical approaches are *single-resource* solutions, designed for conventional switch chip architectures that provided either specialized hardware like ternary content-addressable memory (TCAM) or commodity random-access memory (RAM), such as on-chip static RAM (SRAM) coupled with off-chip dynamic RAM (DRAM), but not both. TCAM enables parallel searches across wildcarded entries in a single clock cycle but requires three times more transistors per bit than SRAM and consumes hundreds of watts [5]. RAM-based approaches are cheaper but require additional complexity and memory compared to pure TCAM solutions. Thus far, commercial switch chip vendors have scaled these *single-resource* solutions by increasing hardware resources.

In this paper, we contend that it is important to reconsider IP lookup due to the continued growth of lookup tables and a recent inflection point in network hardware. A slew of new application-specific integrated circuits (ASICs), such as Intel Tofino [35], AMD Pensando [2], and Nvidia BlueField [55], have transformed the networking chip market [58]. These chips are built on two modern packet processing architectures—Reconfigurable Match-Action Tables (RMT) [9] and disaggregated RMT (dRMT) [15]—which consist of match-action processors with access to large amounts of *both* TCAM and SRAM. We review these architectures at the start of §2. This leads us to our central question: *How can we leverage modern networking chips, utilizing both TCAM and SRAM, to develop new IP lookup algorithms that scale to larger databases than classical single-resource solutions?*

Two main challenges make designing scalable algorithms for RMT and dRMT chips difficult: **(1)** Lack of an abstract model for evaluating and comparing algorithms. Chip-specific arcana such as memory allocation, metadata storage, and action bits must be carefully considered. **(2)** Large but finite resources. While TCAM, SRAM, and pipeline stages are available in substantial amounts, they require careful algorithm design to scale effectively.

Solution: We introduce the CRAM (CAM+RAM) lens, an abstract model of modern packet processing architectures, such as RMT and dRMT, paired with a set of optimization idioms. The CRAM model enables us to estimate algorithm scalability using higher-order space and time metrics, without requiring simulation of ASIC-specific details such as TCAM block sizes, SRAM page sizes, and per-stage memory. The CRAM model goes beyond classical models such as random-access machine (RAM) [23] and parallel RAM (PRAM) [41] by adding TCAM operations and using match-action dependencies to measure time complexity. The optimization idioms provide eight strategies for designing scalable algorithms.

CRAM can be generalized to other hardware architectures, such as smart network interface cards (SmartNICs) [2, 32, 55] and field-programmable gate arrays (FPGAs) [3, 34], and applied to broader network applications like packet classification [30, 44, 74] and in-network machine learning (ML) [73, 91, 93]. However, this is not the focus of our paper. For completeness, we briefly discuss these extensions in §2.4 and §2.5. Instead, we concentrate on applying CRAM to *IP lookup* because of the following observations:

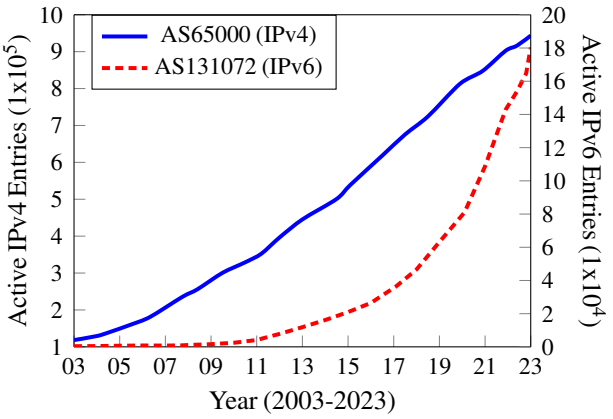


Figure 1: BGP routing table size over the past two decades

O1. Continued IPv4 growth: Over the past two decades, the global IPv4 routing table has grown *linearly* [26, 27], doubling in size every decade (Figure 1). If this trend continues, the IPv4 table could reach two million entries by 2033.

O2. Rapid IPv6 deployment: In the same period, the global IPv6 routing table has grown *exponentially* [25, 28], doubling every three years (Figure 1). Even if growth slows to a linear rate, the IPv6 table could still reach half a million entries by 2033. IPv6 prefixes are also four times wider, though typically, only the first 64 bits are used for global routing.

O3. Virtual private networks (VPNs): Some routers maintain hundreds of VPN routing tables. On such devices, publicly available routing tables account for only a fraction of the total capacity required.

O4. Other tasks: Routers need table memory for additional tasks such as network address translation (NAT) and firewalls. Minimizing the memory used for forwarding allows more features to fit on a chip.

CRAM enables us to rethink pre-existing IP lookup schemes, such as SAIL [83], DXR [89], and multibit tries, to develop new algorithms that scale to larger databases. We start with the best-of-breed algorithms from three classic IP lookup approaches—search on prefix *lengths*, search on prefix *ranges*, and trie-based search. We derive new algorithms using the optimization idioms (§2.2) and predict their scalability with the CRAM model (§2.1). Each new algorithm—rethinking SAIL (RESAIL), Binary Search with Initial CAM (BSIC), and MASHUP—offers unique strengths for different settings.

We chose Intel Tofino-2, an RMT switch chip, for our experiments because we had access to its development environment. While Intel recently announced it will not develop new Tofino models, it remains committed to supporting Tofino and Tofino-2 [53]. We expect our results to hold for dRMT, as RMT is a stricter version of dRMT with additional access restrictions.

This paper makes the following contributions:

1. IP Lookup Algorithms: Three new scalable IP lookup algorithms—RESAIL, BSIC, and MASHUP.

2. CRAM Model: An abstract model for quickly estimating the scalability of packet processing algorithms before detailed implementation.

3. Optimization Idioms: Eight design strategies for minimizing TCAM, SRAM, and pipeline stages.

4. Evaluation: Simulations and Tofino-2 implementations achieve 9X (IPv4) and 3X (IPv6) improvements over pure TCAM solutions, enabling scalability for the next decade.

The remainder of this paper is organized as follows. §2 introduces the CRAM model and optimization idioms, followed by an overview of the idioms in action. It also briefly explores other hardware architectures, broader network applications, and algorithmic requirements. We design three new IP lookup algorithms in § 3, § 4, and § 5. § 6 presents simulation and implementation results, followed by scalability experiments in §7. §8 evaluates the predictive accuracy of the CRAM model, and §9 surveys related work. We conclude in §10.

2 The CRAM Lens

We now formally introduce the CRAM model, list eight optimization idioms, and preview the idioms in action. Additionally, we briefly discuss how CRAM can be generalized to other hardware architectures, applied to broader network applications, and adapted to algorithmic requirements.

The CRAM model abstracts two modern packet processing architectures, RMT [9] and dRMT [15]. Figures depicting both can be found in Appendix A.1. A list of known RMT and dRMT implementations is provided in Appendix A.2.

RMT: RMT is a sequential pipeline architecture of match-action stages. TCAM and SRAM are partitioned among stages such that a stage cannot access the memory of other stages.

dRMT: In contrast, dRMT features programmable processors that execute match-action operations in any order. It disaggregates memory from processors by relocating TCAM and SRAM into a shared external pool.

2.1 The CRAM Model

The CRAM model adds two extensions to the RAM model [23]: first, the ability to perform an SRAM *or* TCAM table lookup; second, an explicit dependency structure between steps (as in RMT compilers [37]) that models the ability to execute multiple steps in parallel. Our goal is for the memory and run time measures of a CRAM model program to be within a small constant factor of the measures for actual hardware implementations.

Thus, a CRAM model program is parameterized by:

- A register size w, and a set R of (w-bit) registers. Let C denote the set of w-bit integers in the range $[0, 2^w - 1]$.
- Sets of unary ($Uops$) and binary ($Bops$) operators on w-bit values, e.g., $Uops = \{+, -, \sim, !\}$ and $Bops = \{+, -, <<, >>, ==, !=, <, \leq, >, \geq, \&, |, \hat{}, \&\&, ||\}$, with behavior as defined in languages like Java and P4 [58].

A CRAM model program consists of a parser P, a deparser D, and a directed acyclic graph G comprised of *steps*. A state S is a function from R to C. P is a function from all bit sequences representing packets to an initial state. D is a function from a final state back to all bit sequences representing packets.

A *step* may optionally begin with a single table lookup operation. A table t consists of a match kind (exact or ternary), a key selector function K_t, a maximum number of entries n_t, and a default value Z_t. K_t's result is a sequence of k_t bits, each representing a chosen bit position within one register of R. An entry e contains a key and associated data, with the d_t bits of associated data stored in a set of w-bit registers A_t.

For an exact match table, the key is a k_t-bit integer. A special case arises for exact match tables with $n_t = 2^{k_t}$, in which the key does not need to be explicitly stored, as it can be used to directly index into the table. For a ternary match table, the key is a pair of k_t-bit integers, a value v_e and a mask m_e, plus an integer priority p_e. All keys in the same table must be distinct.

A step consists of an optional table t followed by a sequence of statements in the form $if(cond) : dest = expr$. Here, $dest$ is an element of R, $expr$ contains a single unary or binary operator with operands from $R \cup A_t \cup C$, and $cond$ is a potentially nested expression with operands from $R \cup A_t \cup C$. No data dependencies are allowed within this sequence, i.e., for any statement in the sequence that assigns a value to $r \in R$, r may not appear in $cond$ or $expr$ of any later statements. This enables all statements within a step to be executed in parallel.

A step reads register r if any bit of r appears in the output of its key selector function K_t, or as part of $cond$ or $expr$ in any of its statements. A step writes register r if r appears as $dest$ in any of its statements.

For all steps u and v in G, if u writes r and v reads or writes r, then there must be a directed path (u,v) or (v,u). This prevents u and v from being executed in parallel. This condition must hold for all registers. A directed path (u,v) indicates that step u must be executed before step v. If there is no directed path between two steps, they may be executed in parallel.

The CRAM model introduces a set of higher-order space and time metrics. The memory footprint of a CRAM model program is evaluated by calculating the total TCAM and SRAM bits across all tables t in G. In a ternary (exact) match table, the memory used for the keys is $n_t k_t$ TCAM (SRAM) bits. For ternary match tables, we only count the v_e component of the key, as these are the logical bits involved in the match. For both types of tables, the memory used for the associated data is $n_t d_t$ SRAM bits. To compute the overall TCAM and SRAM totals, simply sum the bits used across all tables. The latency of a CRAM model program is evaluated by determining the number of steps (nodes) in the longest directed path in G.

2.2 Optimization Idioms

The following idioms can be applied together in various combinations to achieve different space-time trade-offs:

I1. Compress with TCAM: Entries containing wildcards must be expanded to fit into SRAM. For example, the prefix 1** would be stored as 100, 101, 110, and 111. However, by utilizing TCAM, these four SRAM entries can be compressed into a single TCAM entry (1**), thus saving nine bits.

I2. Expand to SRAM: In the dual of *I1*, replace a TCAM block with SRAM if the expanded forms of its prefixes are less than a small constant factor c of the original TCAM entries. We choose $c = 3$ because TCAM requires three times more transistors per bit than SRAM [82].

I3. Compress with SRAM: Despite their high memory cost, directly indexed data structures such as next hop arrays are used because they avoid the extra instructions needed for hashing. However, since most RMT and dRMT ASICs are designed with the cost of performing SRAM-based lookups—whether by hashing or direct indexing—being exactly the same, it is often more advantageous to use compressed forms of SRAM storage such as hash tables instead.

I4. Strategic Cutting: If several entries at a given node share a common prefix, we can save memory by strategically cutting at the bit position where the shared prefix ends, storing only one copy of the repeated bits. While this is how multibit tries [70] work, we extend the concept to TCAM nodes.

I5. Table Coalescing: To reduce memory waste, minimally populated logical tables can be coalesced in shared physical TCAM blocks or SRAM pages. They can be differentiated with tag bits [66]. Although tagging increases the lookup key width, it minimizes physical TCAM and SRAM fragmentation.

I6. Look-aside TCAM: IP lookup schemes are often optimized around common cases such as 24-bit IPv4 prefixes. As a result, uncommon entries (e.g., extremely short or long prefixes) tend to require undue computational or storage costs. We address this by moving the special prefixes into a separate look-aside TCAM that can be trivially searched in parallel.

I7. Step Reduction: A program's number of steps can be reduced by leveraging match-action unit (MAU) parallelism to consolidate data-independent lookups into a single stage.

I8. Memory Fan-out: In traditional RAM model architectures, a lookup table can be accessed multiple times per packet. However, many RMT and dRMT chips restrict each table to one memory access per packet. To address this limitation, we split the original table by fanning out its contents and storing entries accessed by different lookups in separate tables.

2.3 Idioms in Action

We briefly preview the idioms in action for three fundamental classes of IP lookup: search on prefix *lengths*, search on prefix *ranges*, and trie-based search.

From SAIL to RESAIL: Search on prefix *lengths* [10, 21, 40, 77, 83] splits IP lookup into two sub-problems: finding the *length* of the longest match and retrieving the *next hop*. SAIL [83], the best performing IPv4 lookup scheme in hardware settings with fast on-chip SRAM

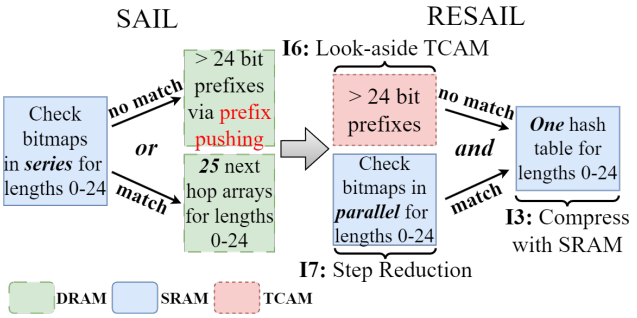


Figure 2: From SAIL to RESAIL via CRAM idioms

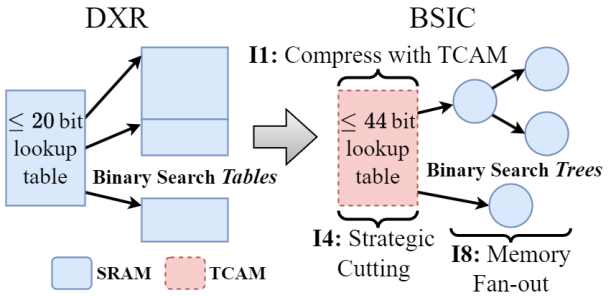


Figure 3: From DXR to BSIC via CRAM idioms

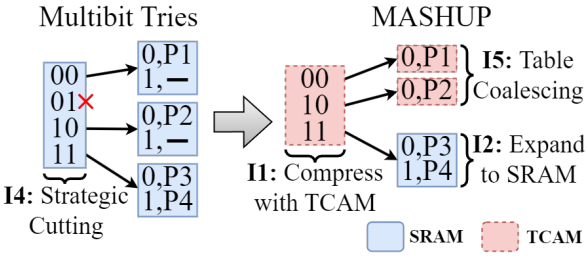


Figure 4: From multibit tries to MASHUP via CRAM idioms

and cheap off-chip DRAM, uses a bitmap of length 2^L to determine whether there is a matching prefix of length L. This works well for prefixes of up to length 24 (the vast majority); for prefixes longer than 24 bits, SAIL uses a complex scheme called pivot pushing that requires excessive prefix expansion [70]. In Figure 2, to obtain RESAIL (rethinking SAIL), we: *I6)* Move the small number of prefixes longer than 24 bits into a separate look-aside TCAM. *I3)* Compress all the next hop arrays into a single hash table using a standard encoding trick [76]. *I7)* Use MAU (stage) parallelism to reduce the number of steps by performing the bitmap lookups in parallel.

From DXR to BSIC: Search on prefix *ranges* [29,45,72,85,89] represents prefixes as range endpoint pairs (e.g., 0**→[000, 011]). Finding the longest matching prefix becomes equivalent to finding the smallest range that encompasses the lookup key. DXR [89], the fastest IPv4 software implementation of range-based searches, uses an initial lookup table to split the search space into multiple smaller binary search tables. In Figure 3, to obtain BSIC (Binary Search with Initial CAM), we: *I1)* Replace the SRAM-based initial lookup table, which supports up to 20-bit prefixes due to direct indexing, with a TCAM-based table that can store prefixes of up to 44 bits (Tofino-2 TCAM block width). *I8)* Replace each binary search *table* with a binary search *tree* that can be fanned out across stages. *I4)* Strategically cut the initial lookup table to balance TCAM required against binary search depth.

From multibit tries to MASHUP: For trie-based search [7,8,19,22,31,54,70], we specifically focus on multibit tries—tries that examine multiple bits, known as a stride, per lookup. We do not consider state-of-the-art *compressed* trie schemes like Poptrie [7] and Tree Bitmap [22], because in the CRAM model, one can directly compress with TCAM without the extra computational and storage costs of bitmap compression. Figure 4 shows a standard multibit trie for the prefixes P1 = 000*, P2 = 100*, P3 = 110*, and P4 = 111*, with a 2-bit stride at the root and a 1-bit stride at the next level—chosen by strategic cutting *(I4)* to minimize the number of downstream pointers. In Figure 4, to obtain MASHUP (mashup of CAM and RAM nodes), we: *I1)* Replace the SRAM root node with a TCAM node to eliminate the empty 01 entry, and do the same for the two upper-right nodes. *I2)* Leave the bottom-right node as SRAM, as it has no wasted space. *I5)* Coalesce the two upper-right TCAM nodes using tag bits (not shown). While the improvement is minimal in this simple example, § 5.1 demonstrates significant gains for large databases.

Although the CRAM versions of these classical schemes may seem simple, they require new algorithms to determine, for example, where to make strategic cuts and which nodes to coalesce. We elaborate on these details when describing RESAIL, BSIC, and MASHUP in §3, §4, and §5, respectively.

2.4 CRAM for other Hardware Architectures

CRAM can be generalized to hardware architectures beyond RMT and dRMT as follows: the space and time metrics of an algorithm specified in the CRAM model serve as lower bounds on the corresponding costs in any implementation, whether in programmable switch ASICs [16, 35, 52, 56], SmartNICs [2, 32, 55], FPGAs [3, 34], or purpose-built fixed-function ASICs designed solely to execute that algorithm. A faithful implementation of a CRAM algorithm achieves a minimum latency equal to that of the longest (critical) path in its directed acyclic graph. While an implementation may have a longer latency, it cannot be shorter. Similarly, the number of bits required may match or exceed the amount specified by the CRAM model, but it cannot be less.

2.5 CRAM for broader Network Applications

Although this paper focuses on IP lookup, we believe CRAM applies to other memory-intensive network applications. These include packet classification (with Access Control Lists (ACLs) and Quality of Service (QoS) as specific

instances), measurement algorithms (such as sketching), regular expression matching, and in-network ML.

In packet classification [30, 44, 74], packet headers are matched against a classifier, where the highest-priority match determines whether to allow or deny traffic, enforce a QoS policy, or apply a custom action. Measurement algorithms [17, 42, 68, 87], by contrast, dynamically build and update a stateful database that tracks network statistics such as per-flow counters, traffic volume, or frequency estimates. Regular expression matching [43, 50, 78] compares unstructured data streams against a database of predefined patterns, often represented as finite automata [14], to detect keywords, signatures, and anomalies. Lastly, in-network ML [73, 91, 93] performs inference by matching a feature vector against a classification model database, which contains decision rules mapping extracted feature values to inference labels.

In practice, these applications often rely on combinations of common data structures, such as decision trees, Bloom filters [11], tries, hash tables, and bitmaps, many of which we demonstrate how to optimize with CRAM. Consequently, our optimization idioms naturally extend beyond IP lookup. For example, the careful balancing of TCAM compression (*I1*) and SRAM expansion (*I2*) used in MASHUP to create a hybrid trie, can similarly be applied to packet classification [47, 67] and in-network ML [39, 79] algorithms that rely on decision trees. Likewise, the look-aside TCAM (*I6*) in RESAIL, which captures longer prefixes, can serve a similar role in offloading other specialized cases, such as multi-field wildcard classification rules [30], heavy-hitter flows with rare protocols [68], multi-line attack patterns [78], and fast-patch updates for classification models [92]. A more in-depth exploration of CRAM's broader applicability is left for future work.

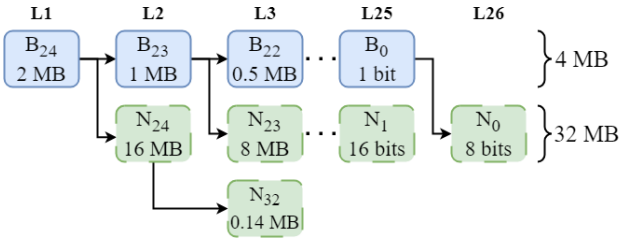
2.6 Other Algorithmic Requirements

Certain algorithmic requirements introduce additional considerations. We briefly examine how CRAM applies to algorithms that require atomic memory updates, stateful data plane operations, and pseudo-random keys.

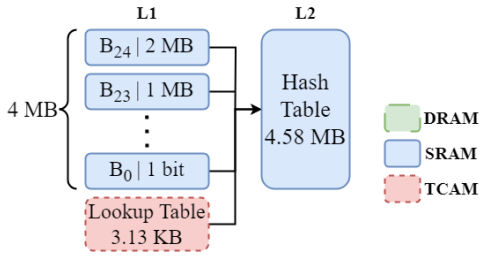
Atomic memory updates: CRAM neither facilitates nor hinders an algorithm's ability to support atomic updates [14, 46, 61]. The CRAM model provides an abstraction to estimate *performance*, not an execution model to predict *implementation feasibility*. If an algorithm requiring atomic updates can be implemented, the CRAM model should accurately predict latency and memory.

Stateful data plane operations: P4 register arrays are the primary mechanism for stateful operations in the data plane, as used in [6, 13, 68]. Stateful operations can be incorporated into the CRAM model by introducing a new SRAM-based register match table, and counting these memory bits separately alongside regular TCAM and SRAM bits.

Pseudo-random keys: For algorithms with pseudo-random keys [12, 36, 86], the efficacy of some CRAM optimization idioms is clearly reduced, as uniformly distributed random bytes are difficult, if not impossible, to compress. In contrast, other idioms are unaffected by key distribution. Specifically, table coalescing (*I5*), look-aside TCAM (*I6*), step reduction (*I7*), and memory fan-out (*I8*) can still be applied. Nevertheless, the benefits of the CRAM model are likely to be significantly diminished, as lack of compression eliminates a major source of optimization.



(a) CRAM model representation of SAIL



(b) CRAM model representation of RESAIL (*min_bmp*=0)

Figure 5: SAIL vs RESAIL for IPv4 prefixes in AS65000

3 RESAIL

By applying the optimization idioms to SAIL [83], we create a new IPv4 CRAM lookup algorithm called RESAIL. Refer to Figure 2 for the intuition to which we now add details.

SAIL Review: SAIL designates 24 as a pivot level and divides the forwarding information base (FIB) into short prefixes ($\leq$24) and long prefixes (>24). SAIL determines whether there is a length-i match for $i \leq 24$ by consulting a bitmap B_i of size 2^i in which bit p is set if and only if p is a length-i prefix in the FIB. Since the total memory footprint of the bitmaps is 4 MB, they are stored in on-chip SRAM. If a match is found in B_i, the next hop is retrieved by directly indexing into a next hop array N_i of size 2^i located in off-chip DRAM. SAIL handles prefixes of length $i > 24$ by using a complex scheme called pivot pushing that expands [70] them to 32-bit entries in N_{32}.

3.1 Applying the Optimization Idioms

We show the CRAM derivation of RESAIL using the IPv4 AS65000 BGP routing table (Sep 2023). Start with the CRAM model representation of SAIL in Figure 5a. To obtain RESAIL in Figure 5b, use the idioms as follows:

Entry	Prefix (Ternary)	Output Port
1	010100**	A
2	011*****	B
3	100100**	C
4	100101**	D
5	10010100	A
6	10011010	B
7	10011011	C
8	10100011	A

Table 1: Example routing table

Index	Key	Value
0	1001001	C
1	0101001	A
2	0111000	B
3	-	-
4	1001011	D

Table 2: Hash table for Table 1 (pivot level = 6)

1. Observe a large number (26) of data dependencies between the bitmaps and next hop arrays. This makes sense in the RAM model as it enables early exits which reduce average execution time. However, these are false dependencies because their lookup keys can be computed in parallel. Therefore, we apply step reduction (*I7*) to reduce all the bitmap and next hop array lookups into a single step. The next hop can then be determined by taking the highest priority result.

2. SAIL relies on a significant amount of DRAM (32 MB) for its directly indexed next hop arrays. Since DRAM is not available in CRAM, we replace the next hop arrays with a more compact data structure by either compressing with TCAM (*I1*) or compressing with SRAM (*I3*). Since the 25% memory penalty of d-left hashing [10] is less expensive than TCAM's 3X higher area cost, we compress with SRAM by replacing the next hop arrays with a single SRAM-based hash table.

3. SAIL uses a special next hop array N_{32} for prefixes of length >24, which are very uncommon. Its entries are prefix expanded to 32 bits. In the worst case, a single prefix may be expanded into 2^8 duplicate next hops. We address this memory inefficiency by replacing N_{32} with a look-aside TCAM (*I6*) that can store prefixes of length >24 without additional expansion.

4. The number of bitmaps serves as a trade-off between the amount of parallelism required and the hash table's memory footprint. For RESAIL, we introduce a parameter *min_bmp* that represents the smallest bitmap available. In Figure 5b, *min_bmp* is 0 which means there are a total of 25 bitmaps from B_{24} down to B_0. Increasing *min_bmp* reduces the number of parallel lookups at the cost of increased SRAM usage.

3.2 Building the Data Structures

Look-aside TCAM: Given a routing database, add all prefixes longer than 24 bits to the look-aside TCAM. Since there are very few IPv4 prefixes of length >24, little TCAM is used.

Bitmaps: For $i = 24$ down to $i = min_bmp$, construct a bitmap (B_i) of length 2^i such that every prefix of length i in the routing database is marked as a 1 at the corresponding index. If *min_bmp* is not equal to 0, use prefix expansion [70] to combine B_0 to B_{min_bmp-1} into B_{min_bmp}. Start with length min_bmp-1 prefixes and work down linearly to length 0. A bit in B_{min_bmp} is flipped from 0 to 1 only if the bit is already a 0. This prevents incorrectly overwriting longer prefixes.

Hash Table: We use d-left [10] for the hash table because it has a low probability of collision even when the ratio of entries to memory is as high as 80%. Hashing the matched prefix directly would require a separate hash table for each length from *min_bmp* to 24, greatly fragmenting memory. Instead, RESAIL uses a standard trick [76] we call bit marking that enables us to generate hash keys of a fixed length. When an entry is added to bitmap B_i, its unique 25-bit hash key is produced by appending a 1 and left shifting by $24-i$ bits. Each hash key is paired with its next hop and inserted into the hash table. The boundary of each prefix can be determined by scanning from the right for the first 1. In effect, bit marking removes the need for multiple hash tables.

Table 2 shows a hash table using 7-bit hash keys for Table 1. For simplicity, this example assumes a pivot level of 6 and a maximum prefix length of 8. Since entries 5-8 from Table 1 are longer than the pivot length, they are not placed into the hash table (they are in the look-aside TCAM instead). The hash table has a size of 5 due to d-left's 25% memory penalty. 011, a 3-bit entry, is appended with a 1 and left shifted 3 times, thus resulting in the hash key 0111000.

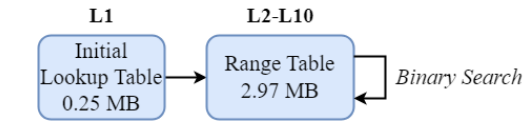
3.3 Performing Lookups

Start by performing two sets of lookups in parallel: **(1)** In the look-aside TCAM, perform a longest prefix match with the full 32-bit IPv4 address. **(2)** From B_{24} ($i = 24$) down to B_{min_bmp}, perform exact match lookups using the first i bits of the destination address to directly index into B_i.

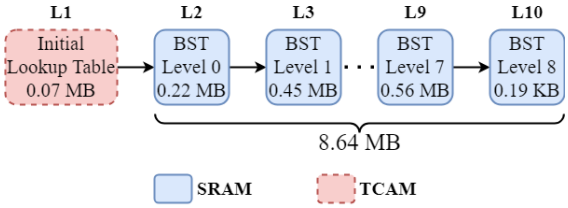
If a match is found in the look-aside TCAM, return its associated next hop. Otherwise, take the longest match across all bitmaps and generate its 25-bit hash key by bit marking.

At this step, either the next hop for a prefix match greater than 24 bits has been found or the final hash key has been created. In the latter case, use the hash key to perform an exact match lookup into the hash table to retrieve the next hop.

Algorithm 1 in Appendix A.5 contains pseudocode for RESAIL lookups. Appendix A.3.1 describes incremental updates, deletions, and insertions in RESAIL.



(a) CRAM model representation of DXR (k=16)



(b) CRAM model representation of BSIC (k=16)

Figure 6: DXR vs BSIC for IPv4 prefixes in AS65000

4 BSIC

By applying the optimization idioms to DXR [89], we create a new CRAM lookup algorithm called BSIC, capable of supporting both IPv4 and IPv6. Refer to Figure 3 for the intuition to which we now add details.

DXR Review: Inspired by [29] and [45], DXR performs binary search on range endpoints using a range table (binary search table). To reduce the depth of binary search, DXR uses an initial lookup table directly indexed by the first k bits of the address. The lookup table returns a pointer to the subsection of the range table in which binary search will be performed. DXR adds two optimizations: **(1)** Merging neighboring ranges that point to the same next hop. **(2)** Discarding right endpoints.

4.1 Applying the Optimization Idioms

We show the CRAM derivation of BSIC using the IPv4 AS65000 BGP routing table (Sep 2023). Start with the CRAM model representation of DXR in Figure 6a. As recommended by [89], set $k=16$ (D16R) for the best IPv4 results. To obtain BSIC in Figure 6b, use the idioms as follows:

1. Since DXR's initial lookup table relies on direct indexing, leaving many entries unused, we compress with TCAM (*I1*). Replacing the SRAM-based initial lookup table with a TCAM-based one reduces its memory consumption by over 3X, from 0.25 MB of SRAM to 0.07 MB of TCAM.

2. In DXR, the range table is repeatedly accessed during binary search. Since lookup tables are limited to a single access per packet in the CRAM model, the range table must be split up. We do so through memory fan-out (*I8*). By converting the range table into multiple binary search trees (BSTs) and distributing search levels across separate tables accessed at different steps, we ensure each table is visited at most once per packet. However, this greatly increases the amount of memory needed because every internal node has to store up to two pointers. In Figure 6, DXR's range table uses only 2.97 MB of SRAM while BSIC's BST levels use 8.64 MB of SRAM—a 2.9X increase. Although costly, memory fan-out is essential because duplicating the entire range table for each search level would require an infeasible amount of SRAM (26.73 MB).

Key	Value	BST Entries (for reference)
0101	Pointer to BST 1	00**
011*	Next Hop B	-
1001	Pointer to BST 2	00**, 01**, 0100, 1010, 1011
1010	Pointer to BST 3	0011

Table 3: Initial lookup table for Table 1 ($k=4$)

3. The parameter k is a strategic cut (*I4*) that balances memory usage in the initial lookup table against the number of required BST levels. Due to the high memory cost of direct indexing, DXR's SRAM-based lookup table is limited to $k<=20$. For example, if $k=24$, DXR's initial lookup table would consume 64 MB of SRAM. To effectively support IPv6, which has longer prefixes, a larger k value is required. Since TCAM can store wildcard entries without prefix expansion, BSIC's TCAM-based initial lookup table can use much larger k values, up to the underlying TCAM block width ($k=44$ for Tofino-2).

4.2 Building the Data Structures

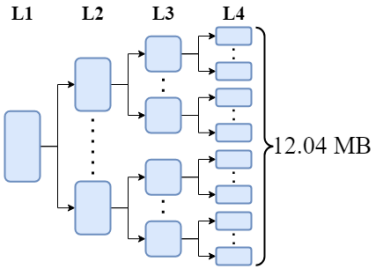
Initial Lookup Table: Given a routing database and a slice size k, populate the initial lookup table by storing all prefixes as unique k-length slices. Duplicate slices are condensed into one entry. Three cases arise when adding a prefix p of length l:

1. If $l < k$, pad p with $k-l$ wildcard (*) bits. Its associated table value is p's next hop.
2. If $l == k$, do not modify p. If there are longer prefixes that share the same k-length slice as p, its associated table value is a pointer to the corresponding BST's root node. Otherwise, its associated table value is p's next hop.
3. If $l > k$, trim p down to k bits. Its associated table value is a pointer to the corresponding BST's root node.

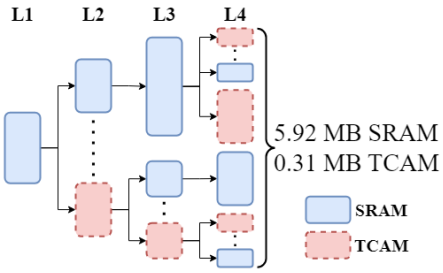
Table 3 shows an initial lookup table with k=4 created using Table 1. The maximum prefix length is 8. The Key column contains all the k-length slices while the Value column stores the associated pointers and next hops. The BST Entries column shows the prefix segments that are pointed to by the slices. Since entries 3-7 in Table 1 share the same k-length slice, they are condensed into a single key 1001 that points to BST 2.

Binary Search Trees (BSTs): To create a BST for a given lookup table entry, identify all prefixes in the database that match the entry up to the kth bit. For all such prefixes, store the remaining bits and next hops as tuples in a list. Take the list of tuples and perform the range expansion and optimizations described in [89]. We defer these details to Appendix A.4.

Use the resulting list of left endpoints to construct a BST in which every node contains four fields: pointers to the left and



(a) CRAM model representation of multibit trie (16-4-4-8)



(b) CRAM model representation of MASHUP (16-4-4-8)

Figure 7: Multibit trie vs MASHUP for IPv4 prefixes in AS65000

right child, the next hop, and the left endpoint itself. Repeat this process for all lookup table entries containing pointers.

4.3 Performing Lookups

Start in the initial lookup table by performing a longest prefix match using the first k bits of the destination address. If either a next hop is returned or a miss occurs, search terminates. If a pointer to a BST is returned, follow the pointer to the corresponding root node and form the next search key by extracting the remaining bits of the destination address. Once at a node, perform standard binary search using the search key.

Algorithm 2 in Appendix A.5 contains pseudocode for BSIC lookups. Appendix A.3.2 describes incremental updates, deletions, and insertions in BSIC.

5 MASHUP

By applying the optimization idioms to multibit tries [70], we create a new CRAM lookup algorithm called MASHUP, capable of supporting both IPv4 and IPv6. Refer to Figure 4 for the intuition to which we now add details.

Multibit Trie Review: Multibit tries are search tries that examine multiple bits per lookup, known as a stride. Reducing the number of strides decreases the number of worst-case memory accesses, but increases prefix expansion and memory usage. We assume each tree level has exactly one stride.

5.1 Applying the Optimization Idioms

We show the CRAM derivation of MASHUP using the IPv4 AS65000 BGP routing table (Sep 2023). Start with the CRAM model representation of a multibit trie in Figure 7a. We find strides 16-4-4-8 yield the best IPv4 results (§6.3). To obtain MASHUP in Figure 7b, use the idioms as follows:

1. For each trie node, we consider both compressing with TCAM (*I1*) and expanding to SRAM (*I2*). If the increase in memory due to prefix expansion [70] is less than 3X, we use SRAM. Otherwise, we use TCAM. This results in a hybrid trie with both TCAM and SRAM nodes, as seen in Figure 7b.

2. Once the trie is hybridized, apply table coalescing (*I5*) by merging partially filled nodes[1] of the same memory type into super-tables, compactly mapping them onto contiguous TCAM blocks or SRAM pages with minimal fragmentation. This requires prepending entries with a tag [66] to distinguish between logical tables. A tag of width x can coalesce 2^x logical tables into one super-table. The combination of node hybridization and table coalescing reduces SRAM usage from 12.04 MB to 5.92 MB at the cost of 0.31 MB of TCAM (Figure 7).

3. The set of strides is a parameter that serves as a strategic cut (*I4*). For a given set of strides, the trie's memory overhead is directly proportional to the number of internal pointers. A simple method for choosing strides, explained in §6.3, is to analyze the database's prefix length distribution.

We omit standard algorithms for building the MASHUP trie, as the process is identical to constructing a multibit trie, which has been extensively studied in prior work [70, 76].

5.2 Performing Lookups

Let S_i represent the i-th stride. Start in the root node by performing a lookup with the first S_0 bits of the destination address. If the current node is TCAM, perform a longest prefix match. Otherwise, perform an exact match. If a miss occurs, terminate the search. If a hit occurs, three values may be returned: a next hop, a pointer to the next node, and a unique tag. If a next hop is returned, save it. Form the lookup key for level i by extracting the next S_i bits of the destination address and prepending the current tag. Repeat the lookup process until either a leaf node is reached or a miss occurs. Upon termination, return the saved next hop.

Algorithm 3 in Appendix A.5 contains pseudocode for MASHUP lookups. Appendix A.3.3 describes incremental updates, deletions, and insertions in MASHUP.

6 Results

In this section, we introduce the databases, target implementations, and parameter values that we selected for our experiments. We use the CRAM metrics to determine the

[1] For MASHUP in §6.4, we greedily fill the largest tables with the smallest ones. This approach is easy to implement, but possibly suboptimal.

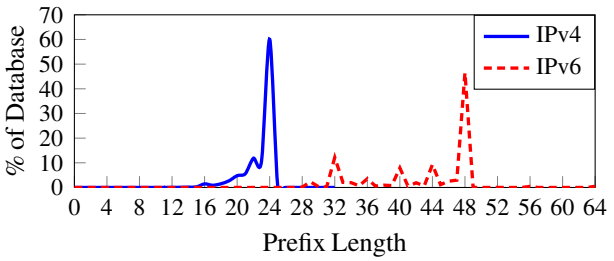


Figure 8: IPv4 and IPv6 prefix length distributions in AS65000 and AS131072, respectively, for September 2023

best IPv4 (RESAIL) and IPv6 (BSIC) algorithms *before* implementation. We then compare the resource utilizations of the best CRAM algorithms with those of state-of-the-art IP lookup schemes *after* implementation.

6.1 Databases

For IPv4, we used the AS65000 BGP routing table (Sep. 2023) [27] which has close to 930k IPv4 prefixes. For IPv6, we used the AS131072 BGP routing table (Sep. 2023) [28] which has close to 190k IPv6 prefixes. Figure 8 shows their prefix length distributions. We identify three key patterns:

P1. Prefix distributions have major and minor spikes. For IPv4, there is a major spike at length 24 and minor spikes at lengths 16, 20, and 22. For IPv6, there is a major spike at length 48 and minor spikes at lengths 28, 32, 36, 40, and 44.

P2. The majority of IPv4 prefixes are longer than 12 bits.

P3. The majority of IPv6 prefixes are longer than 28 bits.

6.2 Target Implementations

We obtained results for two different targets: an ideal RMT chip and Intel Tofino-2 (also RMT). Since RMT is a stricter variant of dRMT with additional access restrictions, we expect our RMT results to be reproducible on a dRMT chip.

Ideal RMT Chip (Simulation): We define an ideal RMT chip to be an RMT chip with Tofino-2 specifications (same memory, number of stages, etc.) [57] that can achieve 100% SRAM utilization and perform at least two dependent ALU operations per stage. The resource utilization for an ideal RMT chip is obtained through simulation by using Tofino-2 SRAM page (128x1024b) and TCAM block (44x512b) sizes. If the number of TCAM blocks or SRAM pages used by a table exceeds the amount available in a MAU (stage), the table is simply partitioned across multiple MAUs. Since Tofino-2 has 20 MAUs, results that require over 20 are considered infeasible.

Intel Tofino-2 (Implementation): The resource utilization for Tofino-2 is obtained through implementation. We implement the best CRAM algorithms using P4 and compile them with the Intel P4 compiler. P4 Insight [33] then outputs detailed resource mappings and visualizations specific to Tofino-2.

Scheme	TCAM Bits	SRAM Bits	Steps
MashUp (16-4-4-8)	0.31 MB	5.92 MB	4
BSIC (*k*=16)	0.07 MB	8.64 MB	10
RESAIL (*min_bmp*=13)	3.13 KB	8.58 MB	2

Table 4: CRAM metrics for IPv4 prefixes in AS65000

Scheme	TCAM Bits	SRAM Bits	Steps
MashUp (20-12-16-16)	0.32 MB	0.77 MB	4
BSIC (*k*=24)	0.02 MB	3.18 MB	14

Table 5: CRAM metrics for IPv6 prefixes in AS131072

6.3 Parameter Values

We choose parameter values based on observations from §6.1.

RESAIL's key parameter is *min_bmp*, the smallest bitmap available. We choose $min_bmp = 13$ because there are so few IPv4 prefixes shorter than 13 bits **(P2)**, thus minimizing the amount of prefix expansion needed.

BSIC's key parameter is k, the initial slice size. As recommended by [89], for IPv4, we choose $k = 16$. For IPv6, we choose $k = 24$ because most IPv6 prefixes are longer than 28 bits **(P3)**. Therefore, a k value that is close to but smaller than 28 can compress over 190k prefixes into just 7k TCAM entries. We briefly explore other choices of k for IPv6 and examine potential latency-memory trade-offs in Appendix A.6.

MashUp's key parameter is its set of strides. Intuitively, we want to select strides that mirror the distribution spikes **(P1)** seen in Figure 8 because they will minimize prefix expansion. For IPv4, we choose 16-4-4-8 (spikes at 16, 20, 24). For IPv6, we choose 20-12-16-16 (spikes at 32, 48). We do not select 32 as the first stride because it is too wide—especially for the root node which may contain many entries. Therefore, we decompose 32 into separate strides of 20 and 12.

6.4 Comparisons *before* Implementation

Recall that CRAM metrics enable quick estimation of algorithm scalability *before* implementation.

We present IPv4 and IPv6 CRAM metrics in Table 4 and Table 5, derived from the optimization steps in §3.1, §4.1, and §5.1, for the IPv4 and IPv6 BGP tables, respectively.

For IPv4, RESAIL outperforms BSIC in all three CRAM metrics. Between RESAIL and MashUp, RESAIL wins in TCAM and steps but loses in SRAM. However, MashUp requires 100X more TCAM than RESAIL, whereas RESAIL requires only 1.4X more SRAM than MashUp. Therefore, we determine RESAIL to be the best CRAM IPv4 algorithm.

For IPv6, we choose between BSIC and MashUp. BSIC wins in TCAM but loses in SRAM and steps. MashUp requires 16X more TCAM than BSIC, while BSIC requires roughly 4X more SRAM and steps than MashUp. As before, we prioritize TCAM because it is more expensive and limited

Scheme	TCAM Blocks	SRAM Pages	Stages
MashUp (16-4-4-8)	235	216	10
BSIC (*k*=16)	74	558	16
RESAIL (*min_bmp*=13)	2	556	9

Table 6: Ideal RMT mapping for IPv4 prefixes in AS65000

Scheme	TCAM Blocks	SRAM Pages	Stages
MashUp (20-12-16-16)	178	47	8
BSIC (*k*=24)	15	211	14

Table 7: Ideal RMT mapping for IPv6 prefixes in AS131072

than SRAM—for example, Tofino-2 contains 19X more SRAM than TCAM. Although BSIC uses more steps than MashUp, this is due to BSIC's use of BSTs, which have a high initial step cost. Therefore, we determine BSIC to be the best CRAM IPv6 algorithm *for Tofino-2*. However, for more stage-constrained ASICs, MashUp is likely better.

To verify the validity of the CRAM metrics, we explicitly map each CRAM algorithm to an ideal RMT chip and present its resource utilization in Table 6 and Table 7. This mapping, which accounts for Tofino-2 TCAM block sizes, SRAM page sizes, and per-stage memory, is precisely the complicated process that the CRAM model seeks to relieve algorithm designers of. Comparing Table 4 with Table 6 and Table 5 with Table 7, observe that the CRAM metrics accurately predict a target algorithm's resource utilization and potential scalability.

6.5 Comparisons *after* Implementation

The previous subsection compared our three new algorithms *before* implementation using CRAM metrics. Here, we compare the best CRAM algorithms *after* implementation on Tofino-2 against the best pre-existing IPv4 and IPv6 schemes.

6.5.1 Baseline Selection

We select four single-resource baselines: SRAM-only for IPv4 and IPv6, and TCAM-only for IPv4 and IPv6.

SRAM-only for IPv4: We choose SAIL [83] as our SRAM-only IPv4 baseline due to its on-chip memory bound for short prefixes, which enables it to scale very well. Although IPv4 schemes like Poptrie [7] and DXR [89] use less memory, they require too many memory accesses and stages.

SRAM-only for IPv6: We choose Hi-BST [65] as our SRAM-only IPv6 baseline because it is the most memory-efficient IPv6 lookup algorithm to date [90]. It uses a treap data structure that maps each prefix to a unique node.

TCAM-only for IPv4 and IPv6: We choose a logical TCAM as our TCAM-only IPv4 and IPv6 baseline because, although TCAM-oriented schemes exist for reducing power consumption [48] or merging multiple FIBs [51], none focus on scaling IP lookup for a single database.

Scheme	TCAM Blocks	SRAM Pages	Stages	Target Chip
RESAIL (*min_bmp*=13)	17	750	16	Tofino-2
RESAIL (*min_bmp*=13)	2	556	9	Ideal RMT
SAIL	-	2313	33	Ideal RMT
Logical TCAM	1822	-	76	Ideal RMT
Tofino-2 Pipe Limit	480	1600	20	-

Table 8: Baseline comparison for IPv4 prefixes in AS65000

Scheme	TCAM Blocks	SRAM Pages	Stages	Target Chip
BSIC (*k*=24)	15	416	30	Tofino-2
BSIC (*k*=24)	15	211	14	Ideal RMT
Hi-BST	-	219	18	Ideal RMT
Logical TCAM	762	-	32	Ideal RMT
Tofino-2 Pipe Limit	480	1600	20	-

Table 9: Baseline comparison for IPv6 prefixes in AS131072

6.5.2 IPv4 Comparison

Table 8 compares the ideal RMT resource utilization of RESAIL with the IPv4 baselines. RESAIL requires 911X fewer TCAM blocks than the logical TCAM and approximately 4X fewer SRAM pages and stages than SAIL. Although SAIL's memory scales efficiently, its high upfront cost makes it impractical for RMT-like chips. RESAIL outperforms the logical TCAM, which only supports IPv4 databases of up to 245,760 entries—about 3.8X smaller than the current IPv4 BGP table.

Table 8 also compares RESAIL on an ideal RMT chip and on Tofino-2. While RESAIL fits on Tofino-2 for AS65000, it requires nearly 1.4X more SRAM pages and 2X more stages than on an ideal RMT chip. This is because Tofino-2 reserves bits in each SRAM word for identifying actions, limiting the maximum SRAM utilization to 50%. The increase in TCAM is due to extra ternary bitmask tables needed for extracting bits.

6.5.3 IPv6 Comparison

Table 9 compares the ideal RMT resource utilization of BSIC with the IPv6 baselines. BSIC uses less SRAM and fewer stages than Hi-BST, at the cost of 15 TCAM blocks. Both BSIC and Hi-BST support the current IPv6 BGP table, whereas the logical TCAM only supports up to 122,880 entries—about 1.6X smaller than the current IPv6 BGP table.

Table 9 also compares BSIC on an ideal RMT chip and on Tofino-2. Our ideal RMT chip assumes each stage can perform at least two dependent ALU operations. However, in practice, a Tofino-2 stage can execute only one level of ALU logic. Consequently, each BST level requires two stages: one for comparing the search key and another for performing the P4 action. This creates a repeating pattern where an SRAM-intensive stage is followed by a stage with minimal

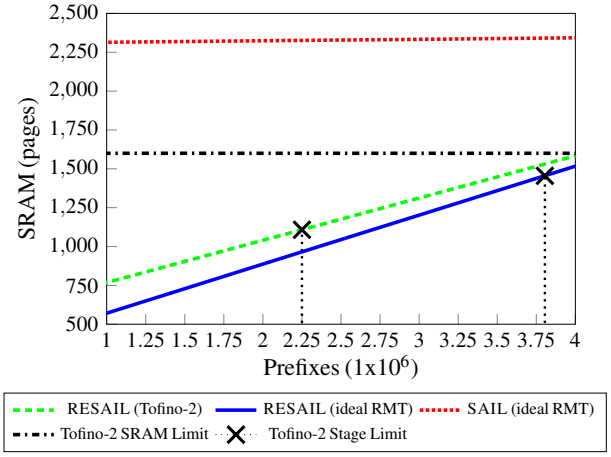


Figure 9: RESAIL vs SAIL scaling (IPv4)

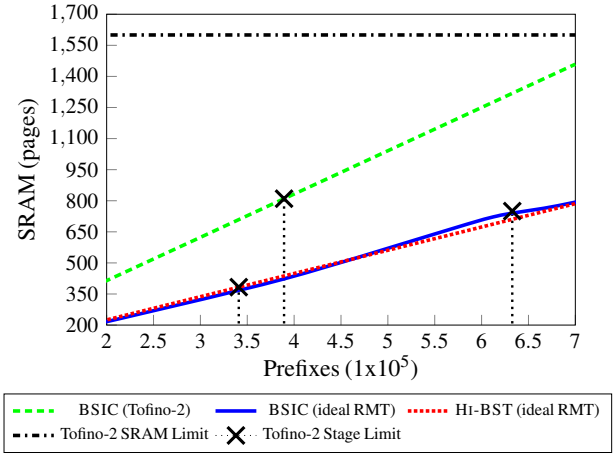


Figure 10: BSIC vs HI-BST scaling (IPv6)

SRAM usage. Although BSIC on Tofino-2 requires 30 stages (ten over the Tofino-2 pipe limit), we successfully fit BSIC for AS131072 by recirculating each packet. However, this effectively halves the number of available switch ports.

7 Scalability

While §6 presents results for *current* BGP tables, this section presents scalability analysis for RESAIL and BSIC on larger *synthetic* routing databases. We omit scalability analysis for MASHUP because it requires too much TCAM (for Tofino-2).

7.1 IPv4 Scaling

Figure 9 shows IPv4 scalability results for RESAIL and our SRAM-only IPv4 baseline, SAIL. We did not generate synthetic prefixes because the resource utilization of RESAIL and SAIL depends on the distribution of *prefix lengths* rather than the distribution of the prefixes themselves. From the perspective of memory usage, RESAIL and SAIL do not distinguish between prefixes of identical length. Therefore, we use a simple scaling model that applies a constant scaling factor to all prefix lengths. For the ideal RMT results of RESAIL and SAIL, we use the steps described in §6.2 to calculate their new utilization. For the Tofino-2 results of RESAIL, we update the corresponding P4 table sizes to reflect the larger databases.

For ideal RMT, SAIL is infeasible because its SRAM cost far exceeds the Tofino-2 SRAM limit. At any given database size, RESAIL for Tofino-2 uses more SRAM than RESAIL for ideal RMT. This is expected since Tofino-2 does not allow 100% SRAM utilization. Notably, RESAIL on an ideal RMT chip scales to around 3.8 million prefixes, 4X larger than the current IPv4 BGP table. RESAIL on Tofino-2 scales to around 2.25 million prefixes, 2.3X larger than the current routing database and significantly beyond SAIL's capacity.

7.2 IPv6 Scaling

Figure 10 shows IPv6 scalability results for BSIC and our SRAM-only IPv6 baseline, HI-BST. For BSIC, we generated synthetic prefixes because its resource utilization depends on the distribution of prefixes and sub-prefixes. To obtain worst-case scalability results, observe that the first three bits of IPv6 prefixes in AS131072 are 000—forming an *IPv6 universe*. We use different combinations of these bits to generate significantly larger synthetic databases from AS131072, an approach we call *multiverse scaling*. Multiverse scaling assumes that the distribution of all prefix lengths scales uniformly. In practice, customer scaling causes some prefixes (e.g., /48s) to scale more rapidly than others (e.g., /24s). However, this stresses only the BSTs and not the initial TCAM, unlike multiverse scaling which models worst-case results for TCAM, SRAM, and stages. For HI-BST, we use the memory calculation provided in [65].

As seen in Figure 10, both instances of BSIC are able to out-scale HI-BST. For an ideal RMT chip, HI-BST only scales to around 340k prefixes, 1.8X larger than the current IPv6 BGP Table. Even though HI-BST is the most memory efficient IPv6 lookup scheme, it requires too many stages. Comparing the two instances of BSIC, we see that BSIC for ideal RMT scales to around 630k prefixes, 3.3X larger than the current routing database. Since BSIC for Tofino-2 uses over 2X more stages, it scales to around 390k prefixes—2X the size of the current IPv6 BGP table.

8 CRAM Model Evaluation

How predictive was the CRAM model? §6.4 showed that the CRAM model accurately predicted RESAIL and BSIC as the best algorithms for IPv4 and IPv6, respectively. We now examine the CRAM metrics in more detail.

Table 10 and Table 11 show results for RESAIL and BSIC on three models: the CRAM model, an ideal RMT model, and a Tofino-2 implementation. The three models form a hierarchy

Scheme	TCAM Blocks	SRAM Pages	Steps (Stages)	Model
RESAIL (*min_bmp*=13)	1.14	549.12	2	CRAM
RESAIL (*min_bmp*=13)	2	556	9	Ideal RMT
RESAIL (*min_bmp*=13)	17	750	16	Tofino-2

Table 10: Predictive accuracy of CRAM for RESAIL (IPv4)

Scheme	TCAM Blocks	SRAM Pages	Steps (Stages)	Model
BSIC (*k*=24)	7.45	203.52	14	CRAM
BSIC (*k*=24)	15	211	14	Ideal RMT
BSIC (*k*=24)	15	416	30	Tofino-2

Table 11: Predictive accuracy of CRAM for BSIC (IPv6)

of abstractions with increasing detail. We scale the CRAM metrics found in Table 4 and Table 5 from raw bits to TCAM blocks and SRAM pages to allow for uniform comparisons. The three models can be understood as follows:

CRAM model: Using the CRAM metrics (raw bits and dependent steps), an algorithm designer can quickly predict scalability *without* seeing the product data sheet.

Ideal RMT model: This model allows for more precise predictions but requires a *basic* understanding of the data sheet, specifically the general organization of memory and stages.

Tofino-2 implementation: This is the most accurate model but also the most complex to develop. It accounts for low-level details that are hard to glean from data sheets, such as action bits and ALU operations per stage. This often requires an *expert* with intimate knowledge of the product.

Consider the predictive accuracy for RESAIL in Table 10 as we move from CRAM to ideal RMT. The TCAM and SRAM measures reflect small rounding errors due to unit conversion. However, the latency increases significantly from 2 steps to 9 stages because, unlike dRMT, RMT stages provide *both* memory and processing—to support 556 RAM pages, more stages are required even when no additional processing is needed.

Next, consider the predictive accuracy for RESAIL in Table 10 as we move from ideal RMT to Tofino-2. There is a small additive increase in TCAM blocks due to extra ternary tables required for implementing RESAIL in P4. Additionally, SRAM pages increase by a factor of 1.35, and stages increase by a factor of 1.78. As discussed earlier in § 6.5.2, this is because the maximum achievable SRAM utilization on Tofino-2 is 50%, necessitating more SRAM pages and stages.

The predictive accuracy for BSIC in Table 11 can be interpreted similarly. The key difference is that the ∼2X increase in SRAM pages and stages from ideal RMT to Tofino-2 is due to the fact that implementing 3-way branching on Tofino-2 requires two stages for each BST level.

Based on our limited experience, the CRAM model provides a useful, easily computed initial model for estimating algorithm scalability. Although its measures of space and time are off by small constant factors, this is no different from Big O notation.

9 Related Work

Models of Computation: Abstract models like RAM [23], PRAM [41], and Turing machines [62] are widely used to analyze algorithms. The RAM model abstracts sequential computers, while the PRAM model abstracts shared memory multiprocessors. Our CRAM model abstracts network processors with two types of memory, parallelism, and programmability.

Combinations of CAM and RAM: Earlier CAM and RAM combinations optimize different metrics. CoolCAM [88], cooler TCAM [49], and EaseCAM [60] all reduce power consumption. Liu [48] compacts routing tables to reduce cost, power consumption, and thermal dissipation. Luo [51] merges FIBs in virtual routers to reduce TCAM usage. Compaction and merging are orthogonal to our ideas. Other hybrid approaches [38, 69, 75, 84] target tasks such as packet classification. In summary, no existing solution optimizes scalability for IP lookup by leveraging *both* TCAM and SRAM.

10 Conclusion

Our paper introduces new algorithms for IP lookup made possible by strategically leveraging *both* CAM and RAM. These algorithms address the scaling challenges of global routing tables and are well-suited to the architectures of modern network processors. For these processors, the CRAM lens provided a fresh perspective on algorithm design. Much like the RAM [23] and PRAM [41] models, CRAM offers metrics for quickly evaluating algorithm feasibility *before* implementation. Using the CRAM model, we developed three new IP lookup algorithms: RESAIL, BSIC, and MASHUP. RESAIL and BSIC scale to much larger databases than the best existing IPv4 and IPv6 lookup schemes, respectively, while MASHUP excels in stage-constrained hardware environments.

We aim to establish CRAM's generality by applying it to other hardware architectures (§ 2.4) and network applications (§ 2.5)—helping to "cram" more packet processing power into each unit of chip area. Ultimately, our findings underscore a simple insight for networking chip vendors: *a little TCAM goes a long way. Adding small amounts of TCAM to supplement SRAM can significantly improve scalability.*

Acknowledgments

We thank our shepherd, Xiaoqi Chen, and the anonymous reviewers for their valuable feedback. We are also grateful to David Maltz and Vladimir Gurevich for their guidance. This research was supported in part by NSF grant CNS-2333587.

References

[1] The Fungible DPU™: A New Category of Microprocessor for the Data-Centric Era : Hot Chips 2020. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–25, 2020.

[2] Advanced Micro Devices, Inc. AMD Pensando Networking. https://www.amd.com/en/products/accelerators/pensando.html.

[3] Advanced Micro Devices, Inc. Vitis Networking P4. https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html.

[4] Anurag Agrawal and Changhoon Kim. Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32, 2020.

[5] B. Agrawal and T. Sherwood. Modeling TCAM power for next generation network devices. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 120–129, 2006.

[6] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 29–43, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Hirochika Asai and Yasuhiro Ohara. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. *SIGCOMM Comput. Commun. Rev.*, 45(4):57–70, August 2015.

[8] Masanori Bando, Yi-Li Lin, and H. Jonathan Chao. FlashTrie: beyond 100-Gb/s IP route lookup using hash-based prefix-compressed trie. *IEEE/ACM Trans. Netw.*, 20(4):1262–1275, August 2012.

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013.

[10] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve ip lookups. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1454–1463 vol.3, 2001.

[11] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485 – 509, 2003.

[12] Xiaoqi Chen. Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, SPIN '20, page 8–14, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 15–29, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Xiaoqi Chen, Andrew Johnson, Mengying Pan, and David Walker. Synthesizing state machines for data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '22, page 81–88, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 1–14, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Cisco Systems, Inc. Cisco Silicon One - Network Processors for Unified Architecture - Cisco. https://www.cisco.com/site/us/en/products/networking/silicon-one/index.html.

[17] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[18] Rajdeep Das and Alex C Snoeren. Memory Management in ActiveRMT: Towards Runtime-programmable Switches. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 1043–1059, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. *SIGCOMM Comput. Commun. Rev.*, 27(4):3–14, October 1997.

[20] Kevin Deierling. NVIDIA's Resource Transmutable Network Processing ASIC. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–14, 2023.

[21] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using bloom filters. In *Proceedings of the 2003 Conference on*

Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '03, page 201–212, New York, NY, USA, 2003. Association for Computing Machinery.

[22] Will Eatherton, George Varghese, and Zubin Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, April 2004.

[23] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, page 240–249, New York, NY, USA, 1998. Association for Computing Machinery.

[24] Michael Galles and Francis Matus. Pensando Distributed Services Architecture. *IEEE Micro*, 41(2):43–49, 2021.

[25] Geoff Huston. AS131072 - IPv6 BGP Table Statistics. https://bgp.potaroo.net/v6/as2.0/index.html.

[26] Geoff Huston. AS65000 - BGP Table Statistics. https://bgp.potaroo.net/as2.0/.

[27] Geoff Huston. CIDR Report. https://www.cidr-report.org/as2.0/.

[28] Geoff Huston. IPv6 CIDR Report. https://www.cidr-report.org/v6/as2.0/.

[29] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, volume 3, pages 1240–1247 vol.3, 1998.

[30] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[31] B. Indira, K. Valarmathi, and D. Devaraj. A Trie based IP Lookup Approach for High Performance Router/Switch. In *2019 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS)*, pages 1–6, 2019.

[32] Intel Corporation. Intel® Infrastructure Processing Unit (Intel® IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[33] Intel Corporation. Intel® P4 Insight Tool. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html.

[34] Intel Corporation. Intel® P4 Suite for FPGAs. https://www.intel.com/content/www/us/en/software/programmable/p4-suite-fpga/overview.html.

[35] Intel Corporation. Intel® Tofino™ 2. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html.

[36] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 103–115, Oakland, CA, May 2015. USENIX Association.

[38] Changhun Jung, Sian Kim, Rhongho Jang, David Mohaisen, and DaeHun Nyang. A Scalable and Dynamic ACL System for In-Network Defense. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1679–1693, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Radhakrishna Kamath and Krishna M Sivalingam. Machine Learning based Flow Classification in DCNs using P4 Switches. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10, 2021.

[40] Kun Suk Kim and S. Sahni. IP lookup by binary search on prefix length. In *Proceedings of the Eighth IEEE Symposium on Computers and Communications. ISCC 2003*, pages 77–82 vol.1, 2003.

[41] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990.

[42] A. Kumar and J. Xu. Sketch Guided Sampling - Using On-Line Estimates of Flow Size for Adaptive Data Collection. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–11, 2006.

[43] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium*

on Architecture for Networking and Communications Systems, ANCS '07, page 155–164, New York, NY, USA, 2007. Association for Computing Machinery.

[44] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for advanced packet classification with ternary CAMs. *SIGCOMM Comput. Commun. Rev.*, 35(4):193–204, August 2005.

[45] B. Lampson, V. Srinivasan, and G. Varghese. IP lookups using multiway and multicolumn search. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, volume 3, pages 1248–1256 vol.3, 1998.

[46] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: a better NetFlow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 311–324, USA, 2016. USENIX Association.

[47] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 256–269, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Huan Liu. Routing table compaction in ternary CAM. *IEEE Micro*, 22(1):58–64, 2002.

[49] Haibin Lu. Improved trie partitioning for cooler TCAMs. In *IASTED International Conference on Advances in Computer Science and Technology*. Citeseer, 2004.

[50] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte Regular Expression Matching with Speculation. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, page 284–303, Berlin, Heidelberg, 2009. Springer-Verlag.

[51] Layong Luo, Gaogang Xie, Steve Uhlig, Laurent Mathy, Kavé Salamatian, and Yingke Xie. Towards TCAM-based scalable virtual routers. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, page 73–84, New York, NY, USA, 2012. Association for Computing Machinery.

[52] Marvell Technology, Inc. Data Center Switches | Enabling the AI cloud with High-bandwidth, low-latency, multi-terabit network switches. https://www.marvell.com/products/data-center-switches.html.

[53] Nick McKeown. Intel's commitment to P4. https://groups.google.com/a/lists.p4.org/g/p4-announce/c/frXi_jjmawE.

[54] S. Nilsson and G. Karlsson. Fast address look-up for internet routers. In *Proceedings of the IFIP TC6/WG6.2 Fourth International Conference on Broadband Communications: The Future of Telecommunications*, BC '98, page 11–22, GBR, 1998. Chapman & Hall, Ltd.

[55] Nvidia Corporation. NVIDIA BlueField Networking Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[56] Nvidia Corporation. The Spectrum Ethernet Platform. https://www.nvidia.com/en-us/networking/products/ethernet/.

[57] P4 Compiler Project. p4c/backends/tofino/bf-p4c at main · p4lang/p4c · GitHub. https://github.com/p4lang/p4c/tree/main/backends/tofino/bf-p4c, 2025.

[58] P4 Project. P4 Ecosystem – P4 – Language Consortium. https://p4.org/ecosystem/?_product_category=hardware.

[59] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

[60] V.C. Ravikumar, R.N. Mahapatra, and Laxmi Narayan Bhuyan. EaseCAM: an energy and storage efficient TCAM-based router architecture for IP lookup. *IEEE Transactions on Computers*, 54(5):521–533, 2005.

[61] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 323–334, New York, NY, USA, 2012. Association for Computing Machinery.

[62] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1997.

[63] Alexandru Seibulescu and Mario Baldi. Leveraging P4 Flexibility to Expose Target-specific Features. In *Proceedings of the 3rd P4 Workshop in Europe*, EuroP4'20, page 36–42, New York, NY, USA, 2020. Association for Computing Machinery.

[64] D. Shah and P. Gupta. Fast updating algorithms for TCAM. *IEEE Micro*, 21(1):36–47, 2001.

[65] Tong Shen, Xian Yu, Gaogang Xie, and Dafang Zhang. High-Performance IPv6 Lookup with Real-Time Updates Using Hierarchical-Balanced Search Tree. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, 2018.

[66] Sandeep Sikka and George Varghese. Memory-efficient state lookups with fast updates. *SIGCOMM Comput. Commun. Rev.*, 30(4):335–347, August 2000.

[67] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, page 213–224, New York, NY, USA, 2003. Association for Computing Machinery.

[68] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 164–176, New York, NY, USA, 2017. Association for Computing Machinery.

[69] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended TCAMs. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, pages 120–131, 2003.

[70] V. Srinivasan and George Varghese. Faster IP lookups using controlled prefix expansion. *SIGMETRICS Perform. Eval. Rev.*, 26(1):1–10, June 1998.

[71] Naru Sundar, Brad Burres, Yadong Li, Dave Minturn, Brian Johnson, and Nupur Jain. 9.4 An In-depth Look at the Intel IPU E2000. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 162–164, 2023.

[72] Subhash Suri, G. Varghese, and P.R. Warkhede. Multiway range trees: scalable IP lookup with fast updates. In *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*, volume 3, pages 1610–1614 vol.3, 2001.

[73] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 329–342, New York, NY, USA, 2023. Association for Computing Machinery.

[74] David E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, September 2005.

[75] Balajee Vamanan and T. N. Vijaykumar. TreeCAM: decoupling updates and lookups in packet classification. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, New York, NY, USA, 2011. Association for Computing Machinery.

[76] George Varghese. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[77] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookups. *SIGCOMM Comput. Commun. Rev.*, 27(4):25–36, October 1997.

[78] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 631–648, USA, 2019. USENIX Association.

[79] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. Mousika: Enable General In-Network Intelligence in Programmable Switches by Knowledge Distillation. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1938–1947, 2022.

[80] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, Renton, WA, April 2022. USENIX Association.

[81] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. Unleashing SmartNIC Packet Processing Performance in P4. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 1028–1042, New York, NY, USA, 2023. Association for Computing Machinery.

[82] Rui Yang, Haitong Li, Kirby K. H. Smithe, Taeho R. Kim, Kye Okabe, Eric Pop, Jonathan A. Fan, and H.-S. Philip Wong. Ternary content-addressable memory with MoS2 transistors for massively parallel data search. *Nature Electronics*, 2(3):108–114, Mar 2019.

[83] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X. Liu, Qi Li, and Laurent Mathy. Guarantee IP lookup performance with FIB explosion. *SIGCOMM Comput. Commun. Rev.*, 44(4):39–50, August 2014.

[84] Ruyi Yao, Cong Luo, Xuandong Liu, Ying Wan, Bin Liu, Wenjun Li, and Yang Xu. MagicTCAM: A Multiple-TCAM Scheme for Fast TCAM Update. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11, 2021.

[85] Changhoon Yim, Bomi Lee, and Hyesook Lim. Efficient binary search for IP address lookup. *IEEE Communications Letters*, 9(7):652–654, 2005.

[86] Sophia Yoo and Xiaoqi Chen. Secure Keyed Hashing on Programmable Switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable Network INfrastructure*, SPIN '21, page 16–22, New York, NY, USA, 2021. Association for Computing Machinery.

[87] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 29–42, USA, 2013. USENIX Association.

[88] F. Zane, Girija Narlikar, and A. Basu. Coolcams: power-efficient TCAMs for forwarding engines. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, volume 1, pages 42–52 vol.1, 2003.

[89] Marko Zec, Luigi Rizzo, and Miljenko Mikuc. DXR: towards a billion routing lookups per second in software. *SIGCOMM Comput. Commun. Rev.*, 42(5):29–36, September 2012.

[90] Chunyang Zhang and Gaogang Xie. Using XorOffsetTrie for high-performance IPv6 lookup in the backbone network. *Computer Communications*, 181:438–445, 2022.

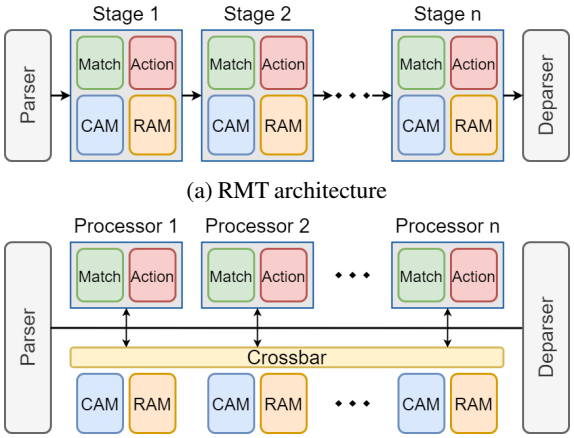
[91] Changgang Zheng, Xinpeng Hong, Damu Ding, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. In-Network Machine Learning Using Programmable Network Devices: A Survey. *IEEE Communications Surveys & Tutorials*, 26(2):1171–1200, 2024.

[92] Changgang Zheng, Zhaoqi Xiong, Thanh T. Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Hybrid In-Network Classification Using Programmable Switches. *IEEE/ACM Trans. Netw.*, 32(3):2555–2570, February 2024.

[93] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Planter: Rapid Prototyping of In-Network Machine Learning Inference. *SIGCOMM Comput. Commun. Rev.*, 54(1):2–21, August 2024.

A Appendix

A.1 Packet Processing Architectures



(a) RMT architecture

(b) dRMT architecture

Figure 11: RMT vs dRMT

A.2 RMT and dRMT Implementations

Product	Architecture	Type	Sources
Intel Tofino-2	RMT	switch ASIC	[4, 18, 35]
Intel Mount Evans (E2000)	RMT	SmartNIC	[32, 71]
AMD Pensando DSC-100	RMT	SmartNIC	[2, 24, 63]
Fungible F1/S1	RMT	SmartNIC	[1]
FlowBlaze	RMT	FPGA	[59]
FlexCore	dRMT	switch ASIC	[80]
Nvidia BlueField-3	dRMT	SmartNIC	[20, 55, 81]

Table 12: Summary of known implementations

While we cannot verify all details in some cases, it is evident that TCAM, SRAM, parallelism, and programmability are present in the sources for the products listed in Table 12.

A.3 Updates, Deletions, and Insertions

A.3.1 RESAIL

Incremental updates, deletions, and insertions for RESAIL are efficient, following the same process as lookups but with modifications to the target entry. For prefixes of length *min_bmp* or greater, only two memory accesses are required (bitmap and hash table). For prefixes shorter than *min_bmp*, the operations are more costly because of prefix expansion. Update operations for prefixes longer than 24 bits are much simpler in RESAIL than in SAIL because we have eliminated pivot pushing.

A.3.2 BSIC

For BSIC, incremental updates, deletions, and insertions are costly and complex due to inherent dependencies between binary search tree levels. A separate database with additional prefix information is needed for rebuilding data structures [89]. If fast update operations are important, RESAIL and MASHUP are better choices.

A.3.3 MASHUP

Incremental updates, deletions, and insertions for MASHUP are nearly identical to lookups, except they modify the target entry. These are standard algorithms [76] for multibit tries that have been well studied. Maintaining a sorted TCAM table under these changes is non-trivial, but effective algorithms exist [64].

A.4 Range Expansion and Optimizations

Range	Next Hop
0000 - 0011	C
0100 - 0100	A
0101 - 0111	D
1000 - 1001	-
1010 - 1010	B
1011 - 1011	C
1100 - 1111	-

Table 13: Range expansion for slice 1001 (Table 3)

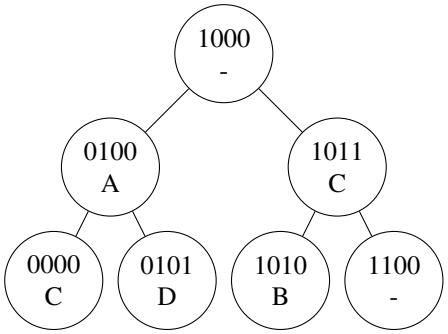


Figure 12: BST for slice 1001 (Table 3)

As in DXR [89], convert all the prefix substrings into ranges by generating their endpoint pairs. Use the endpoint pairs to create sorted, contiguous, and non-overlapping intervals that cover all possible bitstrings of the maximum length. Intervals that are added to complete the full range will "inherit" the next hop of the current lookup table entry's longest prefix match. This is necessary because it is possible for a destination address to be incorrectly directed by the initial lookup table to a BST that does not contain a legitimate match. Therefore, in the case of such a mistake, the search key will land in an interval containing the correct next hop. A simple example of range expansion for slice 1001 from Table 3 is shown in Table 13. Note that the intervals 1000-1001 and 1100-1111 were added to complete the full range. Since there are no valid longest prefix matches for slice 1001, its intervals are assigned a default value of -.

After the full range is created, merge neighboring intervals with the same next hop to reduce the number of nodes required. Discard the right endpoints as they can be inferred from the left ones. Use the remaining left endpoints to create the BST. Figure 12 shows a sample BST for slice 1001.

A.5 Pseudocode

Algorithm 1: RESAIL Lookup ($addr$, min_bmp)

```
Input  : addr, IPv4 address
Input  : min_bmp, smallest bitmap available
Output : hop, next hop
hop ← lookup_table.match(addr)
if hop ≠ None then
    return hop
for i ← 24 ; i ≥ min_bmp ; i−− do
    if B_i.match(addr ≫ (32-i)) == 1 then
        key ← (addr ≫ (32-i)) ≪ (25-i)
        key ← key + (1 ≪ (24-i))
        hop ← hash_table.match(key)
        break
return hop
```

Algorithm 2: BSIC Lookup ($addr$, k)

```
Input  : addr, IPv4 or IPv6 address
Input  : k, initial slice size
Output : hop_best, next hop
level ← 0
len ← 32 if IPv4, 64 if IPv6
hop_best, index
  ← lookup_table.match(addr ≫ (len−k))
if hop_best ≠ None then
    return hop_best
while index ≠ None
    hop, left, right, prefix ← bst_level.match(index)
    if prefix == ((addr ≪ k) ≫ k) then
        return hop
    else if prefix < ((addr ≪ k) ≫ k) then
        index ← right
        hop_best ← hop
    else
        index ← left
    level ← level + 1
return hop_best
```

A.6 Latency-memory trade-offs

A natural question is: *are there latency-memory trade-offs for CRAM algorithms that we can exploit to free up pipeline stages for other processing tasks?* Examining the basic CRAM model, there appears to be a clear trade-off: reducing dependency steps at the expense of increased memory or computation can lead to lower latency.

Algorithm 3: MASHUP Lookup ($addr$, $strides$)

```
Input  : addr, IPv4 or IPv6 address
Input  : strides, set of strides
Output : hop_best, next hop
hop_best ← default
table ← root
tag ← None
index ← 0
level ← 0
while table ≠ None
    key ← addr[index:index+strides[level]]
    hop, next_table, tag ← table.match(tag, key)
    if table.hit() then
        if hop ≠ None then
            hop_best ← hop
        index ← index + strides[level]
        level ← level + 1
        table ← next_table
    else
        break
return hop_best
```

Unfortunately, on real platforms like Tofino-2, *steps* cannot be conflated with *stages* because, in RMT, stages provide a fixed amount of both memory and computation. Therefore, large lookup tables require multiple stages. While a latency-memory trade-off exists for *steps* versus memory in our three CRAM algorithms, no corresponding trade-off exists for *stages* versus memory. Instead, there is an optimal number of stages, beyond which *both* memory and latency increase.

To see this, consider Figure 13 for BSIC (IPv6), in which the only tuning parameter is k—the width of the initial TCAM table. While a latency-memory curve exists, decreasing latency by increasing k actually *increases* the number of stages required. As k grows, the number of stages needed for the initial TCAM table rises significantly, outweighing the reduction in stages gained from decreasing BST depth. In contrast, the basic CRAM model predicts reduced latency as k increases. Unfortunately, this larger TCAM table requires more *stages* but not *steps*.

As shown in Figure 13, the optimal value of k is 24, with both smaller and larger values yielding worse results. Thus, no trade-off exists between stages and memory. This is also the reason we use $k=24$ in our experiments for IPv6.

Similarly, with MASHUP, the strides are the primary tuning parameter. Again, we did not find a useful memory-latency trade-off when considering stages. Finally, RESAIL consistently requires only two steps, with no latency-memory trade-off whatsoever in either the CRAM or ideal RMT models.

While there is no useful latency-memory trade-off for a *fixed* database size in our three algorithms, an obvious trade-off emerges as the database size *increases*: the number of stages (latency) must increase, at least to provide more memory. This is implicit in the linear trade-off curves shown in Figure 9 and Figure 10. Notably, the y-axes in these figures represent SRAM pages. However, since more SRAM requires proportionally more stages, Figure 9 and Figure 10 can be rescaled to (instead) depict stages versus database size, maintaining exactly the same curve shapes.

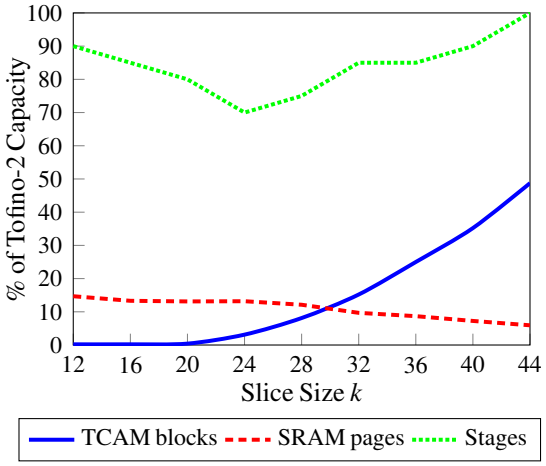


Figure 13: BSIC IPv6 latency-memory trade-off on an ideal RMT chip for AS131072 (Sep 2023)

Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing

Zhenyuan Ruan Shihang Li [†] Kaiyan Fan Seo Jin Park [+] Marcos K. Aguilera [‡]
Adam Belay Malte Schwarzkopf [†]
MIT CSAIL [†]*Brown University* [‡]*VMware Research by Broadcom* [+]*USC*

Abstract. Datacenters today waste CPU and memory, as resources demanded by applications often fail to match the resources available on machines. This leads to stranded resources because one resource that runs out prevents placing additional applications that could consume the other resources. Unusable stranded resources result in reduced utilization of servers, and wasted money and energy.

Quicksand is a new framework and runtime system that unstrands resources by providing developers with familiar, high-level abstractions (*e.g.,* data structures, batch computing). Internally Quicksand decomposes them into *resource proclets*, granular units that each primarily consume resources of one type. Inspired by recent granular programming models, Quicksand decouples consumption of resources as much as possible. It splits, merges, and migrates resource proclets in milliseconds, so it can use resources on any machine, even if available only briefly.

Evaluation of our prototype with four applications shows that Quicksand uses stranded resources effectively; that Quicksand reacts to changing resource availability and demand within milliseconds, increasing utilization; and that porting applications to Quicksand requires moderate effort.

1 Introduction

It is rarely the case that the resources demanded by applications (*e.g.,* compute, memory, etc.) exactly match those available on machines in a datacenter. This mismatch wastes performance and capacity, because as one resource becomes a bottleneck, others are left idle, a notion referred to as *resource stranding* [8, 21, 23, 35, 40, 53]. In an ideal world, applications would use exactly the resources they need without being constrained by their physical availability on each machine.

One idea that could help solve this problem is memory disaggregation—the ability to allocate memory from a common pool that spans across machines. Unfortunately, memory disaggregation forces programmers to give up control of whether they are accessing local or remote memory, which sacrifices performance. This problem is severe enough that cloud providers currently limit their use of memory disaggregation to only the coldest memory pages [32].

Recent granular programming models, by contrast, give the programmer control over where resources are consumed. They decompose applications into granular *units* that bundle together portions of compute and memory, referred to as "proclets" [49], "components" [19], "actors" [39], and "functions" [29]. Each unit resides entirely on a specific machine, so it is guaranteed that state contained within it remains local. Units communicate with each other via the network using remote procedure calls (RPCs), so it is clear to programmers when a memory access could be remote. In some systems, units migrate across machines in response to resource pressure [49]. But this fails to address resource stranding, as the migratable units still couple different resources. Granular programming is also often hard to adopt, as programmers must manually decompose their programs into small units.

Quicksand is a new framework and runtime system that unstrands resources by providing a more performant alternative to disaggregation and offers familiar, high-level programming interfaces to application developers. Quicksand takes inspiration from the idea of granular programming, but crucially lets applications allocate resources of different types largely independently across a rack of machines. Quicksand unstrands resources through three key ideas.

First, Quicksand introduces *resource proclets*, a new abstraction that helps developers specify the use of different resource types in a fine-grained and independent manner. Resource proclets are categorized by the type of resource they primarily use and their functionality: a *compute proclet* handles threads of computation, while a *memory proclet* manages data structures and heap data. Resource proclets can be independently scheduled across machines and efficiently communicate with each other via a low-latency, high-bandwidth datacenter network (e.g., 400 GbE). Compute proclets can efficiently access data from remote memory proclets through prefetching, minimizing the performance impact of cross-machine data access. This allows applications to exploit stranded resources wherever they are available.

Second, Quicksand manages the size of resource proclets so they can be efficiently migrated through the network to rebalance resource use across machines. If a resource proclet becomes too large, it impedes fast migration; and if it becomes too small, communication and metadata overheads will dominate. To address this issue, Quicksand *splits* and *merges* resource proclets to keep them granular: splitting a memory proclet partitions its state into two shards (*e.g.,* by key range of a hash table), and splitting a compute proclet breaks up its work (*e.g.,* loop iterations) into two parts. Merging does the reverse and combines resource proclets, so the right number of proclets emerges dynamically.

Finally, Quicksand provides developers with familiar, high-level libraries while internally managing fine-grained resource allocation through resource proclets, freeing developers from

having to reason about low-level resource management. Developers use these libraries in the same way they use familiar data structures and compute abstractions, such as C++ STL containers and MapReduce-style parallel programming. The libraries internally decompose these high-level constructs into resource proclets. Consequently, Quicksand occupies a sweet spot: its libraries control how applications are decomposed and where memory will be accessed locally by leveraging semantic information. This not only hides the complexity of managing low-level granular primitives but also helps avoid unnecessary and costly remote memory accesses.

To illustrate the power of Quicksand, consider a machine learning pipeline, running on datacenter GPUs shared across applications—a common scenario today [28, 57, 61]. The pipeline preprocesses batches of raw images on CPUs in stages (resizing, noise reduction, etc.) before training a model on the GPUs. We want to keep GPUs fully utilized because they are expensive [31, 37, 38], use CPU and memory wherever available as needed, without having to overprovision them for pre-processing stages. Quicksand's resource proclets enable the pipeline to use stranded resources (*e.g.,* denoise images in memory on another server). To keep GPUs saturated, Quicksand splits and merges resource proclets to adapt its CPU and memory use to changing GPU availability.

Quicksand targets rack-scale environments where machines are typically connected with an overprovisioned network that offers high full-bisectional bandwidth. We prototyped Quicksand, and ported four applications to it: an image processing pipeline for machine learning, a latency-critical service from the DeathStarBench suite [17], a data-intensive sorting tool, and a serverless video encoding system from ExCamera [16]. Experiments show that Quicksand achieves 2–8× better performance than Hermit [45], a state-of-the-art memory disaggregation system, and 2–3× better throughput than Nu [49], a prior granular programming system. Additionally, Quicksand enables millisecond-scale fast reaction to changing load and resource availability by splitting and merging proclets, increasing utilization by 25%–60% compared to baselines that necessitate overprovisioning. Application developers enjoy these benefits with moderate programming effort.

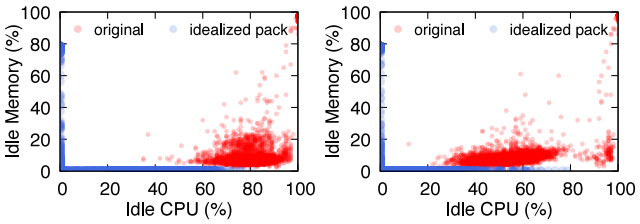
Our prototype has some limitations. It implements only compute and memory proclets, but we envision proclet support for other resource types (e.g., SSDs). Resource proclets are also not inherently fault-tolerant, although existing techniques for fault-tolerant granuluar computing apply [49].

2 Background and Motivation

2.1 Resource Stranding in Datacenters

Resource stranding occurs when hardware resources on machines are underutilized due to a mismatch between the resources demanded by applications and those available, resulting in wasted capacity [8, 21, 35, 40, 53].

We investigate resource stranding by analyzing traces from an Alibaba datacenter [23]. Alibaba colocates batch tasks with high-priority online services to increase utilization. The workload has a strong diurnal pattern [23], so we analyze a demand trough (at t = 500k seconds) and a demand peak (at t = 540k seconds). Figure 1 shows a scatterplot of idle CPU and memory, with each red point representing a specific machine. Even at peak load, machines have an average of 56% idle CPU and 10% idle memory. These idle resources exist because operators overprovision to handle uncertainty about the resources the online services will consume, the first source of stranded resources. This is especially critical for memory, as running out of it leads to crashes. Even though Alibaba already overcommits resources, this approach fails to utilize all available resources, as the level of idle resources is hard to predict and varies over time.



(a) At t = 500k seconds (load trough). **(b)** At t = 540k seconds (load peak).

Figure 1: Resource stranding in an Alibaba datacenter, shown by red points (*original*), arises from overprovisioned resources to accomodate load bursts of online services. Even with theoretically maximal binpacking of batch tasks based on perfect future resource predictions (blue points, *idealized pack*), stranding remains as batch task demands mismatch with available machine resources.

Resource stranding persists even in an idealized scenario without resource overprovisioning. To illustrate this, we simulate an ideal (but impractical) setting where Alibaba can perfectly predict resource needs and densely pack additional batch tasks to maximize utilization. This simulation iteratively fills each machine with batch tasks that have resource requirements and execution times equal to the averages of existing batch jobs in the trace. This continues until no more tasks can be added without exceeding the machine's CPU or memory capacity. This approach is unrealistic because it assumes perfect knowledge of future resource usage, including anticipating load spikes from online services.

Even in this hypothetical setting, some resources remain idle, as shown by the blue points (*idealized pack*) in Figure 1. For instance, under peak load (t = 540k), an average of 14% idle CPU and 3% idle memory persist on machines. These points, clustering along the axes, indicate machines with surplus of one resource type but scarcity of the other. This results from the mismatch between the CPU/memory demand ratios of batch tasks and the availability ratios of those resources on machines, another source of stranded resources.

Resource stranding is a widespread phenomenon: for instance, Microsoft reports up to 30% stranded memory in Azure [8], and Google's datacenters see wasted compute resources because best-effort jobs placed to fill spare compute

encounter out-of-memory events [54].

2.2 Memory Disaggregation

Memory disaggregation can alleviate resource stranding by exposing memory over the network, either from another server or a shared memory pool. Disaggregation can be supported at the kernel layer through paging (*e.g.,* Infiniswap [22], FastSwap [2], Hermit [45]) or at the bus layer through cache coherence (*e.g.,* GenZ [18], CXL [11, 34], LegoOS [51]). Bus-level solutions require new hardware that has not yet been deployed at the time of writing.

Memory disaggregation is appealing as it can be adopted without application changes. However, it faces two major limitations. First, while memory disaggregation facilitates pooling memory across machines, it does not extend this capability to CPUs. Hence, it fails to effectively utilize stranded CPU resources. Second, the transparency of memory disaggregation comes at a performance cost. Any memory access could be local or remote, and remote accesses may introduce delays that stall the CPU. This lack of awareness poses challenges for code optimization, especially problematic when compute intensity (*i.e.,* CPU cycles per byte of data access) is low. For example, Hermit [45] (a state-of-the-art memory disaggregation system) delivers only 9%–11% of the non-disaggregated throughput for a web service application (§7.1).

2.3 Granular Computing Systems

Recent cloud programming trends suggest decomposing applications into granular units, such as "actors" in Ray [39], "proclets" in Nu [49], and "components" in ServiceWeaver [19]. This evolution aligns with the increasing adoption of serverless and microservices paradigms in cloud development. Granular programming can utilize stranded CPU and memory by distributing small, granular units across machines. Unlike transparent memory disaggregation, granular programming explicitly defines unit boundaries; memory accessed within a unit is local, and memory can be accessed remotely only deliberately through remote procedure calls. Hence, developers can optimize their programs by structuring them to operate on local memory when compute intensity is low and exploit remote memory when it is high.

Among granular computing systems, Nu enables fast unit migration across machines to achieve *fungibility*—the ability to use resources interchangeably across machines [49]. However, Nu has major limitations: it still bundles different resources (such as CPU and memory) and fails to avoid resource stranding. Additionally, developing applications in Nu is hard, as developers must manually decompose the application into units and maintain their granularity. Developers may compose units with incorrect granularity, or create initially-granular units that become large over time (*e.g.,* due to memory growth). Improper sizing results in slow migration and imbalanced load, which impacts utilization and performance.

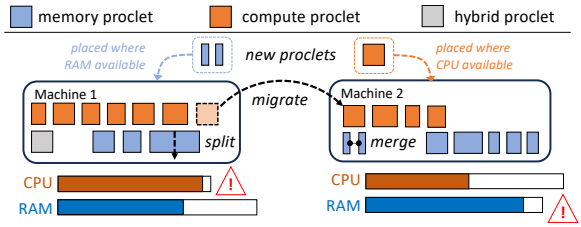


Figure 2: Quicksand places resource proclets where they are available: *e.g.,* memory proclets (blue) on machine 1 with available RAM; and compute proclets (orange) on machine 2 with spare CPU. Quicksand splits, merges, and migrates resource proclets as they grow, shrink, and as resource pressure develops.

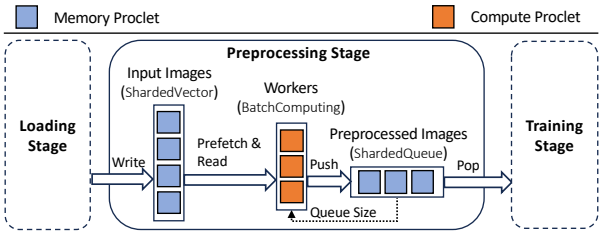
3 Quicksand Overview

Quicksand integrates the principles of resource disaggregation and granular computing in a new system. Quicksand reclaims stranded resources by decoupling resources and aligning allocation with actual use, while making it easy for application developers to use resources wherever they are available.

To dissect the use of different resources, Quicksand introduces *resource proclets*, granular abstractions categorized based on functionality. A compute proclet manages a set of computing threads, while a memory proclet manages megabytes of heap data. Both types of proclet can be dynamically split and merged to ensure a granular size. Each resource proclet functions as an independent unit, with Quicksand's runtime scheduling and migrating it based on application resource demands and availability. Resource proclets efficiently communicate with each other via the datacenter network (when remote) or a fast local data path (when local).

Quicksand places resource proclets on machines based on resource availability. Proclet sizes may change over time. For example, in Figure 2, the application is inserting data into its memory proclets on the left machine, and removing data from its memory proclets on the right machine. Quicksand reacts by dynamically splitting and merging resource proclets to maintain their granularity. In situations where a machine becomes oversubscribed, such as when the left machine's CPU capacity is nearly depleted, Quicksand migrates compute proclets to the right machine where additional CPUs are available. Similarly, Quicksand migrates memory proclets from machines nearing full to those with available space. Additionally, Quicksand supports hybrid proclets (as depicted on the left machine) for scenarios that require a combination of CPU and RAM. These hybrid proclets are also migratable, but resource proclets are easier for Quicksand to schedule because they only require a substantial amount of one resource.

To simplify the use of resource proclets, Quicksand offers developers familiar high-level programming abstractions, freeing them to focus on application logic without thinking about decomposing applications, maintaining granularity, or manually managing resources. Applications link with Quicksand's libraries that provide high-level abstractions: data structures, batch-parallel computations, and building blocks for stateless and stateful services.




```
void preprocessing_stage(qs::ShardedVector<Img>& inputs,
                         qs::ShardedQueue<Img>& outputs) {
 auto f = [outputs](Img i) { outputs.push(preprocess(i)); };
 auto scaling_policy = qs::batch::MinimizeQueue(outputs);
 qs::batch::ForAll(inputs, f, scaling_policy);
}
```

Figure 3: An example of an ML training pipeline illustrating the decomposition of the application into resource proclets using Quicksand's high-level abstractions. It highlights the preprocessing stage, where distributed workers process input images read (with prefetching) from a sharded vector and output to a sharded queue. Quicksand's batch computing framework dynamically scales the number of workers based on the output queue size to match the training stage's consumption rate.

Internally, these libraries provide their functionality by orchestrating resource proclets through a common auto-sharding layer, which dynamically splits and merges resource proclets to keep them granular. This layer is built on resource proclets to decouple state and compute, and leverages Quicksand's runtime to promptly respond to changes in resource demand and availability. Crucially, all of this resource management happens transparently to the application developer, who interacts only with the high-level abstractions.

4 Design

We now discuss how developers create Quicksand applications using its high-level abstractions (§4.1); describe the libraries in Quicksand that support these abstractions (§4.2); explain Quicksand's auto-sharding layer, which Quicksand libraries use to map high-level constructs into resource proclets (§4.3), and finally explain resource proclets in depth (§4.4).

4.1 Quicksand Applications

Application developers write a Quicksand program without worrying about how it is decomposed into resource proclets or how resources are managed. Quicksand handles resource allocation, scaling, and load balancing transparently, freeing developers from manual resource management. It facilitates this by offering three high-level abstractions with familiar APIs: sharded data structures, batch computing, and stateless and stateful services. These abstractions allow developers to focus on application logic while Quicksand's runtime efficiently manages the underlying resources.

The sharded data structures provide in-memory distributed data storage, while exposing APIs akin to C++ STL containers. For example, application developers use `emplace_back` to insert data into a sharded vector and use iterators to traverse the vector. The batch computing abstration enables distributed computing over sharded data structures, exposing declarative API functions such as `Map`, `ForAll`, `Reduce`, etc. The service abstraction offers wrappers for building elastic stateless and stateful services. Developers register a handler function for stateless services, or a class that implements application-specific stateful logic for stateful ones.

To showcase Quicksand's high-level abstractions, let's examine an ML training pipeline that runs on shared datacenter GPUs, which often experience changing availability [28, 57, 61]. As shown in Figure 3, the pipeline has three stages: loading, which loads input images from storage into memory for training; preprocessing, which preprocesses the loaded images; and training, where GPUs train a model on the preprocessed images.

Since input images may consume significant memory, the loading stage utilizes Quicksand's `ShardedVector` to efficiently distribute images across machines using sharding, where each shard is associated with a memory proclet. During preprocessing, the pipeline leverages Quicksand's batch computing abstraction, deploying compute proclets across machines to preprocess the images in parallel. Quicksand might schedule compute and memory proclets into different machines to combine their available resources, and it employs transparent prefetching of the vector to efficiently access remote memory. The pipeline then pushes the preprocessing outputs into a sharded queue for consumption by GPUs. This design might increase network traffic between stages, but it allows Quicksand to decouple compute and memory resources, resulting in better resource utilization and scalability.

If too few compute proclets are provisioned, the GPU can be starved for data, resulting in underutilization of expensive GPU resources. Conversely, provisioning too many compute proclets wastes CPU resources and increases memory usage, as the queue will grow if images are preprocessed faster than the GPU can consume them. To strike this balance, Quicksand dynamically scales the number of compute proclets in the preprocessing stage based on the queue length, aiming to keep the queue shallow while ensuring image production matches the GPU's consumption rate (details in §4.2).

Implementing the core logic of the preprocessing stage requires only a few lines of code (Figure 3). This code has no mention of resource proclets or sharding, the developer does not need to think about resource decomposition.

4.2 Quicksand Libraries

Quicksand offers high-level abstractions via a collection of libraries that applications link to. Each library is dedicated to a specific abstraction, mapping high-level constructs into granular resource proclets through its default policies and mechanisms for splitting and merging. The auto-sharding layer (§4.3) initiates the actual splitting and merging, calling upon these policies and mechanisms accordingly. These libraries abstract away the complexities of resource proclet management, allowing developers to focus on application logic rather than resource allocation and decomposition.

```
void emplace_back(T value)
T& at(size_t index)
size_t size()
```
```
void seal()
  - Seals the vector, making it immutable and iterate-able.
void unseal()
  - Unseals the vector, allowing modifications.
const_iterator cbegin(), cend()
  - Provides read-only iteration with prefetching.
```

Figure 4: Quicksand's sharded data structure API (e.g., a subset of the `ShardedVector` shown above) mirrors C++ STL interface, with additional methods to support efficient iteration with prefetching.

```
void ForAll(ShardedDS& input, F f, Policy p)
  - Applies f to all items of a sharded data structure in parallel.
```
```
ShardedDS Map(ShardedDS& input, F f, Policy p)
  - Transforms input items with f in parallel.
```
```
T Reduce(ShardedDS& input, F f, T init, Policy p)
  - Reduces input items with f and init in parallel.
```

Figure 5: A subset of Quicksand's batch computing API for parallel computation over sharded data structures, with support for custom scaling policies via an optional argument.

The sharded data structure library offers general-purpose data structures, including all common C++ STL container types, with an interface similar to the C++ STL. For instance, Figure 4 shows a subset of the `ShardedVector` API.

Leveraging Quicksand's auto-sharding layer, the library distributes data structures' contents across memory proclets based on a data structure-specific sharding key. For instance, the `ShardedVector` shards over vector indices, such that each memory proclet holds data for a contiguous range; a `ShardedMap` shards into ranges of the map's key space, with each memory proclet managing key-value pairs for a designated key range. The library supports iterators with prefetching for efficient traversal.

The memory proclets of a data structure enforce lower and upper bounds on their heap sizes. Upon element insertion, the auto-sharding layer checks if the shard becomes too large (or too small on element removal). In such cases, the library splits the memory proclet or merges it with a neighboring one.

The mechanisms to split and merge are data structure-specific. For example, the `ShardedQueue` in Figure 3 only supports insertion at the back (`push`), so only the last memory proclet can grow. Consequently, a split always occurs at the tail proclet, generating a new, empty memory proclet for subsequent insertions. By contrast, a sharded map is partitioned by key range and any of its proclets can grow. When a proclet goes above its size threshold, it splits its key range in half and selectively moves key-value pairs to the new proclet.

The batch computing library provides common parallel batch computing operators (*e.g.,* `ForAll`, `Map`, `Reduce`) over input and output stored in sharded data structures, see Figure 5. The computations are internally performed by compute proclets that split and merge via the auto-sharding layer to dynamically adjust parallelism based on available resources or user-specified policies.

```
StatelessServ Register(F handler)
  - Registers a stateless service.
StatefulServ<ID> Register<ID>(F handler, State& s)
  - Registers a stateful service with a client ID type and state.
void Invoke(StatelessServ& s, Args args)
  - Invokes a stateless service with arguments.
void Invoke(StatefulServ<ID>& s, ID id, Args args)
  - Invokes a stateful service with a client ID and arguments.
```

Figure 6: Quicksand's service library API for registering and invoking stateless and stateful services.

Recall the ML training example in Figure 3. The program uses the `ForAll` function to parallelize image preprocessing. Initially, it creates a single compute proclet, responsible for processing the entire range of the input `ShardedVector` by applying `f` to each element. When there are idle CPU cores in the cluster, the library splits the input range of the compute proclet, creating a new compute proclet to handle the other half. By iteratively splitting compute proclets, the library can utilize all available CPUs in the cluster. The batch computing library supports custom scaling policies, such as the `MinimizeQueue` policy in Figure 3. Every two milliseconds, it checks the queue length: if below a small threshold, it adds compute proclets. Otherwise, it adjusts the number of compute proclets by a proportional factor of the queue's shrinking rate, aligning production with consumption speed.

The completion time of compute proclets may vary, due to data skew or slow machines. Upon the completion of a fast compute proclet, the auto-sharding layer detects an idle core and reactively subdivides an ongoing proclet. This process effectively automates the mitigation of stragglers.

The service library facilitates the creation of elastic low-latency services, whether stateless or stateful, capable of dynamically responding to fluctuations in load. Figure 6 shows its API for application developers.

The stateless service uses compute proclets through the auto-sharding layer. Each invocation request from clients implicitly has a random 64-bit number, which Quicksand internally uses as the sharding key in the auto-sharding layer to evenly distribute the requests among compute proclets.

The stateful service uses pairs of compute and memory proclets managed by the auto-sharding layer. Each client request explicitly comes with a per-client or per-session identifier, which Quicksand uses as the sharding key to implement sticky routing through the auto-sharding layer.

Internally, both stateless and stateful services parallelize client interactions to scale out by dispatching requests to different compute proclets based on sharding keys.

Both service types adopt the same policy: Quicksand monitors the CPU utilization of each compute proclet at two-millisecond intervals. If the average usage exceeds one core, the library splits it into two, enabling services to scale up in response to load increase. Conversely, a service winding down from a load spike may have compute proclets that underutilize CPU resources; the library reactively merges proclets, adapting a service's resources to its current load. This two-

```
class VectorShard<T> : std::vector<T> {
  using Key = size_t; using Val = T;
  std::pair<Key, VectorShard<T>> Split() { /*...*/ }
  void Merge(VectorShard<T>& other) { /*...*/ }
};
using MyShardedVector<T> = qs::ds::ShardedDS<VectorShard<T>>;
```

Figure 7: A user can implement their own sharded vector by creating a wrapper (`VectorShard`) around `std::vector`, specifying the types of the sharding key and value, and implementing the split/merge functionality for the vector data.

millisecond interval strikes a balance between reactivity and stability. A shorter interval could lead to frequent, unnecessary adjustments due to transient load spikes, while a longer interval might delay responses to real workload changes. When splitting and merging a stateless service's proclets, the library splits or combines the key ranges they are responsible for. For a stateful service, the application developer provides `Split` and `Merge` functions over the service's internal state stored in proclets.

4.2.1 Extensibility

While Quicksand's libraries cover common use cases for cloud applications, users (*i.e.,* developers) can extend them for additional functionality. For instance, they could augment the sharded data structure library with new data structures, or the batch computing library with new operators (*e.g.,* flatmap, joins, or streaming-based transformations). The service library likely needs no extension as its stateless and stateful APIs cover all intended use cases, with applications supplying the business logic.

The interface of Quicksand libraries facilitates this extensibility. For example, a user can implement their own version of sharded vector in a few lines of code, as shown in Figure 7. The user wraps `std::vector`, adding type definitions for the sharding key and value, and provides implementations for `Split` and `Merge`, which split a vector shard's elements and combine two vectors' elements, respectively. Then, the user defines a new, application-facing type (`MyShardedVector`) by aliasing to the `ShardedDS` class provided by Quicksand's auto-sharding layer, parameterized with the shard implementation (`VectorShard`).

4.3 The Auto-Sharding Layer

Quicksand's libraries use the auto-sharding layer to shard high-level programming constructs into resource proclets and maintain their granularity. The auto-sharding layer accomplishes three tasks: (1) it keeps track of how sharding keys correspond to shards encapsulated within resource proclets, (2) it directs the execution of a given function to the appropriate shard, and (3) it splits or merges shards based on a user-supplied policy using the shard-specific split or merge functionality.

The auto-sharding layer is implemented by Quicksand's `AutoSharder` class, which requires a sharding key type, a proclet type to be sharded, and two policy functions `ShouldSplit` and `ShouldMerge`. These functions assess whether a shard's current state calls for splitting or merging. In turn, `AutoSharder` provides a `Route` method to direct a function invocation to the right shard, given a sharding key, a hint, and the function lambda. The hint indicates whether the shard might need merging or splitting.

Internally, `AutoSharder` keeps a map from sharding key ranges to resource proclets. This map is centrally stored in a mapping proclet, but each `AutoSharder` instance caches it to reduce load. The `Route` method checks the local map and dispatches the lambda to the appropriate shard. If the local cache is stale, the shard rejects the dispatch, in which case `Route` consults the central map and retries. After executing the lambda, the `Route` method uses the hint in combination with `ShouldSplit` or `ShouldMerge` to possibly split or merge the proclet shard using the shard-specific functions.

As an example, the sharded vector in Figure 7 can be built with `AutoSharder` using the vector index as the sharding key and memory proclet as the proclet type. The `ShouldSplit` and `ShouldMerge` functions check that the shard's heap size falls within a desired range. The sharded vector's `push_back` method routes execution of `std::vector`'s `push_back` to the last shard, while the `at` method routes execution to the appropriate shard based on the index key.

4.4 Resource Proclets

Quicksand defines three resource proclet types. The intent is that each resource proclet type primarily consumes one type of resource (compute, memory, etc). The actual definitions of resource proclets are based on their functionality: a compute proclet executes computations through a lambda API, while a memory proclet stores state through a read/write API. §7.4.1 shows that these abstractions in practice indeed meet the intent of using a single resource: compute proclets have a CPU-to-memory ratio greater than 0.1 core per MiB, so they mainly consume CPU; and memory proclets have a ratio of less than 0.001 core per MiB, so they mainly consume memory. Quicksand's runtime monitors the resource usage of resource proclets, issuing warnings if usage exceeds the thresholds. Quicksand also defines a hybrid proclet that provides both compute and memory functionality to support colocation of compute and memory when necessary for performance; it has a more balanced CPU-to-memory ratio (between 0.001–0.1 core per MiB). Appendix §A.2 shows the full API.

4.4.1 Compute Proclet

Compute proclets are conceptually akin to computing threads; developers use them to implement an application's computational logic. A compute proclet still consumes some memory to hold thread stacks and a small heap for temporary data. To create a compute proclet, Quicksand provides two functions: `MakeCompProc(Fn)` simply executes a given function, and `MakeCompProc(Range, Fn)` runs the function over a specified range of data. The range variant is useful for splitting (and merging similarly): upon splitting a compute proclet, the range is split into half, and the new compute proclet takes over the other half. Like popular serverless platforms, the compute

proclet API enables developers to execute computations without thinking about server infrastructure or locality. Unlike serverless functions, compute proclets can split, merge, and migrate during their execution, enabling Quicksand applications to utilize idle CPUs and react to compute pressure.

Compute proclets support both ephemeral and long-running computations. A compute proclet can create more compute proclets and wait for their completion. However, a compute proclet cannot directly access another compute proclet's memory. Instead, compute proclets communicate with memory proclets (via the network or the local data path) to share computation results, which we discuss next.

4.4.2 Memory Proclet

Memory proclets are conceptually akin to small heap regions (a few MBs); developers use them to keep application data. A memory proclet may consume a small amount of CPU to read and write data. Quicksand offers the `MakeMemProclet` function to create a memory proclet encapsulating a user-defined shard. This function returns a smart pointer to the memory proclet that exposes the `Read` and `Write` functions. Quicksand's libraries use these functions (and their asynchronous counterparts) to operate on the memory proclet's contents. As memory proclets can be remote, `Read` returns a copy of the indexed element.

Quicksand supports *sealing* memory proclets to make them read-only. Sealed memory proclets allow for efficient runtime prefetching, as there is no risk of reading stale data even with concurrent accesses. When a memory proclet is sealed and code invokes `Iterate` on it, the Quicksand runtime leverages the semantic information provided by the high-level abstractions to predict access patterns. It then prefetches subsequent elements in the background, effectively hiding access latency for remote data. Finally, memory proclet supports splitting and merging by invoking the `Split` and `Merge` functions of the encapsulating shard.

4.4.3 Hybrid Proclet

Hybrid proclets colocate a computing thread with heap data, providing compute-data locality for applications with low compute intensity. However, a hybrid proclet offers the least placement flexibility, as it may consume both nontrivial amounts of CPU and memory resources. Consequently, Quicksand discourages the use of hybrid proclets via constrained APIs. A hybrid proclet cannot be created directly; it can be realized only by promoting an existing memory proclet using an explicit `Hybridize` call. Once hybridized, it provides the `Offload` method to execute a C++ lambda within the hybrid proclet (computation shipping). A hybrid proclet can be demoted to a memory proclet by calling `Dehybridize`.

5 Implementation

Our Quicksand prototype has 10k lines of C++ code, which implement resource proclets atop Nu [49].

We co-designed the Quicksand runtime with resource proclets, leveraging information from the thread scheduler and slab memory allocator to precisely monitor per-proclet resource usage, with minimal overhead (<1%). Each machine's runtime periodically reports resource usage and idle resources to Quicksand's centralized controller, facilitating centralized decision-making on proclet placement and migration.

To safely and efficiently support resource proclet splitting and merging, we employ RCU locks [36]. Proclet access operations acquire the RCU reader lock, while splitting/merging events acquire the writer lock. This design minimizes the impact on application performance by shifting most synchronization overheads to infrequent splitting and merging events.

Initiation of splitting and merging, managed by the auto-sharding layer (§4.3), involves updating the sharding mapping stored in the centralized mapping proclet. Stale clients efficiently discover updates by fetching the latest mapping from an append-only log at the centralized proclet. This log sequentially records all changes made to the mapping.

We optimized our data structure library (§4.2) to minimize network overhead. For instance, our sharded data structures offer batching, grouping mutations (*e.g.,* `push_back` operations in `ShardedVector`) locally before flushing to memory proclets using large network packets. This significantly improves the performance of small-object operations. In addition, the library supports concurrent flushing, adapting parallelism to align with the application's production rate. Similarly, it supports concurrent prefetching for iterating over data structures.

6 Application Case Studies

We now describe our experience implementing and porting four applications for Quicksand. They cover batch and latency-critical applications, a range of compute intensities, memory usage, and resource proclet usage (Figure 8). Thanks to Quicksand's high-level abstractions, all applications required moderate programming effort, in terms of lines of code and development time; porting existing applications (social network and video encoding) involved modifying 3%–5% of the code and building new applications (ML training and sorting) required adding about 20 Quicksand-related lines of code. Figure 3 illustrates the decomposition of the ML training application, while Figure 9 presents the others. Sorting is the only application that uses hybrid proclets.

ML training pipeline. We implemented the compute-intensive batch application in Figure 3 as an OpenCV-based pipeline with a data loading stage, a CPU-based preprocessing stage, and a GPU-based training stage. We use images from the ImageNet ILSVRC2012 dataset [50] as our training data (35.3 GiB). Our implementation has 300 LoC, of which 20 LoC relate to Quicksand. We leverage Quicksand's `ShardedVector` to store input images, `ShardedQueue` for preprocessed images, and the batch computing library for parallel preprocessing. The batch computing abstraction dynamically scales workers based on the output queue size, matching the training stage's consumption rate. Since our testbed lacks GPUs, we simulate the training stage by consuming images at a rate of 0.2 milliseconds per image, based on the reported

Apps	Type	Dataset	Comp. Intensity	Mem. (GiB)	Abstractions Used	# resource proclets: comp.+mem.+hybrid	QS-related LoC/ Total LoC	Development Time (days)
ML Training	Batch	ILSVRC 2012 [50]	High	35.3	`Sharded{Queue, Vector}` `BatchComputing`	158 + 1,340 + 0	21 / 307	1
Social Network	Latency-critical	soc-delicious [13]	Low	33.4	`ShardedMap` `StatelessService`	91 + 19,034 + 0	98 / 1800	2
Sorting	Batch	TritonSort [46]	Low	112.0	`ShardedPartitions` `BatchComputing`	182 + 4,108 + 4,108	22 / 98	1
Video Encoding	Batch	bunny_480p [9]	High	17.5	`ShardedVector` `BatchComputing`	168 + 625 + 0	203 / 6083	4

Figure 8: Our four case study applications leverage (different) high-level abstractions provided by Quicksand, resulting in moderate porting and development efforts. Their memory usage and the number of resource proclets vary over time, with peak values reported.

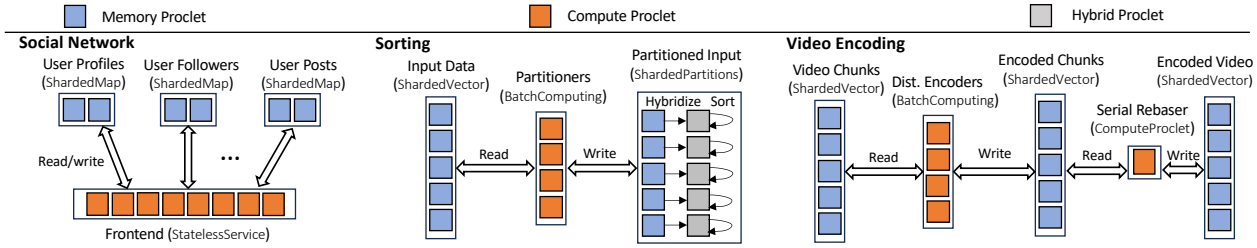


Figure 9: Quicksand's high-level abtractions automatically decompose applications into resource proclets and maintain their granular sizes.

rate of image classification models [14, 44].

Social Network. We ported the SocialNetwork application from DeathStarBench [17]. This is a latency-critical, Twitter-like web service, originally built using twelve microservices. It has a low compute intensity and uses 33.4 GiB memory. The original implementation employs an NGINX web frontend for processing client requests, which accesses user data stored in Redis. This architecture naturally separates CPU-intensive (frontend) and memory-intensive (data store) components, aligning well with Quicksand's model of resource decoupling. Consequently, it is straightforward to port this application to Quicksand, and gain benefits such as resource fungibility and elasticity that are not present in the original implementation.

Our port structures the application into a stateless front-end service layer and several stateful key-value stores, using Quicksand's `StatelessService` and `ShardedMap` abstractions. The stateless service wraps all the business logic originally handled by the microservices, while the sharded map stores user data originally handled by Redis. Our port has 1.8k LoC compared to the 6.8k LoC in the original application. This reduction is primarily due to the elimination of boilerplate code required for interacting with frameworks like gRPC, Redis, etc., which were used in the original implementation. In our port, 98 LoC are Quicksand-related, while the remaining code is application logic. By contrast, an already-granular implementation with Nu [49] (2.2k LoC) required developers to manually implement substantial functionality (such as decomposition, shard resolution, etc.) in the application code. Quicksand provides this functionality out-of-the-box and adds extra functionality (such as load-adaptive dynamic sharding) despite reducing the code size by 20%.

Sorting. We built an in-memory distributed sorting tool that sorts 400 million 100B key-value pairs, similar to TritonSort [46]. This is a batch application with low compute intensity and uses 112 GiB of memory. Our implementation has 307 LoC, of which 21 LoC are Quicksand-related. As depicted in Figure 9, our implementation uses Quicksand's batch computing library and custom data structures. It has two stages: *(i)* shuffling input data into key-based ranges; *(ii)* parallel sorting of these ranges. The first stage uses the batch computing library to concurrently read input data from a `ShardedVector` (with prefetching) and insert them into a `ShardedPartitions` data structure. This data structure uses the input as the sharding key and maintains an unordered vector of key-value pairs within each shard. Upon splitting, it uses the quickselect algorithm [25] to evenly partition these pairs into two ranges. After shuffling, sorting each partition (*i.e.,* the data of each shard) produces the final result. Given the low compute intensity of sorting, we improve locality by promoting memory proclets into hybrid proclets for fast in-place sorting, rather than sorting using compute proclets.

Video encoding. We ported the distributed video encoder from ExCamera [16], a batch application with high compute intensity designed for serverless platforms like AWS Lambda [6]. ExCamera has two stages: 1) a parallel encoding stage that encodes all video chunks, and 2) a sequential rebasing stage that stitches all encoded chunks together in sequence. Since ExCamera's code is no longer maintained or runnable, we created a faithful baseline implementation. Our port, as shown in Figure 9, utilizes Quicksand's `ShardedVector` and batch computing abstractions. It initially loads video chunks from the `bunny_480p` dataset (with a memory footprint of 17.5 GiB) into the input `ShardedVector`. It then uses the batch computing library to read chunks in parallel from the input vector, encodes them, and writes the outputs into another `ShardedVector` for encoded chunks. Finally, a single

compute proclet rebases all chunks from the encoded vector into the final encoded video. The port changes 203 LoC of the application's total 6k LoC.

7 Evaluation

Our evaluation focuses on answering following questions:

1. Does Quicksand unstrand resources better than existing disaggregation and granular programming systems (§7.1)?

2. How does Quicksand respond to changes in resource availability (§7.2) and resource demand (§7.3)?

3. Does the resource proclet abstraction effectively separate the use of different resources (§7.4.1)?

4. Does Quicksand's rapid scaling and fine granularity improve utilization and performance (§7.4.2 and §7.4.3)?

Setup. Our testbed has eight machines, each equipped with an Intel Xeon E5-2680 v4 CPU and 64 GB memory, connected by a 100 GbE network, running Linux kernel v5.10. One machine hosts Quicksand's cluster controller. For the ML pipeline experiments, we use one machine as the GPU server and the remaining six machines as the CPU servers (for preprocessing). For the social network experiments, one machine serves as the load generator and the remaining six machines as servers; and for the sorting and video encoding experiments, all seven machines run the application.

7.1 Using Stranded Resources

We first investigate how Quicksand performs on machines with stranded resources (*e.g.,* servers with ample free CPU but little memory or vice-versa). We use the ML training pipeline, which has high compute intensity, and the social network application, whose compute intensity is low. To comprehensively evaluate Quicksand's ability to handle various cases of stranded resources, we divide the same amount of total resources between machines in different ways (Figure 10). In the *CPU-unbalanced* setup, three machines have few available CPU cores, but memory is equally distributed; in the *memory-unbalanced* setup, three machines have little memory, but available cores are equally distributed; in the *both-unbalanced* setup, three machines have few CPU cores and plenty of memory, while others have plenty of available CPU cores, but little available memory.

We compare Quicksand to two existing systems: Hermit [45], a state-of-the-art memory disaggregation framework, and Nu [49], a recent granular programming framework. Hermit is not designed to scale past the CPU resources of one machine (§2.2). To overcome this limitation, we give Hermit an advantage by running one Hermit instance per machine and evenly distributing tasks among them. This requires extra programming effort that counteracts Hermit's transparency promise. We compare the performance of all systems on the unbalanced setups to an ideal, balanced setup, where the aggregate resources are the same but distributed over only three machines, each with ample CPU and memory. Since Hermit is designed for memory disaggregation, we also evaluate it on a *memory-disaggregated* setup, which partitions the same amount of total resources into three CPU servers with minimal memory and three pure memory servers.

	Machine 1-3		Machine 4-6	
	# CPUs	RAM (GiB)	# CPUs	RAM (GiB)
Ideal	26	12	N/A	N/A
CPU-unbalanced	6	6	20	6
Mem-unbalanced	13	1	13	11
Both-unbalanced	6	11	20	1
Mem-disaggregated	26	1	0	11

Figure 10: To create resource stranding, we partition resources (78 CPUs, 36 GiB memory) in different ways: CPU-unbalanced, memory-unbalanced, and both-unbalanced. We also consider a memory-disaggregated setup and an ideal balanced setup.

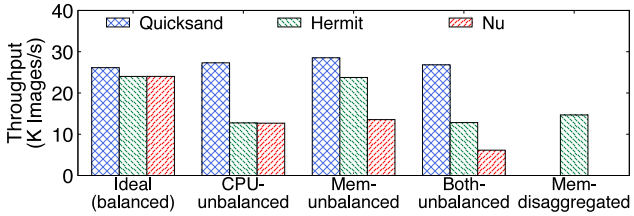


Figure 11: The ML pipeline on Quicksand combines resources from different machines, achieving throughput similar to the ideal scenario. Quicksand has 1.2–2× higher throughput than Hermit in unbalanced setups, as Hermit cannot use stranded CPU resources and suffers from kernel paging overheads when combining memory resources; and Quicksand's throughput exceeds Nu's by 2–4× in unbalanced setups as Nu always bundles CPU and memory resources.

A good result for Quicksand would show performance similar to Hermit and Nu on the ideal (balanced) setup, and better performance than these systems in the unbalanced setups. With high compute intensity, a strong result would show Quicksand coming close to the ideal setup's performance; while when compute intensity is low, resource proclet network's overheads dominate, and a good result would show limited performance loss compared to the ideal setup.

Figure 11 shows the results for the compute-intensive ML pipeline. In the ideal setup, each machine uses all 26 cores and ≈11 GiB memory, achieving throughput of 26k images/sec. In unbalanced setups, Quicksand achieves similar throughput despite the resource imbalance[1], thanks to its effective semantic-based data prefetching mechanism. Hermit and Nu achieve 92% of the throughput of Quicksand in the ideal setup—the overhead arises because they spawn a large number of compute threads, causing lower cache locality and extra context switches. By contrast, Quicksand splits and merges compute proclets to adapt to the available CPU resources. In the mem-unbalanced setup, Hermit combines stranded memory resources across machines through paging, but paging is expensive and limits Hermit to 83% of Quicksand's throughput. Hermit cannot combine stranded CPU resources, as its modified baseline evenly distributes tasks; Quicksand therefore achieves 2× higher throughput in the CPU-unbalanced

[1] In fact, Quicksand's throughput in unbalanced setups is up to 10% *above* the ideal setup; this is an artifact of lighter last-level cache contention.

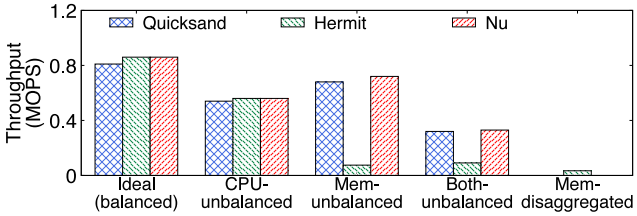


Figure 12: The social network application's low compute intensity poses a challenge in using stranded resources. Yet, Quicksand attains 40%–84% of the ideal throughput in severely imbalanced setups, outperforming Hermit, which achieves only 4%—11% due to poor locality from paging. Nu matches Quicksand's performance as it allows developers to control memory placement to optimize locality.

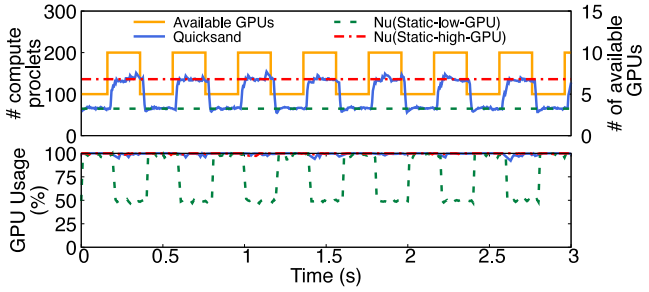
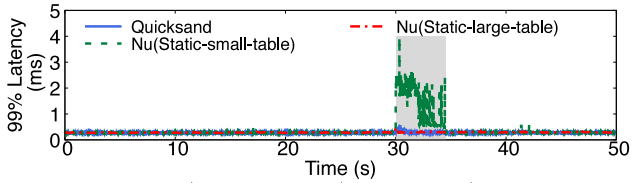


Figure 13: Quicksand quickly adapts the ML pipeline's preprocessing rate to the changing GPU availability by scaling the number of compute proclets. Compared to two Nu-based baselines that statically provision CPU resources, Quicksand achieves either 25% higher GPU utilization or uses 25% fewer CPU cores.

and both-unbalanced setups. Nu cannot combine stranded CPU and memory, as its proclets bundle different types of resources; Quicksand therefore outperforms Nu by 2–4×.

Figure 12 shows the results of the social network application with low compute intensity, where it is challenging to hide the overhead of accessing data on remote machines. In the ideal setup, each machine uses all 26 cores and ≈11 GiB memory, and all three systems achieve similar throughput (810k ops/sec). Quicksand's performance is slightly lower than the baselines as its auto-sharding layer adds a ≈6% overhead. All systems also achieve similar throughput in the CPU-unbalanced setup because the application has a low compute intensity. When there is memory imbalance (memory-unbalanced and both-unbalanced), it is crucial to optimize data locality to achieve good performance. Hermit's transparent paging prevents programmer control over memory placement and results in poor locality: Hermit achieves 9–11% of the ideal throughput in unbalanced setups (4–8× worse than Quicksand), and merely 4% in the memory-disaggregated setup. By contrast, Nu allows programmers to control placement to optimize locality and achieves 40–84% of the ideal throughput, similar to Quicksand.

These results demonstrate that Quicksand successfully uses stranded resources, and matches or outperforms state-of-the-art disaggregation and granular computing frameworks.



	Number of shards	Avg. shard size (MiB)	Avg. mem. usage (GiB)
Quicksand	1,086–19,034	1.7	17.25
Static-small-table	1,053	14.0	14.75
Static-large-table	19,928	1.4	27.5

Figure 14: In the social network application, Quicksand dynamically splits and merges memory procets of the hash table, ensuring granularity. This allows Quicksand to mitigate memory pressure (gray region) with rapid proclet migration, resulting in stable low tail latency. The Nu-based *static-small-table* baseline statically provisions a small hash table, resulting in larger shards that are slower to migrate, thereby experiencing disruptions. The *static-large-table* baseline, by contrast, statically overprovisions a large hash table, achieving low latency but using 37% more memory than Quicksand.

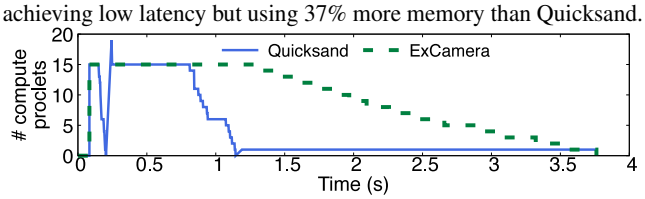


Figure 15: In the video encoding application, Quicksand adapts to the changing parallelism of the application by reducing the number of compute proclets, saving 60% CPU compared to ExCamera.

7.2 Adapting to Changing Resource Availability

We now study how well Quicksand reacts to changes in the availability of scarce resources.

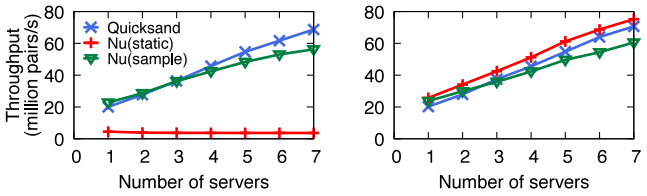
7.2.1 Reacting to Changing GPU Availability

In this experiment, we vary the number of GPUs available for the training stage of the ML pipeline between five and ten every 200ms, mimicking the burstiness reported by Park et al. [44]. We measure the amount of CPU cores allocated by Quicksand for the pre-processing stage. We compare Quicksand to two baselines that statically provision CPUs, implemented using Nu: *static-high-GPU* provisions CPUs for the peak number of available GPUs (ten), while *static-low-GPU* provisions for the minimum number (five). A good result for Quicksand would demonstrate it saturating available GPUs, comparable to *static-high-GPU*, without wasting CPUs.

Figure 13 shows that Quicksand rapidly scales compute proclets by splitting or merging them to match changing GPU availability. As a result, Quicksand achieves GPU saturation, while *static-low-GPU* achieves only 75% average GPU utilization. Quicksand consumes an average of 105 cores, but *static-high-GPU* consumes 140 cores. This shows that Quicksand effectively adapts to rapid changes in GPU availability.

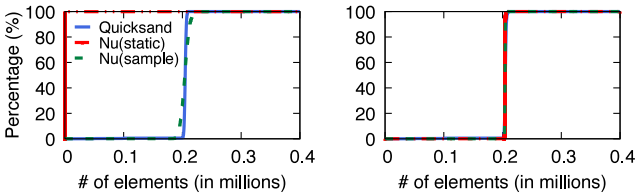
7.2.2 Reacting to Memory Pressure

We now investigate if Quicksand can quickly react to memory pressure to avoid performance disruption by splitting, merging, and migrating resource proclets across machines.



(a) Normal distribution. **(b)** Uniform distribution.

Figure 16: Sorting normally-distributed data, Quicksand achieves higher throughput than the *static* baseline because it avoid stragglers through dynamic splitting. For both normally and uniformly distributed data, Quicksand achieves higher throughput than the *sample* baseline because it avoids the extra sampling phase.



(a) Normal distribution. **(b)** Uniform distribution.

Figure 17: The CDF of shard sizes in the sort benchmark. Quicksand balances shard sizes well by dynamically splitting resource proclets. The *sample* baseline also balances shard sizes but requires an extra phase. The *static* baseline experiences serious load imbalance with a normal distribution of input due to its static partitioning.

We run the latency-critical social network application on all available machines with constant load (1.5 MOPS), and launch a memory antagonist that keeps allocating new pages at one machine, creating memory pressure. The memory usage in the social network application mainly comes from its sharded hash table, so the experiment focuses on this data structure. We compare Quicksand to two baseline systems that statically provision the number of shards, implemented using Nu: *static-small-table* provisions for a small hash table with a few shards; while *static-large-table* baseline overprovisions for a large hash table with many shards. A good result for Quicksand would show it achieving constant, low latency despite memory pressure and consuming less memory than the *static-large-table* baseline.

Figure 14 shows the results. Between 0–30 seconds, there is no memory pressure, so all setups achieve low and stable 99th percentile latency (≈0.3 ms). The memory antagonist starts at t=30 seconds and all setups react by migrating shards during the gray region. Both Quicksand and *static-large-table* experience only a small latency increase as they can quickly migrate their small (<2 MiB) shards. However, Quicksand uses 37% less average memory than *static-large-table*, as it rightsizes the number of shards through dynamic splitting and merging. *Static-small-table*, by contrast, ends up with a few large shards (≈14 MiB) that are slow to migrate, and therefore experiences latency spikes. After resolving memory pressure through migration, the latency returns to normal. These results show that Quicksand effectively mitigates memory pressure while keeping good tail latency and memory usage.

7.3 Adapting to Changing Resource Demand

We now study if Quicksand adapts to changing resource demand caused by phased program behavior or load imbalance.

7.3.1 Phased Application Behavior

The video encoding application has multiple concurrent stages that encode chunks, followed by a serial stage that stitches together encoded chunks. We run the ExCamera[6,16] encoding scheme[2] and compare our performance to the baseline. A good result for Quicksand would show its resource usage matching the application's available parallelism.

Figure 15 shows the results. The concurrent encoding stages in the beginning (0–0.8s) have high parallelism, and both ExCamera and Quicksand use 15 CPUs. Quicksand's gap at 0.2s occurs because our prototype performs synchronization between two consecutive parallel stages; when Quicksand scales back up, it briefly overshoots the right level of parallelism and scales back down to 15 compute proclets. After 0.8s, the parallelism gradually decreases as encoding tasks complete. In Quicksand, compute proclets write their outputs to a sharded vector built from memory proclets; once they finish encoding, they terminate. Quicksand starts the serial stitching stage with a single core at 1.2s, which combines data from the sharded vector. By contrast, ExCamera's serverless implementation couples compute and state; the encoding lambdas must remain alive until the stitching lamdba consumes their output, leading to many allocated but idle CPUs between 1.2–3.7s. Even though Quicksand and ExCamera complete the job at the same time, Quicksand saves 60% CPU resources, represented by the area between the two lines.

This result demonstrates that Quicksand can quickly adapt to phased application behavior to improve resource utilization without impacting performance.

7.3.2 Load Imbalance

We study Quicksand's ability to adapt to load imbalance, which causes poor utilization and stragglers [10]. We use the sorting application with an input that follows either a skewed normal distribution or a uniform distribution without skew. We compare Quicksand with two baselines implemented using Nu. The *sample* baseline handles the skewed input by running an initial phase that samples 0.5% of the input to learn its distribution and choose appropriate pivots to partition the data across machines. The *static* baseline ignores skew and partitions by uniformly spaced pivots. Quicksand avoids the need for sampling, as its auto-sharding dynamically splits memory proclets when they become too large. We measure the sorting throughput and shard size distribution as we scale the number of machines. A good result for Quicksand would show that it adapts to skew and achieves balanced partitions without the cost of an additional sampling phase.

Figure 16a shows the sorting throughput for the normal

[2] In ExCamera's notation, this means encoding chunks of 6 frames in parallel, and then combining them in strings of 16 chunks [16].

Apps	Compute proclet			Memory proclet		
	CPU (cores)	Mem (MB)	Ratio (core/MB)	CPU (cores)	Mem (MB)	Ratio (core/MB)
ML training	1.0	0.7	1.43	0.0038	31	0.00012
Social network	0.58	5.4	0.11	0.00044	1.3	0.00034
Sorting	0.34	1.3	0.26	0.022	32	0.00068
Video encoding	0.75	4.1	0.18	0.0016	12	0.00013

Figure 18: The CPU and memory usage for compute and memory proclets, averaged over time and across all proclets within each category, along with their CPU-to-memory ratios. Hybrid proclets, used only in the sorting application, have an average use of 0.28 cores and 32 MB memory, resulting in a CPU/memory ratio of 0.0087.

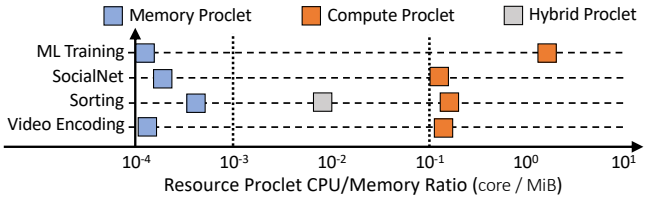


Figure 19: Different types of resource proclets vary significantly in CPU/memory ratio (*x*-axis, in log-scale). Compute proclets range from 0.1 to 10 in ratio, hybrid proclets range from 0.001 to 0.1, and memory proclets range from 0.0001 to 0.001.

distribution. *Static* performs and scales poorly due to stragglers caused by load imbalance, while *sample* and Quicksand perform and scale well. Figure 17a explains the load imbalance of *static*: a few partitions have most of the data, while others have little data. *Sample* and Quicksand have balanced partitions of 200K elements and thus balance load well across machines. Quicksand performs better than *sample* as it avoids the need for an extra sampling phase, saving 1.4 seconds.

Figures 16b and 17b show the results for a uniform distribution. Here, *static* is optimal because it partitions the input well without sampling work. Quicksand is slightly slower because splitting resource proclets incurs overhead. *Sample* is the slowest due to the extra sampling phase.

These results show that Quicksand efficiently adapts to load imbalance by splitting resource proclets dynamically.

7.4 Drill-Down Experiments

7.4.1 The Effectiveness of Resource Proclets

We now investigate the effectiveness of resource proclets in separating the usage of different types of resources. Figure 18 presents the resource consumption for each application's resource proclets. For each type of resource proclet, it shows the CPU and memory usage (averaged over time and across all proclets within that category) along with the CPU/memory ratio (*i.e.,* average CPU usage to average memory usage).

Different types of resource proclets exhibit drastically different CPU/memory ratios, as shown in Figure 19. Compute proclets primarily utilize CPU resources with small memory usage, exhibiting CPU/memory ratios between 0.1 and 10; this indicates a significant bias towards CPU in its resource usage. With compute proclets, Quicksand is able to unstrand 1 CPU core in exchange for consuming 2 MiB of memory.

Memory proclets, by constrast, predominately utilize memory resources with low CPU consumption, exhibiting CPU/memory ratio ranging from 0.0001 to 0.001—four to five orders of magnitude lower than compute proclets; this indicates a significant bias towards memory in its resource usage. With memory proclets, Quicksand is able to unstrand 1 GiB memory with an average CPU consumption of 0.3 cores.

Hybrid proclets fall between memory proclets and compute proclets, with CPU/memory ratio ranging from 0.001 to 0.1.

This shows that compute proclets and memory proclets effectively separate the use of CPU and memory resources.

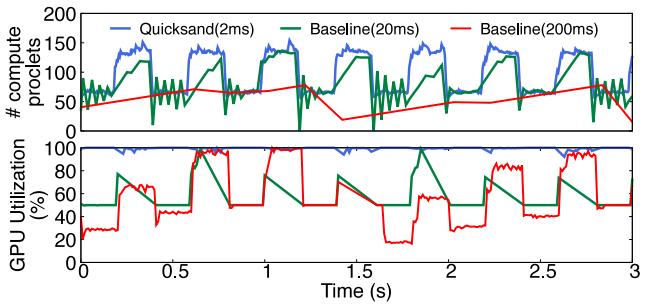


Figure 20: Quicksand's rapid scaling (2 ms) is crucial for saturating GPUs in the ML training pipeline. Slowing down reaction time to 20 ms or 200 ms causes GPU utilization to drop as it takes too long to provision enough CPUs to keep the GPUs busy.

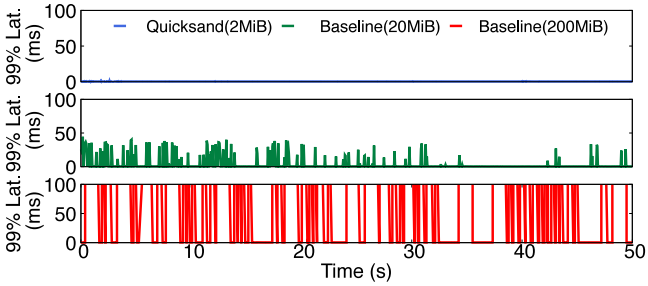


Figure 21: Fine granularity of memory proclets in Quicksand (2 MiB) is crucial for low tail latency in the social network application. Larger memory proclets (20 MiB or 200 MiB) causes intermittent latency disruption due to the impact of splitting larger proclets.

7.4.2 Benefit of Rapid Scaling

We now investigate the importance of Quicksand's rapid scaling to react to changes in resource availability. We use the ML pipeline and the same setup as in §7.2.1, but slow down Quicksand's reaction time from 2 ms to 20 ms or 200 ms to measure the impact of fast scaling on utilization.

Figure 20 shows the result. Unlike the original Quicksand, baselines with slower reaction time fail to saturate GPUs as they scale the number of compute proclets too slowly to match the changing GPU availability. This shows Quicksand's rapid scaling is crucial for harnessing changing resources.

7.4.3 Benefit of Fine Granularity

Finally, we examine the necessity to keep resource proclets fine-grained. We run the social network application on all available machines with constant load (1.5 MOPS). For latency-critical applications, by default Quicksand splits memory proclets above a 2 MiB threshold. Larger proclets impact tail latency in two ways: first, splitting them takes longer, causing a longer stall; second, larger proclets imply fewer proclets, thus stalling a proclet impacts a larger proportion of requests. To quantify the impact, we compare the original Quicksand to modified versions that use 20 MiB and 200 MiB as the split threshold.

Figure 21 shows the result: original Quicksand has a stable 99th percentile latency, but coarser-grained proclets suffer latency disruptions during splitting. These results show that Quicksand's fine granularity is crucial for tail latency.

8 Discussion

8.1 Applicability and Limitations of Quicksand

Quicksand is ideal for cloud applications that benefit from flexible resource allocation and fine-grained scaling. However, some traditional single-machine applications, like high-performance computing simulations that depend on low-latency memory access, may be difficult to adapt without performance loss. Similarly, applications optimized for specific hardware, such as those requiring direct access to accelerators or GPUs, may face challenges. Legacy systems with monolithic architectures may also need significant refactoring to fully utilize Quicksand.

8.2 Relation between Quicksand and Disaggregation

Unlike recent hardware-based resource disaggregation systems [5, 26, 34, 51, 55, 59], which rely on specialized interconnects like CXL [11] or GenZ [18], Quicksand enables practical resource disaggregation on commodity hardware through its software-based approach. By decomposing applications into independent, granular resource proclets, Quicksand disaggregates compute and memory, unlike pure memory disaggregation systems [2, 22, 45]. Quicksand's abstractions simultaneously give library developers more control over data locality than transparent memory disaggregation. While hardware-based solutions may ultimately offer lower latency, Quicksand represents a practical intermediate step that delivers many disaggregation benefits on today's commercial datacenter hardware.

9 Related Work

Data Analytics Frameworks. Distributed frameworks like Spark [58], MapReduce [12], and Dryad [27] offer high-level APIs (e.g., DataFrames) to automate task decomposition for data analytics, but they are ill-suited for other domains like sharded services. While providing some resource management, they operate at coarser granularities and timescales, coupling resources and lacking the flexibility to respond dynamically to resource fluctuations. As a result, they cannot leverage stranded resources as effectively as Quicksand, which uses fine-grained proclets and millisecond-scale adaptability.

Autoscaling Frameworks. Slicer [1] and Shard Manager [33] help developers add sharding to applications, and add or remove shards in response to load. However, they target long-running, large-scale systems with big shards and minute-scale reaction times. Instead, Quicksand targets millisecond-scale reaction times to unstrand short-lived resources.

Resource Decoupling. Resource decoupling, as demonstrated by Monotasks [42, 43], involves breaking down jobs into single-resource tasks, akin to resource proclets. However, Monotasks primarily aim to facilitate performance analysis of Spark jobs, which leads to distinct design choices and implications: Monotasks cannot combine stranded resources across machine and operates at a coarser time granularity.

VM Consolidation. VM consolidation is a classic method for enhancing resource utilization by packing VMs onto physical machines [24, 30, 54]. However, it is inefficient due to resource overprovisioning and fragmentation [49]. StopGap [41] introduces VM splitting and fusion to improve consolidation, but it is restricted to stateless services (relying on reconfiguring the load-balancer to support splitting and fusion), and second-scale reaction times (as VMs are heavyweight).

Resource Harvesting. Resource harvesting aims to improve resource utilization by using idle resources that may be available only intermittently, using evictable VMs [3, 7, 20] and subsequent improvements [4, 47, 52, 56, 60]. This idea targets secondary applications that execute opportunistically but need not run if the resources are unavailable or disappear. Quicksand targets primary applications and uses more fine-grained execution abstractions.

10 Conclusion

We presented Quicksand, a new system to unstrand datacenter resources. Quicksand introduces resource proclets, which decouple the consumption of resources of different types, automatically regulate resource granularity, and support fast migration. Quicksand offers familar high-level abstractions to programmers while hiding the complexity of resource decomposition into resource proclets in extensible libraries.

Our results show that Quicksand successfully unstrands resources other systems cannot utilize, while also promptly responding to changes in resource availability and demand. Quicksand is available as open-source software [48].

Acknowledgements

We thank our shepherd Kay Ousterhout; the anonymous reviewers; and members of the PDOS group at MIT and the ETOS group at Brown University for their helpful feedback. We appreciate CloudLab [15] for providing the experiment platform. This work was funded in part by a Facebook Research Award; Google Faculty Awards; the DARPA FastNICs program under contract #HR0011-20-C-0089; the NSF under awards CNS-2104398 and CNS-2045170; VMware; and by ACE, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. "Slicer: Auto-Sharding for Datacenter Applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. "Can Far Memory Improve Job Throughput?" In: *European Conference on Computer Systems (EuroSys)*. 2020.

[3] *Amazon EC2 Spot Instances*. URL: https://aws.amazon.com/ec2/spot (visited on 08/20/2024).

[4] Pradeep Ambati, Iñigo Goiri, Felipe Vieira Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[5] Christoph Anneser, Lukas Vogel, Ferdinand Gruber, Maximilian Bandle, and Jana Giceva. "Programming Fully Disaggregated Systems". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2023.

[6] *AWS Lambda*. URL: https://aws.amazon.com/lambda/ (visited on 08/20/2024).

[7] *Azure Spot VM*. URL: https://azure.microsoft.com/en-us/services/virtual-machines/spot (visited on 08/20/2024).

[8] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. "Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms". In: *IEEE Micro* (2023).

[9] *Big Buck Bunny 480P Video*. URL: https://media.xiph.org/video/derf/y4m/big_buck_bunny_480p24.y4m.xz (visited on 08/20/2024).

[10] Emilio Coppa and Irene Finocchi. "On Data Skewness, Stragglers, and MapReduce Progress Indicators". In: *ACM Symposium on Cloud Computing (SoCC)*. 2015.

[11] *CXL Consortium*. URL: https://www.computeexpresslink.org (visited on 08/20/2024).

[12] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2004.

[13] *Delicious online social network dataset*. URL: https://networkrepository.com/soc_delicious.php (visited on 08/20/2024).

[14] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. "IOS: Inter-Operator Scheduler for CNN Acceleration". In: *CoRR* (2020). arXiv: 2011.01302.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. "The Design and Operation of CloudLab". In: *USENIX Annual Technical Conference (ATC)*. 2019.

[16] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing using Thousands of Tiny Threads". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[17] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[18] *Gen-Z Consortium*. URL: https://en.wikipedia.org/wiki/Gen-Z_(consortium) (visited on 08/20/2024).

[19] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. "Towards Modern Development of Cloud Applications". In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2023.

[20] *Google Cloud Preemptible VM Instances*. URL: https://cloud.google.com/compute/docs/instances/preemptible (visited on 08/20/2024).

[21] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. "The cost of a cloud: research problems in data center networks". In: *ACM SIGCOMM Computer Communication Review* (2009).

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with Infiniswap". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2017.

[23] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. "Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces". In: *IEEE/ACM International Symposium on Quality of Service (IWQoS)*. 2019.

[24] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

[25] C. A. R. Hoare. "Algorithm 65: Find". In: *Communications of the ACM* (1961).

[26] *Intel Rack Scale Design Architecture*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf (visited on 08/20/2024).

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: European Conference on Computer Systems (EuroSys). 2007.

[28] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *USENIX Annual Technical Conference (ATC)*. 2019.

[29] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *CoRR* (2019). arXiv: 1902.03383.

[30] *Kubernetes*. URL: https://kubernetes.io (visited on 09/20/2022).

[31] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. "Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines". In: *Proceedings of Machine Learning and Systems* (2022).

[32] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. "Software-Defined Far Memory in Warehouse-Scale Computers". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2019.

[33] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. "Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2021.

[34] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2023.

[35] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. "Eigen: End-to-End Resource Optimization for Large-Scale Databases on the Cloud". In: *Proceedings of the VLDB Endowment (PVLDB)* (2023).

[36] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. "Read-copy update". In: *AUUG Conference Proceedings*. AUUG, Inc. 2001.

[37] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. "Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2022.

[38] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. "Analyzing and Mitigating Data Stalls in DNN Training". In: *Proceedings of the VLDB Endowment (PVLDB)* (2021).

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[40] Yuanjiang Ni, Pankaj Mehra, Ethan Miller, and Heiner Litz. “TMC: Near-Optimal Resource Allocation for Tiered-Memory Systems”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2023.

[41] Vlad Nitu, Boris Teabe, Leon Fopa, Alain Tchana, and Daniel Hagimont. “StopGap: elastic VMs to enhance server consolidation”. In: *Proceedings of the Symposium on Applied Computing*. 2017.

[42] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. “Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks”. In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[43] Kay Ousterhout, Christopher Canel, Max Wolffe, Sylvia Ratnasamy, and Scott Shenker. “Performance Clarity as a First-Class Design Principle”. In: *Workshop on Hot Topics in Operating Systems (HotOS)*. 2017.

[44] Seo Jin Park, Joshua Fried, Sunghyun Kim, Mohammad Alizadeh, and Adam Belay. “Efficient Strong Scaling Through Burst Parallel Training”. In: *Proceedings of Machine Learning and Systems*. Edited by D. Marculescu, Y. Chi, and C. Wu. 2022.

[45] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. “Hermit: Low-Latency, High-Throughput, and Transparent Remote Memory via Feedback-Directed Asynchrony”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2023.

[46] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. “TritonSort: A Balanced and Energy-Efficient Large-Scale Sorting System”. In: *ACM Transactions on Computer Systems (TOCS)* (2013).

[47] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi A. Noghabi, and Jian Huang. “BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2022.

[48] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. *Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing*. URL: https://github.com/NSDI25-Quicksand/Quicksand (visited on 10/15/2024).

[49] Zhenyuan Ruan, Seo Jin Park, Marcos Aguilera, Adam Belay, and Malte Schwarzkopf. “Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2023.

[50] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* (2015).

[51] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. “LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[52] Supreeth Shastri and David Irwin. “HotSpot: automated server hopping in cloud spot markets”. In: *ACM Symposium on Cloud Computing (SoCC)*. 2017.

[53] M. Taubenblatt and A. Tantawi. “Quantifying Composable Data Center Utilization”. In: *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022.

[54] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. “Borg: The next Generation”. In: *European Conference on Computer Systems (EuroSys)*. 2020.

[55] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Wei Wang, and Qinfen Hao. “CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers”. In: *IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2023.

[56] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. “SmartHarvest: Harvesting idle CPUs safely and efficiently in the cloud”. In: *European Conference on Computer Systems (EuroSys)*. 2021.

[57] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. “Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent”. In: *USENIX Annual Technical Conference (ATC)*. 2023.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.

[59] Yiying Zhang. “Make It Real: An End-to-End Implementation of A Physically Disaggregated Data Center”. In: *ACM SIGOPS Operating Systems Review* (2023).

[60] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. “History-based harvesting of spare cycles and storage in large-scale datacenters”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[61] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. “HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

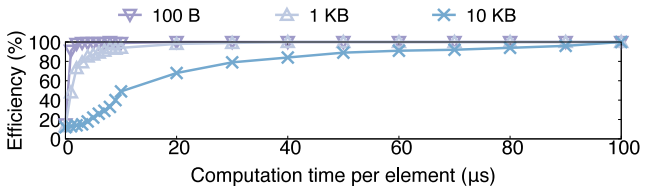


Figure 22: Quicksand's efficiency (y-axis) when compute proclets operate on memory proclets at different servers, as a function of compute intensity, normalized to the performance of local computation in a single machine with the same total resources. Quicksand's efficiency increases with compute intensity, requiring 3 μs of computation per 100B element to reach 99% efficiency.

A Appendix

A.1 Impact of Application Compute Intensity

We study the efficiency of Quicksand when its resource proclets separate computation from data across servers, *i.e.,* when a compute proclet operates on remote memory proclet. While prefetching can mitigate some remote access overhead, it introduces its own costs and becomes less effective as compute intensity decreases. Intuitively, the more CPU cycles spent per memory proclet access, the better Quicksand should amortize the overheads of remote access. We thus conduct a sensitivity analysis of performance under different compute intensities.

Our microbenchmark uses 26 compute proclets that iterate through remote data in separate sharded vectors and compute on it. The total memory is ≈15 GiB. We run this microbenchmark on two imbalanced servers (20 cores with 1 GiB memory, and 6 cores with 15 GiB memory). We quantify efficiency with respect to a baseline that computes on local memory with the same total resources (26 cores, 16 GiB memory). We consider vector elements of size 100 B, 1 KB, and 10 KB; and vary the amount of computation per element from 0 μs to 100 μs. A good result would show Quicksand achieving near-ideal efficiency at a broad range of compute intensities.

Figure 22 shows the results. When the element size is 100 B, Quicksand requires 3 μs of computation per element to reach 99% efficiency. For larger elements, higher compute intensity is required to amortize the communication cost: it requires 30 μs and 100 μs for 1 KB and 10 KB elements respectively to reach 99% efficiency.

These results suggest that Quicksand can decouple compute and memory efficiently at moderate compute intensities (microseconds of computation per remote access).

A.2 Resource Proclet API

Type	API	Description
Compute Proclet	`CompProc<Fn> MakeCompProc(Fn)`	Makes a compute proclet that runs `Fn`.
	`CompProc<RangedFn> MakeCompProc(Range, Fn)`	Makes a compute proclet that runs `Fn` over a `Range` of data.
	`CompProc<RangedFn> Split()`	Splits the current compute proclet.
	`void Merge(CompProc<RangedFn>&)`	Merges a compute proclet.
Memory Proclet	`MemProc<Shard> MakeMemProc<Shard>(Args)`	Makes a memory proclet that hosts a `Shard` object
	`Shard::Val Read(Fn)`	Reads data via running `Fn`.
	`void Write(Fn, Shard::Val)`	Writes data via running `Fn`.
	`void Seal()`	Makes the memory proclet read-only; enables prefetching.
	`void Unseal()`	Makes the read-only memory proclet writeable; disables prefetching.
	`Iterable<Shard> Iterator()`	Returns iterator that iterates over shard contents in an impl-defined order.
	`HybridProc<Shard> Hybridize()`	Promotes the memory proclet into a hybrid proclet.
	`MemProc<Shard> Split()`	Splits the current memory proclet.
	`void Merge(MemProc<Shard>&)`	Merges a memory proclet.
Hybrid Proclet	`std::invoke_result_t<Fn> Offload(Fn)`	Ships the computation `Fn` to the hybrid proclet.
	`MemProc<Shard> Dehybridize()`	Demotes the hybrid proclet back into a memory proclet.
	`HybridProc<Shard> Split()`	Splits the current hybrid proclet.
	`void Merge(HybridProc<Shard>&)`	Merges a hybrid proclet.

Figure 23: Quicksand's compute proclet API enables the execution of functions (optionally over a specified data range), while the memory proclet API enables reads, writes, and iterations. When sealed, memory proclets permit safe prefetching. A memory proclet can be promoted into a hybrid proclet, allowing for computation offloading. Compute proclets with ranges, memory proclets, and hybrid proclets support splitting and merging.

Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs

Quanxi Li [1,2] Hong Huang [1,2] Ying Liu [1,2] Yanwen Xia [1,2] Jie Zhang [3] Mosong Zhou [4]
Xiaobing Feng [1,2] Huimin Cui [1,2*] Quan Chen [5] Yizhou Shan [4] Chenxi Wang [1,2*]

[1]*SKLP, ICT, CAS* [2]*University of Chinese Academy of Sciences*
[3]*Peking University* [4]*Huawei Cloud* [5]*Shanghai Jiao Tong University*

Abstract

The Microsecond (µs)-scale I/O fabrics raise a tension between the programming productivity and performance, especially in disaggregated memory systems. The multithreaded synchronous programming model is popular in developing memory-disaggregated applications due to its intuitive program logic. However, our key insight is that although thread switching can effectively mitigate µs-scale latency, it leads to poor data locality and non-trivial scheduling overhead, leaving significant opportunities to improve the performance further. This paper proposes a memory-disaggregated framework, *Beehive*, which improves the remote access throughput by exploiting the asynchrony within each thread. To improve the programming usability, Beehive allows the programmers to develop applications in the conventional multithreaded synchronous model and automatically transforms the code into *pararoutine* (a newly proposed computation and scheduling unit) based asynchronous code via the Rust compiler. Beehive outperforms the state-of-the-art memory-disaggregated frameworks, *i.e.*, Fastswap, Hermit, and AIFM, by 4.26×, 3.05×, and 1.58× on average.

1 Introduction

Memory disaggregation allows applications to utilize the available memory of other servers and acquire orders of magnitude higher memory resources to avoid performance degradation due to insufficient memory, such as swapping data to disk. However, accessing remote memory exhibits microsecond (µs)-scale access latency, which incurs non-trivial performance penalties over accessing local memory [15, 60, 88, 108, 109]. Hence, a new challenge arises in the memory disaggregated data center—how to mitigate the µs-scale latency of remote access.

State of the art. The conventional wisdom of mitigating data accessing overhead is to avoid CPU idle time through the multithreaded synchronous programming model [15, 56, 92, 105]. Taking the memory disaggregated platform, AIFM [92], as an example: when accessing remote data, the thread will be suspended after issuing a request and then yield the core to another ready thread immediately, as shown in Figure 1(a). To further improve CPU efficiency, a series of kernel-bypass techniques have been proposed to alleviate the thread context switching overhead, such as Shenango [82], AIFM [92], and ZygOS [87]. The user-level thread (referred to as *uthread*) context switching consumes only around 200 cycles for context saving and restoration [82], 10× faster than kernel thread (*i.e.*, POSIX thread, referred to as *kthread*), which helps quickly fill the idle CPU by another ready task during the µs-scale remote access.

However, we observed some overlooked performance penalties for applications with an increasing number of uthreads. Intuitively, the µs-scale remote access latency, *e.g.*, reaching up to 6 µs [88, 92], easily outweighs the uthread switching overhead. Hence, launching a large number of uthreads to saturate the CPU resources and achieve high remote memory access concurrency is necessary and profitable. However, a series of consequences caused by thread switching leads to serious performance loss.

Taking Multi-Grid PDE [65] as an example (§ 3). When running on the state-of-the-art remote memory data plane, AIFM [92], the application's performance is still throttled by two main reasons—first, frequent uthread switching worsens data locality, leading to much higher CPU cycles spent on local memory access due to significantly enlarged LLC misses. Second, the scheduling overhead takes up to 23% of the CPU time, which becomes worse as the number of uthread increases or as the local memory size shrinks.

In other words, hiding remote access latency via threading can stealthily consume non-trivial CPU resources due to poorer data locality and heavier scheduling, which shifts the bottleneck from remote memory access to local memory and uthread managing. Although tuning the number of uthreads can alleviate the aforementioned inefficiency, the sweet spot of each application can be different (§ 5.4), which usually incurs unpredictable tuning and profiling overhead.

Major insights. The main question is, *can the application generate enough concurrent remote memory accesses to mitigate the µs-scale latency and also provide good data locality to avoid excessive cache misses?*

On the one hand, improving multithreaded program's data locality has been extensively studied for decades [36, 58,

* Corresponding authors: Chenxi Wang and Huimin Cui.

72, 94]. A series of locality-aware memory allocators have been proposed and integrated into the existing systems. For example, they segregate the objects of different threads via the thread local allocation buffer (TLAB) to avoid false sharing [19, 40, 59] and aggregate the adjacent allocated objects for a better cache/page locality [58]. Therefore, letting a core execute a thread without preemption can largely maintain good data locality due to substantial accessing data in the TLAB.

On the other hand, we find a mass of opportunities to parallelize the code of big data applications. The code of a thread can be split into a series of tiny code blocks (*e.g.*, coroutines [5, 8]) without data dependency, which enables asynchronous execution within a thread —when encountering remote data access, the subsequent coroutines can continue execution without blocking on waiting for the completion of previous remote access, as shown in Figure 1(b). Compared to thread switching, the routine-based asynchronous execution has much smaller working set size (WSS) and context switching overhead, which makes it a good fit for achieving high concurrency.

Beehive. Based on the aforementioned observation, we propose Beehive, an asynchronous framework designed for memory-disaggregated servers. Instead of relying on thread (uthread or kthread) switching to hide the µs-scale latency, Beehive lets each application thread **run to completion** [16] to maintain well-organized data locality and exploit high CPU efficiency and remote access concurrency via coroutine-based **asynchronous execution model** [9, 30, 84, 86]. To eliminate the well-known programming obstacles of asynchronous programming [15], such as *stack ripping* [9] and *callback hell* [56], Beehive allows developers to write programs in the conventional synchronous model with provided APIs and automatically transform the programs to the asynchronous model via the Rust compiler (§ 4).

Beehive's design targets two main challenges:

First, *how to transform synchronous code to asynchronous model automatically?* For code transformation, the compiler needs to split the code into a series of coroutines and then schedule the generated routines based on the program dependency analysis. However, pointer aliasing remains the major obstacle for data dependency analysis. To tackle this challenge, Beehive is implemented in Rust to benefit from its *ownership* types that provide *uniqueness* and *immutability* constraints during programming, which helps manage the pointer aliasing issues [41, 76] (§ 4.3).

Second, *how can we efficiently mitigate µs-scale remote memory access latency via coroutines?* A large number of concurrent remote accesses is necessary. Hence, first, Beehive implements a lightweight execution and scheduling unit, *pararoutine*, based on the Rust coroutine [8] to support high concurrency. The switching (yielding the core and scheduling another task to run) of a pararoutine is 2.3× faster than Shenango uthreads due to the stackless design and a cache-locality-optimized allocator that needs only five instructions to allocate a routine frame. Second, pararoutines employ cooperative scheduling for batch processing applications to contain the scheduling overhead while providing preemption for the latency-critical applications to keep the low tail latency (§ 4.4). Note that Beehive also supports thread switching when it cannot generate enough pararoutines to fill idle CPUs.

In summary, Beehive outperforms state-of-the-art solutions, *e.g.*, AIFM [92] and Hermit [88], through better data locality, more lightweight switching, and similar concurrency.

Results. We have evaluated Beehive with eight workloads that cover a wide spectrum of memory access patterns. Experimental results demonstrate that Beehive outperforms the state-of-the-art kernel paging-based frameworks, Fastswap [11], Hermit [88], and a runtime-based framework, AIFM [92], by, on average, 4.26×, 3.05× and 1.58×, correspondingly. In addition, Beehive also reduces the tail latency by, on average, 502× and 13.2×, compared to Hermit and AIFM. Beehive is available at https://github.com/ICTPLSys/Beehive.

2 Background

This section presents existing disaggregated memory systems and discusses the remote memory programming models.

2.1 Runtime-based Disaggregated Memory

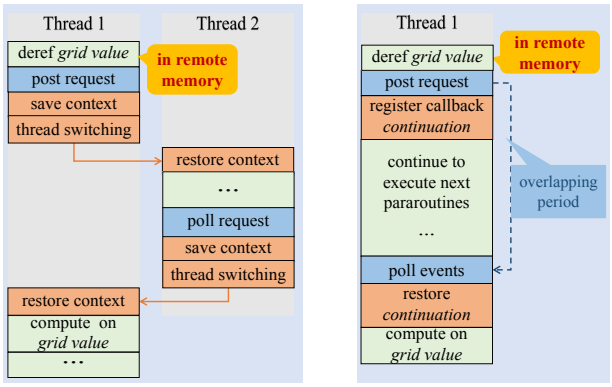
Disaggregated memory systems are generally categorized into three types: runtime-based [21, 24, 47, 66, 92, 106, 107], kernel paging-based [43, 88, 95, 108, 112], and hardware-based [22, 42, 45, 60, 61]. Although the paging-based solutions are transparent, they usually incur severe I/O amplification issues and lengthy remote access latency due to managing memory via the kernel paging [13, 21, 92]. On the other hand, considering that there is no commodity hardware-based solution at this moment, we implement Beehive according to the design of AIFM [92], a runtime-based framework that is built on the Shenango uthreads [82]. There are two major reasons: First, the data is managed in fine-grained object granularity, *e.g.*, *remoteable pointer* [92], to alleviate the I/O amplification and also utilize the program semantics to improve the prefetching effectiveness. Second, kernel-bypassing techniques [70, 87] help alleviate the thread switching overhead to achieve much higher remote access concurrency.

2.2 Programming Model

Compared to hardware mechanisms, *e.g.*, out-of-order execution and non-blocking caches [64], software solutions are more suitable for hiding the µs-scale latency due to their ability to generate high remote memory access concurrency and flexible data access granularity. Remote data management can be programmed in two different ways: *synchronous model* [15, 48, 56, 92, 105] and *asynchronous model* [9, 30, 84, 86].

Synchronous Programming. When developing a synchronous program, developers think linearly about the program's control flow —the thread will block on the I/O call site until the remote access completes, and there is no need to worry about accessing incomplete data. The remote access concurrency can be achieved via threading, which schedules another ready thread to preempt the CPU to alleviate the I/O stall (Figure 1(a)), shifting the programming burden to the thread library [105]. Most existing runtime-based solutions use the synchronous model for programming productivity and robustness [15, 24, 92].

Asynchronous programming. On the contrary, the control flow of asynchronous programs is much more flexible. As shown in Figure 1(b), after issuing the remote data access, the thread can skip the related code that may access the data being fetched and continue executing the subsequent code, hiding the remote access latency. Compared to synchronous programs, asynchronous programs can achieve similar concurrency with lower context switching overhead and better data locality due to sticking to the thread's local data. However, the developers need to be aware of the program data dependency and manually register the skipped code as a *continuation* that needs to be processed after the pending remote access is finished, as in the FaRM [34] (a transaction-based asynchronous programming model for distributed shared memory), which is prone to error and has become one of the main challenges of adopting asynchronous programming [9, 56].



(a) Synchronous execution (b) Asynchronous execution

Figure 1: The (a) synchronous and (b) asynchronous execution flow of traversing grid points in Multi-Grid PDE [65].

2.3 Program Dependency Analytics

Accurate and thorough program dependency analytics is the foundation of transforming the synchronous application into the asynchronous model. Pointer aliasing remains the main obstacle to program dependency analysis [31, 38, 46, 98]. However, *ownership* types are effective in eliminating the alias issues in large-scale programs [25, 26, 69, 80] and have been widely adopted by system languages, such as Rust [69], which gives us an opportunity to build the Beehive compiler for automatic code transformation.

Pointer aliasing means that a memory location is accessible by multiple symbolic names, such as pointers and references. The alias may escape the compiler's static analysis. As a result, the data can be modified through some of the aliases without the compiler being aware of the aliasing. Hence, the compiler needs to conservatively treat the potential alias when converting the code into concurrent routines to avoid violating execution orders, such as read-after-write (RAW). As a result, plenty of asynchronous code transformation opportunities (*i.e.*, concurrency) are wasted in large-scale programs that have complicated pointer usage and data dependency.

Ownership. When programming with the Rust *ownership* types, the references have the *uniqueness* and *immutability* properties, which help control aliasing [69, 80]. Uniqueness ensures that an object can only have one active mutable reference at a time. Another reference can *borrow* the mutable reference from the owner for a period of time (within a scope), during which time the owner cannot modify the data. As a result, the compiler can statically capture the transition of data's unique mutable reference and locate all the data dependencies, *i.e.*, true dependency (read-after-write, RAW), anti-dependency (write-after-read, WAR), and output dependency (write-after-write, WAW) [80], that can be used for concurrent routine generation. Note that multiple immutable references (read-only) can be created from the owner for reading, which is safe for the concurrent routine generation. Refer to § 4.3 for more details.

3 Motivation

This section demonstrates the performance bottleneck in today's remote memory systems, which shifts from remote memory access to local memory access and thread management, and the potential benefits of exploiting the asynchrony within a uthread.

We use Multi-Grid PDE (referred to as MGP, § 5.1), a memory-intensive scientific partial differential equation solver from NAS Parallel Benchmarks (NPB) [65], as the example to demonstrate the performance breakdown. MGP performs V-cycle multi-grid method to compute a 3D scalar Poisson equation, which exhibits good spatial locality due to its iterative stencil computation on fine-grained and coarse-grained grids. We evaluate MGP on two kinds of far-memory data planes, the paging-based Hermit [88] and the runtime-based AIFM [92]. We execute the original MGP on Hermit and re-implement the MGP on AIFM by using its uthread and Shenango runtime [82]. When running on Hermit, MGP is configured to one kthread per core, while the number of application uthreads is tuned to optimal, *i.e.*, 64 uthread per core, when running on AIFM (§ 5.4). Note that we configured the same number of uthreads for its prefetching and evacuation uthread pool as AIFM does [92]. We allocate 8 cores and 34 GB memory in total, among which 13% memory is

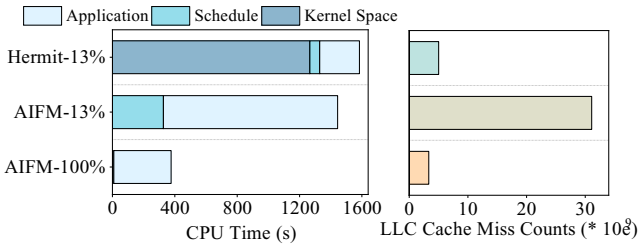


Figure 2: Performance breakdown of MGP when running on Hermit and AIFM. Although AIFM can reduce remote access overhead, it incurs non-trivial thread scheduling overhead and results in degraded application performance due to excessive LLC misses.

local and the rest is remote. The local and remote servers are connected via a pair of 100 Gbps Mellanox ConnectX-5 InfiniBand adapters.

As shown in Figure 2, Hermit and AIFM achieve similar performance, *i.e.*, AIFM slightly outperforms Hermit by 10%, while they are throttled by different reasons—(1) when running on Hermit, application throughput is throttled by the kernel paging that is used for remote access due to the page table management [12, 13], swap entry allocation [108], and other kernel space overhead [112]. Note that Hermit is the SOTA far-memory data plane and has already improved the paging efficiency for disaggregated memory; (2) AIFM is designed to address the kernel space overhead and I/O amplification [21] (which is negligible in MGP due to the good spatial locality). AIFM reduced 64% of CPU cycles spent on remote access (on-demand fetching, prefetching, and evacuation) by avoiding trapping into the kernel and efficient thread switching. However, except for the runtime bookkeeping overhead [24], we observed non-trivial thread scheduling overhead and significantly degraded application throughput due to LLC misses, which becomes the new bottleneck.

Thread switching leads to poor data locality. MGP organizes the data as two arrays, one for the fine-grained grid and another for the coarse-grained grid. The two grids are sliced into chunks and evenly assigned to different uthreads for load-balancing and high parallelism. Each uthread iterates and process the slices alternating between the fine and coarse grids. As shown in Figure 2, AIFM leads to 6.5× more LLC misses than running on Hermit due to frequent uthread switching. There are two root causes of why the multithreaded synchronous model has poor data locality.

On the one hand, the application data needs to be partitioned across application uthreads to reduce data racing, resulting in a separate data hotspot for each application uthread. At the same time, AIFM relies on frequent uthread switching to fill the CPU idle time during the µs-scale synchronous remote accesses, which urges for a large number of uthreads and high switching frequency, resulting in a large working set size (WSS). For example, MGP requires 64 application uthreads to achieve optimal performance (§ 5.4), causing the application uthread switching up to 4×10^8 times during the execution.

On the other hand, the runtime-based solution, *i.e.*, AIFM, is designed to manage data at fine granularity to exploit application semantics and reduce the I/O amplification [92], which further increases the demand for uthread-level concurrency. For example, AIFM maintains two separate uthread pools for object prefetching and evacuation, 17 for prefetching, and 40 for evacuation under the configuration of 8 cores [92], further increasing the number of uthread switchings.

Thread scheduling overhead. Another major performance penalty of the AIFM-based MGP comes from the thread scheduling overhead, which consumes 23% of total CPU cycles in this case. Although, the uthread context switching consumes only around 200 cycles [82], the cost of scheduling is non-trivial. During each uthread switching, the uthread needs to be paused and inserted into a waiting queue for the next step of scheduling, the cost of which can be 10× higher than the context switching. The uthread switching frequency in AIFM is high, reaching up to 2.2 million times per second. Hence, it is beneficial to continue reduce the thread context switching and scheduling overhead.

On the other hand, the large number of concurrent uthreads lead to heavy synchronization overhead, both within the uthread group and across different uthread groups. As aforementioned, MGP utilizes up to 121 uthreads under 8 cores (64 for application, 17 for prefetching and 40 for evacuation). Taking the synchronization between the application and the evacuation uthread groups as an example, the evacuator is used to concurrently compact the live objects for defragmentation. To avoid data race, a fence (*dereference scope*, § 4.4) is used to synchronize the application uthreads and the evacuation uthreads—the evacuation uthreads needs to be held until all the executing application uthreads exit their dereference scopes at least once [24, 92]. Hence, more live uthreads can potentially lead to more synchronization overhead.

Conclusions. Mitigating µs-scale latency through frequent uthread switching is the root cause of the poor data locality and scheduling overhead, not the number of uthreads. Hence, exploiting asynchrony within a thread provides opportunities to maintain a better data locality while achieving even similar (even higher) remote access concurrency.

4 Design and Implementation

4.1 Overview

Beehive is an asynchronous framework built for applications running on the memory-disaggregated platform. Beehive is developed based on the Rust language to benefit from the *Ownership* type system and contains three layers, programming interfaces (§ 4.2), a Rust compiler (§ 4.3) and the associated runtime (§ 4.4), as demonstrated in Figure 3.

For the asynchronous execution, programmers only need to develop the application in a conventional multithreaded syn-

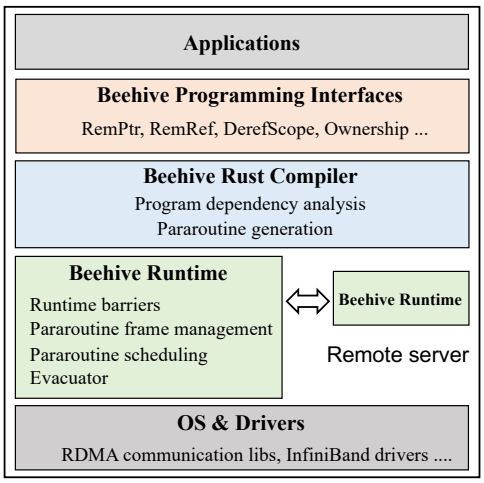


Figure 3: Design overview of the Beehive.

```
// remoteable pointer
struct RemPtr<T> { metadata: u64 /* see Figure 4 */ }
impl<T> RemPtr<T> {
    fn deref(&self) -> RemRef<T>;
    fn deref_mut(&mut self) -> RemRefMut<T>;
    // allocate remoteable object,
    // the position can be local, remote or offloaded
    fn allocate(&mut self, value: T, position: Pos);
    /* ... */
}
// remoteable references
struct RemRef<'a, T>; // immutable
impl<T> Deref for RemRef<'_, T>;
struct RemRefMut<'a, T>; // mutable
impl<T> DerefMut for RemRefMut<'_, T>;
```

Listing 1: Remote data management interfaces

chronous model via the Beehive interfaces, and the provided Rust compiler transforms the code of a thread into a series of *pararoutines* (§ 4.4.1) that can execute concurrently. Beehive compiler adds runtime barriers to support data location checks and movement between the local and remote memory. When encountering remote access, the compiler wraps the rest of the pararoutine code snippet as a *continuation* and inserted into a waiting queue after issuing the remote access request. The thread will continue to execute the subsequent code (or pararoutines) continuously to process the thread's local data.

4.2 Programming interfaces

Listing 1 demonstrates the Beehive APIs for manipulating the remote data. There are two types of interfaces: (1) the *remoteable pointer* (`RemPtr`) that is derived from the Rust `Box<T>` type is used for allocating objects in the remoteable space (Figure 7); and (2) the *remoteable references* (`RemRef` and `RemRefMut`) that is derived from the Rust *reference* type is used for (mutable and immutable) borrowing and accessing the remote objects.

Remoteable pointer. When allocating data via the remoteable pointer (`RemPtr`), the data can be explicitly allocated in the remote memory, local memory or offloading spaces (§ 4.4.2) by setting the parameter `position`. For each `RemPtr`, there are 64 bits of metadata recording the address and control information. As shown in Figure 4, the low 48 bits are used for address and the high 16 bits are used to record control information, including a `present` (P) bit for local and remote status, a `moving` (M) bit to indicate if the object is being moved by the evacuator, an `accessed` (A) bit and a `dirty` (D) bit for hotness tracking, eight bits to record the number of references (`ref_count` (RC)). There are also several `reserved` bits that can be used by the developers to customize their own hotness tracking mechanisms, such as the CLOCK replacement algorithm [101, 111] and the LRU-like algorithm [17].

```
present   : 63
moving    : 62
accessed  : 61
dirty     : 60
ref_count : 59-52
reserved  : 51-48
addr      : 47-0
```

(a) Local object

```
present   : 63 // = 0
moving    : 62 // = 0
offloaded : 61
reserved  : 60
size      : 59-48
r_addr    : 47-0
```

(b) Remote object

Figure 4: The metadata layout of the remoteable pointer (`RemPtr`) for objects (a) in the local memory or (b) in the remote memory.

Remoteable pointer dereferencing. Beehive adds runtime barriers, *i.e.*, `deref` and `deref_mut`, to the `RemPtr` to dynamically check the the control bits of the remoteable pointer during execution for object management (referred to as *dereferencing*). For example, the object will be retrieved into the local memory if the `present` bit is unset, or the dereferencing will be hold if the `moving` bit is set. The barrier is optimized to minimize the execution overhead. For the hot path (object is already in the local memory), only five instructions are needed for the location check, which is comparable to AIFM [92]. For the slow path (object is in the remote memory), we use atomic instructions to modify the pointer, avoiding the cost of locking, and the overhead is trivial when compared to the µs-scale latency for retrieving the data from the remote memory.

Remoteable reference. A `T`-typed object in the remoteable space is uniquely owned by a `RemPtr<T>`. To avoid the runtime barrier overhead of accessing data via the `RemPtr`, Beehive lets programmers borrow the data from the `RemPtr` to a zero-runtime-overhead remoteable reference, `RemRef<T>` (or `RemRefMut<T>` for mutable access), as shown in Listing 2, for subsequent accesses. During the borrowing, the dereferencing methods, *e.g.*, `deref` for immutable borrow and `deref_mut` for mutable borrow, will be invoked to inspect the control bits of the remoteable pointer and retrieve the data to local if necessary. After the one-time dereferencing (aligned with borrowing), the data will reside in the local memory and can be accessed through the zero-runtime-overhead `RemRef` (or `RemRefMut`) as that of raw pointers.

```
elements: [RemPtr<T>; N];

impl<T, const N: usize> RemArray<T, N> {
    fn for_each<F>(&self, f: F) where F: Fn(&T) {
        for e in self.elements.iter() {
            // immutable borrow
            let r: RemRef<T> = e.deref();
            f(*r); // zero-runtime-overhead access
            // end of the immutable borrow
        }
    }
    fn initialize(&mut self, val: T) {
        for e in self.elements.iter_mut() {
            e.allocate();
            // uniquely mutable borrow
            let mut r: RemRefMut<T> = e.deref_mut();
            // let r2 = e.deref(); Compiler Error
            *r = val;
            // end of the mutable borrow
        }
    }
}
```

Listing 2: Example code of an implementation of RemArray. The programming should follow the constraints of Rust *ownership*.

```
pararoutine! {
    let remref_1 = remptr_1.deref_mut();
    *remref_1 += 1;
};
pararoutine! {
    let remref_2 = remptr_2.deref_mut();
    *remref_2 += 1;
};
```

(a) User code with pararoutine interfaces

```
let closure_1 = #[pararoutine] || -> bool {
    // inlined deref_mut()
    if !present(remptr_1) {
        post_request(remptr_1);
        yield remptr_1;
    }
    let remref_1 = RemRef(remptr_1);
    *remref_1 += 1;
};
scheduler.run_async(closure_1);
let closure_2 = /* ... */;
scheduler.run_async(closure_2);
```

(b) Compiler generated pararoutine closures

Listing 3: Code sample of pararoutine generation. The code snippet of pararoutine is created as a closure [7],*e.g.*, `closure_1`, and passed to the scheduler.

Pararoutine scope. We let developers explicitly define the pararoutine scope and rely on the compiler to finish the generation of the pararoutine, *e.g.*, frame allocation and free variable capturing (Listing 3). Automatic parallelization (relying on the compiler to analyze and automatically split the code into concurrent pararoutines) remains a very challenging problem [28, 100, 110] beyond the scope of this paper. Hence, we leave the automatic parallelization as future work and let developers manually split the code. Note that each remoteable pointer dereferencing and the following accesses must be wrapped by a pararoutine, otherwise a compilation error occurs.

Ownership and program dependency analysis. Beehive forces programmers to follow Rust's *ownership* constraints when developing applications, providing the compiler with a clear picture for program dependency analysis (§ 4.3). Each object in Beehive should have only one active mutable reference or multiple immutable references at any time. To modify an object, the type system ensures that the object is borrowed as mutable by one and only one remoteable reference, *i.e.*, `RemRefMut`, from the remoteable pointer, *i.e.*, `RemPtr`, as shown in Listing 2 (Line 16). Another mutable borrow can only occur if the previous `RemRefMut` goes out of its scope (the end of its lifetime), or explicitly releases the mutable borrow. As a result, the compiler can precisely track the flow of mutable borrowing statically, which is essential for addressing the pointer aliasing during program dependency analysis [26, 98] and data races [6], which form the basis of pararoutine generation and scheduling.

4.3 Beehive Rust Compiler

Beehive introduces a new unit of computation and scheduling, *pararoutine* (§ 4.4.1), which is derived from Rust's experimental coroutine feature [8], an unstable language construct currently in development. The pararoutine is defined by programmers and generated by the Beehive compiler. Beehive first statically orchestrates the pararoutines to improve the concurrency according to the program dependency graph (PDG) through compiler code motion. Then, Beehive dynamically schedule the pararoutines to avoid tail latency issues during execution (§ 4.4). The pararoutine generation can be split into three basic steps: program dependency analysis, layering and optimizations, pararoutine generation and static scheduling, as shown in Figure 5.

Program dependency analysis. Beehive should guarantee that there is no data or control dependency across the concurrently executing pararoutines. Beehive compiler uses the program dependency graph (PDG) for data and dependency analysis, which is a program representation which makes the data and control dependencies explicit for each expression, and is widely employed in program analysis [37, 81, 90]. In a PDG, each vertex, *e.g.*, u and v, corresponds to a statement, a predicate expression or a pararoutine, and an edge, $u \rightarrow v$, indicates vertex v is dependent on vertex u. A dependency edge can represent either a data dependency or a control dependency (*e.g.*, decided by the conditional and branching statements) [10]. Dependencies between statements will prevent them from executing asynchronously. The PDG analysis relies on type safety to ensure code properties like immutability, which is ensured in the safe Rust and the APIs provided by Beehive. See Appendix 4.5 for more details.

Before constructing the PDG, Beehive applied classic loop transformations such as the unrolling, fission, and fusion [37, 85] to preprocess the code to increase potential concurrency. Additionally, Beehive provides an `async_loop` API for developers for better pararoutine static scheduling. Given that loop optimization is well-studied and not our focus, we

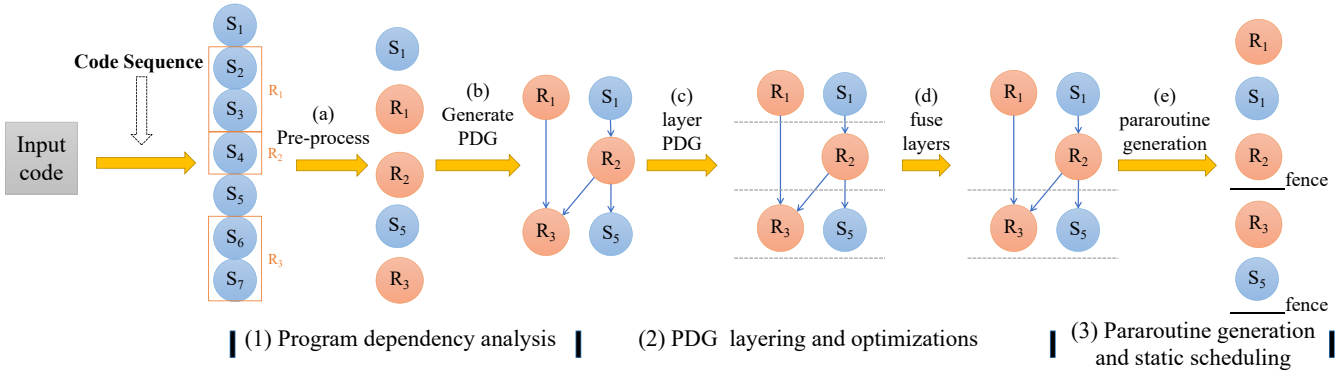


Figure 5: Automatic asynchronous code transformation. S is for statements and R is for pararoutines.

omit further details. Then, Beehive merges the statements belonging to the same pararoutine (defined by developers, as shown in Listing 3(a)) into one vertex, as shown in Figure 5 (a). Finally, Beehive creates the PDG on the optimized code sequences.

The PDG is constructed by following (Ferrante et al [37]), with pointer aliasing managed by ownership in Beehive. In particular, given two PDG vertices u, v with u preceding v in the program order, (1) a data dependency $u \rightarrow v$ exists when u, v access the same data object and at least one of them is write, and (2) a control dependency $u \rightarrow v$ exists if u determines whether or not v executes. Finally, we get a pararoutine-based PDG, as shown in Figure 5 (b).

Layering and optimizations. To enable efficient concurrent execution, the Beehive compiler groups pararoutines into *layers*. The pararoutines can execute asynchronously within each layer, while they cannot go out of the layer and must be synchronized by the fence primitives inserted between layers. Layers are initially generated by Algorithm 1. Each vertex represents a pararoutine or a statement outside the pararoutine. The compiler groups the dependency-free vertices, which have no immediate predecessors, into the same layer. Then, it removes the visited vertices from the PDG and continues the procedure until all the vertices are partitioned into different layers. Beehive applies optimizations to remove unnecessary fences by merging unnecessary layers. For example, a layer will be merged into its immediate predecessor if there is no dependency across pararoutines. This layer fusion pass is shown in Algorithm 2.

Pararoutine generation and static scheduling. The Beehive compiler statically schedules the pararoutines based on the layered PDG, as shown in Figure 5 (e). The pararoutines within a layer can execute asynchronously. Hence, Beehive can reschedule and output the pararoutines in any topology order. However, the statements and pararoutines should be scheduled by following the dependency, *e.g.*, the vertex $S1$ and vertex $R2$ must be scheduled in sequence. In addition, the compiler tends to let the pararoutine execute

Algorithm 1 Layering the PDG

```
1:  Input: G = {V, E} - PDG of pararoutines
2:  procedure PDG-LAYERING
3:      while V ≠ ∅ do
4:          Create a new layer l
5:          for all v ∈ V do
6:              ▷ Find vertices without immediate predecessors. ◁
7:              if HasNoDependency(v) then
8:                  ▷ v can be either pararoutine or statement. ◁
9:                  add v into l
10:         Remove vertices in l from V
11:         output l
```

Algorithm 2 Layer fusion

```
1:  Input: G = {V, E, L} - Layered PDG
2:  procedure LAYER-FUSION
3:      V_R ← {v ∈ V | v is a pararoutine}
4:      l ← pop the front layer from L
5:      while L is not empty do
6:          l' ← pop the front layer from L
7:          if ∃v_r ∈ l ∩ V_R, ∃v ∈ l', v depends on v_r then
8:              output l
9:              l ← l'
10:         else
11:             l ← l ∪ l'                    ▷ fuse the two layers
12:     output l
```

before the dependency-free statements, *e.g.*, $R3$ and $S5$, to prepare enough instructions to fill the idle CPU during remote access triggered by the pararoutine. After the vertex sequence is scheduled, Beehive compiler inserts the fence primitives between each layer to synchronize the pararoutines.

Each pararoutine is closed by making free variables that are used but not defined in this pararoutine captured and accessible in this pararoutine [73]. The generated pararoutine is as shown in Listing 3.

4.4 Beehive Runtime

The Beehive Runtime is responsible for task scheduling and memory management.

4.4.1 Pararoutine

Beehive introduces a stackless compute abstract, *pararoutine*, which is derived from the Rust experimental coroutine [8]. A pararoutine contains a series of compute instructions and at least one remoteable pointer dereferencing (and subsequent access through remoteable references). The pararoutine supports lightweight cooperatively scheduling at the dereferencing point and can be preempted at any point with heavier overhead.

Cooperative scheduling. The efficient cooperative scheduling of pararoutinelets the Beehive programs achieve higher remote memory concurrency while consuming less CPU resources. Pararoutine's cooperative scheduling outperforms the Shenango uthread switching by up to 2.3×, mainly due to two reasons: the reduced context and the lock-less waiting queue.

First, Pararoutines significantly reduce context switching overhead. Pararoutine switching only incurs one extra continuation invocation, which is as lightweight as a function invocation and leads to one store instruction to register continuation, as few as four instruction to restore context (*e.g.*, `mov, pop`), one `ret` and one indirect jump instruction. Compared to pararoutine switching, the uthread switching incurs several times more register and memory operations, including uthread state maintenance, lock operations, and register saving and restoration. For example, the Shenango uthread switching involves 30 assembly instructions, including 21 memory operations and 1 lock operation [83].

Second, exploiting concurrency within the uthread helps reduce the scheduling overhead. During the switching, only three instructions are needed to push the pararoutine into a lock-less per-thread waiting queue or pop a ready pararoutine for execution. On the contrary, the uthread needs to consider more complicated scheduling conditions, such as task stealing and shedding, which involves a series of uthread state check and locks [83]. Note that Beehive also supports task stealing at the uthread level, which works well to achieve high throughput and low latency (§ 5).

Beehive also supports thread switching when there is insufficient concurrent pararoutines within a thread. Specifically, when the program reaches a pararoutine fence, the thread will execute the scheduling code and poll the pending continuations, which may result in thread switching if it cannot acquire any ready task to process for 1 µs (an empirical threshold). Although thread switching may impair locality, we prioritize improving CPU efficiency.

Preemptive scheduling. Cooperative scheduling is good at achieving high CPU efficiency but usually fails to maintain µs-scale tail latency [70]. This is especially true because pararoutines are defined statically by developers without the benefit of runtime information. Some pararoutines can be large and lead to long tail latency issues due to the head-of-line blocking. For example, the suspended pararoutines (continuations) can not be rescheduled to run in time due to waiting for the completion of a large pararoutine, resulting in violations of the service level objective (SLO). Hence, Beehive provides µs-scale preemption to the pararoutine to improve the predictability of latency critical applications.

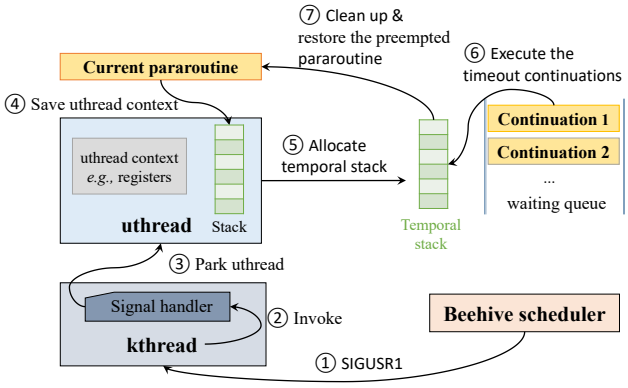


Figure 6: Preempt the large pararoutine to avoid tail latency issues

The pararoutine preemption is implemented on the SIGUSR1 signal, and the procedure is shown in Figure 6. When the scheduler detects that a suspended continuation exceeds the threshold of waiting time, it sends a SIGUSR1 signal to the kthread that the uthread runs on and invokes the signal handler. As a result, the uthread that executes the pararoutine (that leads to the blocking) will be interrupted and parked. The uthread context will be saved on the stack frame. Next, the signal handler will allocate a temporal stack and pop the stranded continuations from the waiting queue to run. Finally, the scheduler will resume the suspended pararoutine on the original stack frame. The preemption consumes 1.2 µs, which is comparable to the uthread preemption [82].

4.4.2 Memory management

Beehive's heap is divided into three spaces, as shown in Figure 7: the *native space* that resides in the local memory and is used for local-only data; the *remoteable space* that spans the local and remote memory, and the data can be swapped in and out on demand; and the *offloading space* that is used for supporting the offloaded computations.

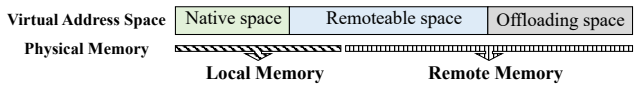


Figure 7: Memory layout of the Beehive application

Object allocation. If the programmers allocate data via the native (original) Rust data types, such as the smart pointer (`Box<T>`), the data will be put into the *native space*, which

is pinned in the local memory, and managed by the original Rust allocator.

Programmers need to explicitly use the remoteable pointer to allocate objects into the remoteable space. The objects are initially allocated in local memory and will be evicted by the *evacuator* to the remote memory when becoming cold. However, programmers can explicitly adjust the allocation site to the remote memory (for cold data) or offloading space by tuning the parameter, `position` (Listing 1). Beehive utilizes the log-structured allocator [24, 92, 93] to manage the remoteable space. Each thread maintains a thread local allocation buffer (TLAB) to reduce the lock contention. The TLAB is divided into a series of *segments*, which is set as 8MB in default by following the original design [93].

Beehive reserves a dedicated address space as the offloading space. The offloaded data is also managed by the remoteable pointer, *i.e.*, `RemPtr`, with the `offloaded` bit set. Programmers can offload computations on the offloaded data and retrieve the results via the RPC call.

Evacuator. Beehive adds a concurrent evacuator to the log-structured allocator for defragmentation and cold data eviction. When memory availability drops below the threshold (20% by default), the evacuator initiates concurrent scanning of cached local memory objects and evicts cold data to remote memory using the CLOCK replacement algorithm. This lightweight approach achieves comparable effectiveness to LRU-like alternatives [24, 111]. There are four reversed bits in remoteable pointers (Figure 4(a)) that enable optional implementation of the LRU-style algorithm.

Beehive adopts AIFM's design, *i.e.*, *dereference scope*, to synchronize the application threads with the concurrent evacuation threads to avoid data races. However, instead of using the dereference scope as a coarse-grained fence, *i.e.*, the evacuator is blocked until the application threads exit their dereference scope at least once, Beehive uses the dereference scope in a per-object way similar to Atlas [24]. An object enters the dereference scope when being dereferenced, *e.g.*, `deref` or `deref_mut`, and exits the scope when the remoteable reference dies (going out the scope of the `RemRef` or `RemRefMut`). The benefits are twofold. First, fine-grained synchronization opportunistically facilitates the evacuator in reclaiming more space. Achieving higher remote access concurrency means that more objects will be pinned in the local memory (within the deference scope), and timely memory reclamation helps alleviate the out-of-memory error due to the live-lock. Second, a frequently launched evacuator helps improve the data locality.

4.5 Limitations and Generality

Beehive relies on the Rust borrow checker for PDG analysis. Consequently, for programs with unsafe Rust code, the PDG analysis is less accurate, resulting in restricted pararoutines scheduling (§ 4.4). For example, the Rust compiler cannot properly handle the interior mutable types such as `Cell` and `RefCell`, which are implemented with `UnsafeCell`. As a result, Beehive will give up the PDG analysis and automatic asynchronization for these affected pararoutines. For libraries with source code, safety violations can be identified by checking for unsafe code blocks [33]. In this case, Beehive can skip applying automatic asynchronization to the affected code snippets to ensure correctness. On the other hand, the code transformation should be disabled if unsafe binary libraries are used due to a lack of effective ways to detect the usage of unsafe code. In this situation, developers can still manually schedule the pararoutines through the provided Beehive APIs, *e.g.*, `async` and `await`, as conventional asynchronous programming frameworks do [27, 34, 35].

In addition, the design of Beehive is not limited to Rust borrow checker. For example, Beehive can adopt deterministic parallelization [18] and reference immutability [41, 76, 80] to perform PDG analysis and automatically transform the code into asynchronous mode in other languages. We chose to implement Beehive on Rust because it is popular and widely adopted by the system community. System developers are more familiar with Rust's programming syntax. On the other hand, Beehive is designed to mitigate the µs-scale hardware latency while preserving applications' data locality. Hence, Beehive's techniques can be potentially applied to emerging storage devices exhibiting similar I/O latencies. We leave these as future work.

5 Evaluation

5.1 Setup and Methodology

We wrote about 8,000 lines of Rust and C++ code to implement Beehive. We ran experiments with one compute server and one memory server connected by the 100 Gbps Mellanox ConnectX-5 InfiniBand adapters. Each server is equipped with 2 Intel Xeon Gold 6342 CPUs, 256 GB of memory. We use one memory server for the evaluation by default, while evaluating the scalability by using up to 24 servers in § 5.4.

Baselines. Beehive is built on Rust following the design of AIFM. Hence, we implement all the evaluated applications on AIFM with all the mentioned optimizations [92], *e.g.*, non-temporal hints and prefetching techniques, enabled for performance and latency comparison. In addition, we run original applications on two state-of-the-art kernel paging-based platform, Hermit [88] and Fastswap [11], as another baseline to evaluate the runtime overhead potentially introduced by Beehive. Both Fastswap and Hermit employ a paging-based data path. Hermit consistently outperforms Fastswap due to its higher remote access throughput by exploiting asynchrony during swap-out.

The Rust versions of applications exhibit very similar performance when compared to the C++ versions, as shown in Figure 8, under the 100% memory configuration. Hence, the performance benefits of Beehive mainly come from the good data locality and the lightweight scheduling. We imple-

mented asynchronous optimization by modifying less than 5% of the core codebase per application in the Rust versions. AIFM utilizes a kernel-bypass TCP/IP networking stack implemented by Shenango for uthreads, while Beehive, Hermit, and Fastswap employ RDMA. The TCP stack overhead in AIFM is less than 2% [92], meaning that differences in the network stacks should not impact the conclusions drawn from the evaluation.

Workloads. As shown in Table 1, we evaluated seven applications under eight workloads, including Llama2 [55, 102], an open-source large language model, Multi-Grid PDE [65], a scientific partial differential equation solver, Social Network [39], a web service framework , and a synthetic workload, Key-Value Store, that is implemented in microservice, as well as Ligra [97], a graph processing framework, LiveGraph [114], a graph database, and DataFrame [74], a real-world data analysis program.

From the perspective of memory access patterns, these benchmarks can be divided into three categories: regular, random and mixed.

First, Llama2 (LMA) and Multi-Grid PDE (MGP) are both memory-intensive with good spatial locality when processing their data in arrays and vectors. The main difference between these two applications is that LMA exhibits regular access patterns, *i.e.*, sequential patterns, when performing matrix multiplication. However, MGP exhibits mixed access patterns due to its iterative stencil computation on two grids. Llama2 is a collection of pretrained and fine-tuned large language models (LMA) optimized for dialog use cases. In the evaluation, we use the float32 model with 7 billion parameters and deploy it on the CPU side. Performing inference in CPU has been widely adopted by a series of studies [44, 62]. As mentioned in § 3, MGP uses the V-cycle multi-grid method to compute a 3D scalar Poisson equation on grids.

Second, Social Network (SN) [39] and Key-Value Store (KVS) both exhibit random memory access patterns with poor spatial data locality. SN is a web service that stores the user relationship [91], *i.e.*, a dynamic sparse graph, into multiple hash tables and allows users to retrieve, update and delete the user data, which exhibits skewed access patterns. The KVS stores its data in a huge hash map and allows clients to manipulate the data by keys under uniformed distribution [4].

Third, Livegraph (LG) [114], Ligra (L) [97] and DataFrame (DF) [74] are data analytic frameworks exhibiting mixed memory access patterns. LG is a transactional graph database for real-time graph analytics. LG achieves high throughput by ensuring that adjacency list scans, a key operation in graph workloads, are purely sequential while exhibiting complicated patterns when accessing the vertices and edges. Ligra is designed for a single multicore machine and large shared memory, which is an order of magnitude faster on a per-core basis for graph traversal than distributed graph systems. The vertices and edges are stored in separate arrays. Each thread needs to access the vertices array and the edge array back and forth, exhibiting mixed access patterns. DF is a data analytics framework in Pandas [79]. The DF organizes data as a table-structured in-memory data structure and does a series of operations, such as *slicing*, *copy* and *aggregate*, which exhibit different access patterns.

Application	Dataset
Llama2 (**LMA**) [102]	Llama-7B [55, 102]
Multi-Grid PDE(**MGP**) [65]	NAS-benchmark [65]
Social Network (**SN**) [39]	Facebook network-A-anon [91]
Key-Value Store (**KVS**)	Synthetic, 50% Get, 50% Insert [4]
Livegraph Betweenness Centrality (**LG-BC**) [114]	Arabic-2005 23M Vertices, 640M Edges [91]
Ligra Connected Component (**L-CC**) [97]	Synthetic 134M Vertices, 1B Edges [14]
DataFrame Taxi (**DF-T**) [74]	NYC Taxi [2]
DataFrame Flight (**DF-F**) [74]	Airline on-time Performance [3]

Table 1: Applications and inputs for the evaluation.

Setup. Every application is running with five different local memory configurations: 13%, 25%, 50%, 75% and 100%, each of which represents a ratio of the whole memory usage of the application that can fit in local memory. The first four configurations are used to evaluate applications' performance when using different size of remote memory. The 100% (referred to as *all local*) is used to demonstrate the runtime overhead of AIFM and Beehive, introduced by barriers, profiling and other bookkeeping overhead [92]. All applications have the same number of application and evacuation threads. The number of application threads are tuned to optimal for each application (§ 5.4).

5.2 Throughput

We first measured the throughput of the applications with varying local memory ratios. Beehive outperforms Fastswap, Hermit and AIFM, respectively, by 4.26×, 3.05×, and 1.58×, over the eight workloads from 13% to 75% local memory. Beehive demonstrates particularly strong performance for workloads with well-partitioned data across threads and parallelizable, independent work within threads, such as MGP and DF. These types of applications benefit significantly from Beehive's ability to maintain data locality while exploiting asynchronous execution within threads.

Llama2 (LMA) and Multi-Grid PDE (MGP). As mentioned in § 5.1, both LMA and MGP are memory-intensive applications with good spatial locality. LMA mainly performs matrix multiplication exhibiting sequential access patterns (while MGP exhibits mixed patterns due to its iterative stencil computation). The data prefetching of Fastswap, Hermit, AIFM and Beehive all work very well for LMA, resulting in little on-demand remote accesses. In addition, LMA almost has no temporal locality. Hence, LMA is not sensitive to local memory ratios under these four platforms, as shown in Figure 8(a). The performance is mainly limited by the RDMA bandwidth.

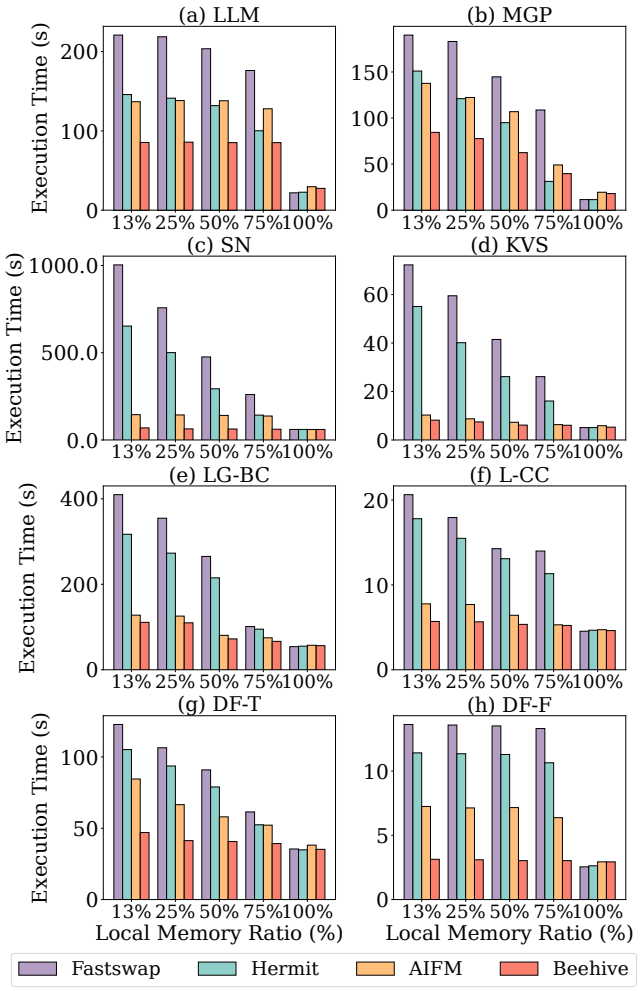


Figure 8: Throughput comparison between Fastswap, Hermit, AIFM and Beehive with varying local memory ratios.

Due to the good spatial locality of both LMA and MGP, most of fetched data will be accessed before being swapped out, which significantly alleviates the data amplification issues for the paging-based solutions, *e.g.*, Fastswap and Hermit [21, 22]. Hence, compared to Hermit, the performance improvement of runtime-based solutions, *i.e.*, AIFM and Beehive, are marginal. AIFM and Beehive outperform Fastswap by 1.56× and 2.41×, and Hermit by 0.93× and 1.47× on average, due to eliminating the overhead of trapping into kernel space and rich program semantics for prefetching during the remote memory access. At the same time, as the local memory ratio increases, *e.g.*, reaching up to 75%, AIFM and Beehive even start underperforming Hermit, *e.g.*, by 36% and 21% for MGP, due to runtime overhead, such as the barrier and profiling overhead. During the process, MGP continuously updates the data, leading to a large amount of dirty data being swapped out to remote. Compared to Fastswap, the asynchronous reclamation of Hermit [88] achieve by up to 1.94× higher throughput.

However, we still observed non-trivial performance improvement by utilizing the async execution model. Beehive outperforms AIFM by 1.58× and 1.54× on average, correspondingly under four remote memory configurations for LMA and MGP. This is because LMA and MGP partition the data, *e.g.*, matrices and arrays, into a bunch of tiles and assign them to different threads, leading to poor data locality during thread switching. The LLC misses sharply increase as the local memory shrinks, which enlarges the thread switching frequency due to escalating remote accesses. Taking MGP as an example, on average, AIFM introduces 3.5×, 5.3×, 5.5× and 7.3× more LLC misses than Beehive under 75%, 50%, 25% and 13% local memory ratios, correspondingly.

Social Network (SN) and Key-Value Store (KVS). SN exhibits random memory access with highly skewed distribution, while KVS exhibits uniformed memory access pattern. As a result, under the four remote memory configuration, both Beehive and AIFM significantly outperform Fastswap by 8.27× and 5.14×, and Hermit by 5.4× and 1.7× for SN and KVS on average, respectively. Beehive and AIFM fetch data in fine-grained object granularity, significantly reducing the data amplification seen in Fastswap and Hermit.

Unlike in LMA and MGP, the data of KVS is shared by application uthreads. Each uthread scan the data randomly so that the uthread does not have its own hotspot. Hence, even the uthread switching number reaches up to 2.3M/s under 13% local memory, Beehive's LLC misses is only slightly 2.6% less than AIFM. However, Beehive's scheduling overhead is 6.9× smaller than AIFM, leading to 1.28× higher throughput. As discussed in § 4.4, although Beehive has the same number of application uthreads, the number of active uthreads is much less than in AIFM. In addition, the pararoutine switching is 2.3× faster than uthreads. Due to the skewed workload of SN, the number of remote access is significantly reduced. However, under the four remote memory configurations, Beehive still outperforms AIFM by 2.21× on average.

Livegraph BC (LG-BC) and Ligra CC (L-CC). Due to the mixed access patterns, both Livegraph and Ligra have I/O amplification issues. Hence, compared to Fastswap and Hermit, Beehive improves the performance by 3.03× and 2.44×for LG-BC, and 3.04× and 2.62× for L-CC, correspondingly. AIFM improves the performance by 2.67×, 2.15× and 2.46×, 2.12× for L-BC and L-CC, correspondingly.

Compared to AIFM, Beehive's performance improvement is modest, by 1.13× and 1.23× for LG-BC and L-CC, respectively, on average. The performance benefits of Beehiveare limited mainly due to two reasons: first, the complicated graph results in intricate PDG, which limits the number of generated pararoutines in each layer (*e.g.*, 3.8 pararoutines per layer on average for the L-CC); second, the LG-BC is very memory-intensive: the average RDMA bandwidth reaches up

to 8.7GB/s under the 13% local memory. Both Beehive and AIFM are quickly limited by the limited network bandwidth.

DataFrame Taxi (DF-T) and Flight (DF-F). We perform two sets of operations on two different datasets. For the DF-T, a series of *Filter*, *Range extraction*, *GroupBy* operations are performed on the NYC Taxi dataset [2], while *Aggregate* and *Copy* are performed on the Flight data set [3]. Offloading is disabled in Beehive and AIFM to evaluate the efficiency of remote memory access. Compared to Fastswap and Hermit, the performance improvements of AIFM is less significant for the DF-T (1.46× and 1.25× on average) than DF-F (1.94× and 1.6× on average). This is mainly because DF-F contains more fine-grained data access.

For DF-T and DF-F, Beehive outperforms AIFM by 1.54× and 2.27× on average, under the four remote memory configurations. The performance improvements mainly come from the reduced LLC misses. As aforementioned, DataFrame organizes data as a table-structured in-memory data structure that contains millions of columns and rows. The columns are sliced into chunks and evenly assigned to different uthreads for load-balancing and high parallelism. For AIFM, frequent uthread switching leads to frequent changes of thread local data, resulting in 2.8× and 8.8× higher LLC misses than Beehive, for DF-T and DF-F, respectively.

5.3 Latency

We then evaluated the latency of the applications with a fixed configuration of 16 cores and 13% local memory ratio. We choose Social Network (SN) and Key-Value Store (KVS) to measure the tail latency of different platforms under varied offered load.

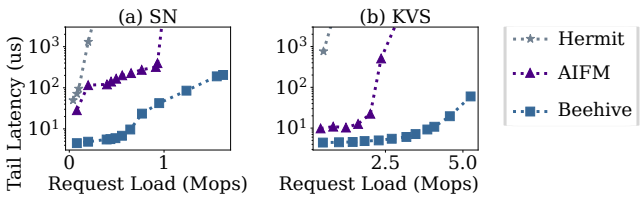


Figure 9: 90th latency as a function of offered load

Social Network(SN). Both SN and KVS exhibit random access patterns, leading to non-trivial I/O amplification issues for Hermit. Hence, the 90th tail latency of Hermit rapidly grows under the request rate of 0.1 Mops and 0.5 Mops for SN and KVS, respectively. Compared to AIFM, Beehive improves the compute capacity of cores by reducing the LLC misses (by 40%) and scheduling overhead (by 80%), significantly reducing the queuing latency of SN. As a result, Beehive's tail latency stays low until 0.64 Mops, and then the tail latency rapidly grows.

Key-Val Store (KVS). KVS is similar to SN as they both access key-value pairs from hash tables. The difference is that the request key in KVS follows a uniform distribution, as opposed to highly skewed accessing in SN. For the 90th latency, Beehive outperforms AIFM and Hermit by 5.5× and 2,869× under the load of 2Mops. Beehive achieves superior efficiency in two aspects—(1) improved CPU efficiency due to reduced WSS and (2) the effective preemptive scheduling.

5.4 Performance Drill Down

Sensitivity to the number of threads. As demonstrated in Figure 10, at first, the performance of four kinds of applications running on AIFM all increases with the number of uthreads due to improving CPU efficiency by switching to other ready tasks after issuing the requests, effectively hiding the µs-scale far memory access latency. However, as the number of uthreads increases, applications' throughput decreases to varying degrees due to LLC misses and scheduling overhead, exhibiting different optimal numbers of uthreads for AIFM.

On the contrary, Beehive is not sensitive to the number of uthreads. The performance differences are within 4% for the evaluated applications. This is due to Beehive's cooperative scheduling approach, where each uthread only yields after completing its task, effectively maintaining only one active uthread per core. L-CC-Beehive exhibits the most significant performance degradation. This is because the Beehive compiler cannot generate enough pararoutines for each layer, eventually resulting in uthread switching (§ 5.2).

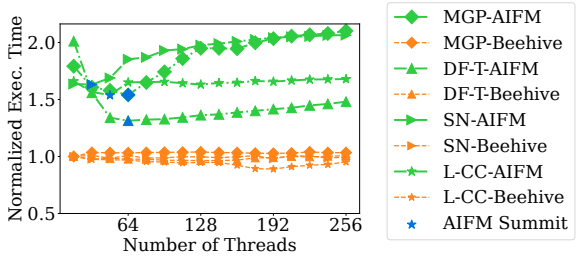


Figure 10: 16 cores are used, with 13% local memory ratio. The execution time is normalized to Beehive with 16 application uthreads.

LLC miss and scheduling overhead. In this section, we evaluate the LLC miss and scheduling overhead between Hermit and Beehive for four kinds of applications, as shown in Figure 11. Compared to AIFM, Beehive reduced LLC misses by 57% and scheduling overhead by 82%, on average.

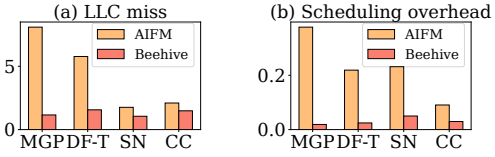


Figure 11: LLC miss (normalized to Hermit) and scheduling overhead to the CPU time, under 13% local memory ratio.

Scalability of the memory servers. Developing memory-disaggregated applications based on µs-scale I/O fabrics, *e.g.*,

InfiniBand [1], has two main benefits—first, it can utilize the conventional hardware; second, it can build a memory pool larger than rack-scale. Hence, we evaluate the scalability of Beehive by requiring memory resources from up to 24 memory servers (connected to one switch). As shown in Figure 12, performance degradation is limited to 2.5% for all the applications. However, we observed significant performance variation for the SN. The root cause is that SN exhibits random memory access patterns with a large memory footprint, resulting in frequent and unpredictable queue pair (QP) switching between different servers. According to previous research [34, 103], excessive QP switching can potentially lead to performance fluctuation.

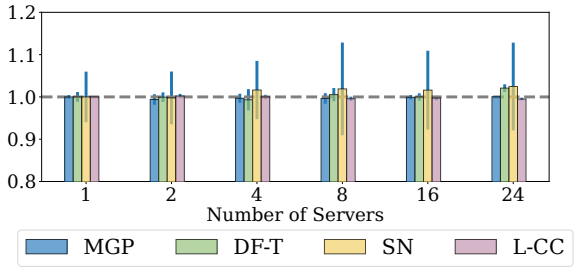


Figure 12: Impact of the number of memory servers on performance, with error bars, normalized to using one memory server

6 Related Work

Emerging hardware. Hardware-based schemes, such as the Pond [60] and Kona [21], are expected to provide direct load/store access to remote memory in cache line granularity, sub-microsecond-level data accessing latency, and hardware-support cache coherency across servers. However, these schemes need specialized hardware that is not commercially available at this moment. In addition, they still need software support to mitigate the µs-scale latency due to inefficient concurrency of remote accesses [15, 57]. For example, the number of miss status holding registers (MSHRs, used by CPUs to track the on-the-fly memory requests) is so limited that it cannot provide enough memory parallelism to hide nanosecond-scale local memory (DRAM) access [57, 104].

Kernel paging. Kernel-based schemes are popular because they are transparent to existing applications. The remote memory works as the second-tier memory layer and is managed by the kernel paging system —the cold pages will be swapped out to remote memory when there is not enough local memory space, and the swapped-out pages will be fetched back when being accessed by the application. However, the kernel-based schemes are notorious for non-trivial page fault overhead and I/O amplification [22, 24, 29, 89, 92] that not only degrades the throughput but also leads to performance variation and latency issues [88, 108].

Prefetching. Prefetching is effective to hide the data access latency. The prefetching techniques have been studied extensively, and a series of software-based [68, 71, 92, 108] and hardware-based [50, 51, 61, 96, 113] prefetchers are proposed. Beehive can also benefit from these works and adopts similar semantics-guided prefetching techniques as AIFM [92].

Disaggregated memory frameworks. A series of frameworks [20, 23, 34, 35, 53, 54, 67, 75, 77, 78, 99, 103, 106, 107] have been proposed for distributed memory models, such as distributed shared memory (DSM) [63] and disaggregated memory [95]. Although some of these works adopt asynchronous programming models, they mainly focus on providing high-performance asynchronous remote access primitives [34, 35, 78, 103] or reducing the synchronization overhead while maintaining the data coherency across distributed servers [20, 67, 106]. The developers are still expected to handle the asynchronous remote access manually.

For example, Storm [78] proposes *one-two-sided* operations, which use one-sided reads for simple data retrieval and two-sided RPCs for complicated memory operations, such as doing pointer chasing in a tree structure. In addition, FaRM [34] provides an event-driven programming model and still asks developers to define and register the *continuations* explicitly, which Beehive generates them automatically. For DSM frameworks, GAM [20] adds an extra level of cache atop the distributed memory to reduce costly remote memory access by exploiting applications' data locality, while dRust [67] reduces RDMA communications further during the cache coherency maintenance by leveraging the ownership semantics. In addition, RackOut [77] re-organizes the data in rack-scale granularity to improve the load balance of pooled memory. These works are orthogonal to Beehive, which mainly focuses on improving the data locality and reducing the scheduling overhead for applications exhibiting excessive thread switching. Their techniques can be potentially integrated into Beehive to enhance the network performance. We leave this to future work.

7 Conclusion

We present Beehive, a runtime framework developed for memory-disaggregated platforms. Beehive achieves superior remote access concurrency by exploiting the asynchrony within each thread while keeping the programming usability as the multithreaded synchronous programming framework.

Acknowledgment

We thank the reviewers for their comments and are particularly grateful to our shepherd Malte Schwarzkopf for his feedback. This work is supported by National Key Research and Development Program of China under grant 2024YFE0204100, the National Natural Science Foundation of China (62090024, 62232015, 62302479,U23B2020) and the Innovation Funding of ICT, CAS (E361010, E261110). Dr. Jie Zhang is supported in part by the Natural Science Foundation of China under Grant No. 62332021 and 62472007.

A Appendix of Beehive Programming

A.1 Case Study for Programming Effort

To demonstrate Beehive's programming efficiency, we have refactored and minimized a representative code snippet from the Social Network (SN, Table 1) application to construct a minimal working example. Through careful abstraction of non-essential implementation details, we have distilled the core functionality while preserving the essential characteristics of the original code snippet. Next, we will use the function `post_reply`, which is used for posting a reply from user A, *i.e.*, `sender`, to user B, *i.e.*, `receiver`, as an example to demonstrate the programming effort of Beehive when compared to that of AIFM.

Let us start from implementing the function in AIFM, as shown in Listing 4. When the `sender` replies to the `receiver`, the `post_reply` function accesses their profiles stored in remote objects to generate a textual message "`sender name` replies to `receiver name`". A dereference scope is created in Line 2 to synchronize the application threads with the evacuator in the specific scope, *i.e.*, from Line 2 to Line 14, to prevent potential data races (Section 4.4.2). The evacuator will be blocked when the remote objects, such as the object referenced by `sender` in Line 7, are being dereferenced and accessed. Lines 5-6 retrieve `sender_ptr`, the remoteable pointer referencing the `sender`'s profile data. This pointer is then dereferenced to fetch the corresponding data from the remote in Line 7. The value of the filed `name` of `sender` is copied in Line 8 after the remote access is complete. Lines 9-12 perform analogous operations to obtain `receiver`'s name. Finally, the message is composed of data retrieved from the remoteable objects.

```
1  std::string post_reply(const Request& req) {
2      DerefScope scope;
3      std::string sender_name;
4      std::string receiver_name;
5      RemUniquePtr<User>& sender_ptr =
6          get_user(req.sender_id);
7      const User* sender = sender_ptr.deref(scope);
8      sender_name = sender->name;
9      RemUniquePtr<User>& receiver_ptr =
10         get_user(req.receiver_id);
11     const User* receiver = receiver_ptr.deref(scope);
12     receiver_name = receiver->name;
13     return sender_name + " replies " + receiver_name;
14 }
```

Listing 4: Programming with the AIFM.

The code in Listing 5 is basically the same as those written with AIFM, except for some annotations. To enable asynchronous execution in Beehive, developers simply annotate the two remote access operations with `pararoutine!` (Line 4, Line 9), and the compiler will generate the pararoutines as discussed in § 4.3. In contrast, manual implementation using Rust coroutines require significant additional code, including yield condition checks, waiting for coroutines to complete execution, and even manually scheduling and resuming the suspended coroutines on remote access completion. Thus manual approach substantially may increase the development effort, resulting in potential concurrent bugs and performance issues [9, 15, 56, 105].

```
1  fn post_reply(req: &Request) -> String {
2      let mut sender_name = String::new();
3      let mut receiver_name = String::new();
4      pararoutine! {
5          let sender_ptr = get_user(req.sender_id);
6          let sender = sender_ptr.deref();
7          sender_name = sender.name as String;
8      }
9      pararoutine! {
10         let sender_ptr = get_user(req.receiver_id);
11         let receiver = sender_ptr.deref();
12         receiver_name = receiver.name as String;
13     }
14     format!("{} replies {}", sender_name,
15         receiver_name)
15 }
```

Listing 5: Programming with the Beehive.

A.2 Performance Impact of Beehive Asynchronization

Figure 13 demonstrates the performance differences with (*Beehive-async*) and without (*Beehive-sync*) compiler code transformation, which is responsible for generating and enabling the synchronous execution of pararoutines. For the code written in Beehive while disabling the code transformation, the annotated pararoutines will be ignored, and the code will execute in the conventional multithreaded synchronous model—switching to another user-level thread when encountering remote accesses (§ 4.4). As a result, the performance of MGP, DF-T, SN, and L-CC on average drops by 72.6%, 70.9%, 81.1% and 32.8%, correspondingly, when compared to enabling the code transformation, *i.e.*, Beehive-async. However, Beehive-sync still outperforms AIFM by 1.09× on average due to the optimized evacuator and the adoption of atomic instructions (§ 4.4).

A.3 Preemptive Scheduling

Improvement from coroutine. Pararoutines are fundamentally based on stackless coroutines but are automatically scheduled by the compiler and runtime system. This automation eliminates the need for explicit `await` statements, streamlining the programming process. The data structure of pararoutines is designed with additional fields that support runtime scheduling and resumption (e.g., the pointer of the accessing remoteable object), enhancing operational efficiency. A significant improvement over traditional coroutines is the support for preemptive scheduling.

In addition, a cache-locality-aware allocator is implemented for pararoutine frame allocation to alleviate the context switching overhead further, drawing inspiration from the slab [19] and the TCMalloc [40] allocators. Each allocation is lock-free, requiring less than 10 instructions, with all frames

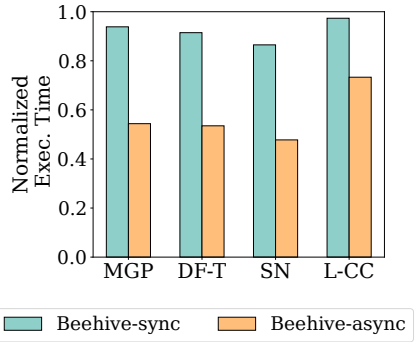


Figure 13: Performance comparison with (*i.e.*, Beehive-async) and without (*i.e.*, Beehive-sync) asynchronous remote access by enabling and disabling the Rust compiler code transformation. The execution time is normalized to AIFM. All experiments are executed under 13% local memory ratio.

allocated from a thread-private memory block that fits in the L1 cache to minimize cache misses.

Preemption. The Beehive preemption is designed to alleviate the HoL (head-of-line) blocking within latency-critical applications exhibiting wide service time distribution [32, 52] . For example, SN contains different kinds of requests whose service time, *i.e.*, execution time without queuing latency, ranges from 0.7µs to 76µs, a tend fold difference, as shown in Figure 14. As a result, the heavy-tailed request, *i.e.*, `Compose post`, can easily block hundreds of short requests and enlarge their execution time by more than 200×, hurting the overall responsiveness. This is because, when compared to short requests, heavy-tailed requests usually contain much larger and more pararoutines, which can occupy excessive CPU resources under the cooperative scheduling. In this section, we set an execution threshold, *i.e.*, 14µs (20 times the average execution time), for the short requests to demonstrate how preemptive scheduling help improve the SLOs of SN. Note that the service time of KVS requests, *i.e.*, `Get` and `Insert`, that are used in this paper are both short requests, as shown in Figure 14. Hence, we only evaluate the SN in this section.

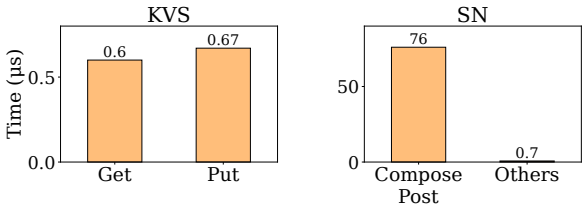


Figure 14: The service time of requests used in SN and KVS.

We tested the effects of preemptive scheduling and cooperative scheduling on request latency under a local memory ratio of 13% and a load of 0.8 Mops to avoid lacking computing capability. As illustrated in Figure 15, preemptive scheduling significantly reduces the occurrence of short request 99th latencies exceeding 20µs. The tail latency is 85% smaller than that of cooperative scheduling on average. At the same time, the 99th latency of the heavy-tailed request is slightly enlarged by 30% on average when compared to that of cooperative scheduling. Note that we add a preemption mechanism to the stackless pararoutine without fine-tuning its scheduling policy, which is not the main contribution of this work. The previous µs-scale scheduling policies, such as the Concord [49], Perséphone [32] and Shinjuku [52] are orthogonal to Beehive. We will improve the scheduling efficiency according to previous research in future work.

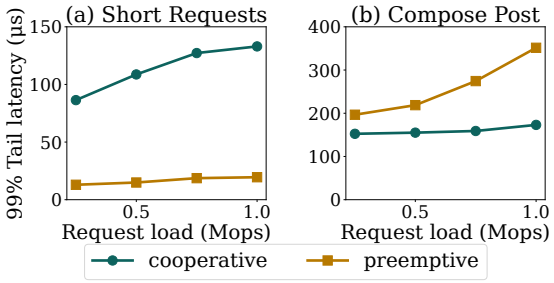


Figure 15: The 99th tail latency of SN requests under 13% local memory.

References

[1] *White paper: Introduction to InfiniBand*. Mellanox Technologies Inc., 2003.

[2] Tlc trip record data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2016.

[3] Airline on-time performance data. https://www.kaggle.com/datasets/bulter22/airline-data, 2021.

[4] Ycsb - yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB/, 2021.

[5] C++ coroutines. https://en.cppreference.com/w/cpp/language/coroutines, 2023.

[6] Data races and race conditions in rust. https://doc.rust-lang.org/nomicon/races.html, 2023.

[7] Rust closures. https://doc.rust-lang.org/rust-by-example/fn/closures.html, 2023.

[8] The rust unstable book. https://doc.rust-lang.org/beta/unstable-book/language-features/coroutines.html, 2023.

[9] A. Adya, J. Howell, M. Theimer, B. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.

[10] F. E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[11] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput. In *EuroSys*, 2020.

[12] N. Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, Santa Clara, CA, July 2017. USENIX Association.

[13] N. Amit, A. Tai, and M. Wei. Don't shoot down tlb shootdowns! EuroSys '20, 2020.

[14] D. Anderson, G. E. Blelloch, L. Dhulipala, M. Dobson, and Y. Sun. The problem-based benchmark suite (pbbs), v2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 445–447, New York, NY, USA, 2022. Association for Computing Machinery.

[15] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[16] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI*, pages 49–65, 2014.

[17] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, Nov. 2020.

[18] R. L. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 97–116, New York, NY, USA, 2009. Association for Computing Machinery.

[19] J. Bonwick and J. Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.

[20] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[21] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[22] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS '19, pages 127–135, New York, NY, USA, 2019. Association for Computing Machinery.

[23] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 152–164, New York, NY, USA, 1991. Association for Computing Machinery.

[24] L. Chen, S. Liu, C. Wang, H. Ma, Y. Qiao, Z. Wang, C. Wu, Y. Lu, X. Feng, H. Cui, S. Lu, and G. H. Xu. A tale of two paths: Toward a hybrid data plane for efficient far-memory applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[25] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[26] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, page 48–64, New York, NY, USA, 1998. Association for Computing Machinery.

[27] T. Contributors. Tokio - an asynchronous rust runtime. https://tokio.rs/, 2024.

[28] A. Craik and W. Kelly. Using ownership to reason about inherent parallelism in object-oriented programs. In R. Gupta, editor, *Compiler Construction*, pages 145–164, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[29] S. Cui, L. Jin, K. Nguyen, and C. Wang. Semswap: semantics-aware swapping in memory disaggregated datacenters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page

9–17, New York, NY, USA, 2022. Association for Computing Machinery.

[30] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, page 186–189, New York, NY, USA, 2002. Association for Computing Machinery.

[31] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 57–68, New York, NY, USA, 2002. Association for Computing Machinery.

[32] M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for highly-dispersed datacenter workloads with perséphone. In *SOSP 2021*, October 2021.

[33] R. P. Developers. Unsafe rust - the rust programming language. https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html, 2025.

[34] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[35] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pages 54–70, 2015.

[36] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, page 68–77, New York, NY, USA, 2005. Association for Computing Machinery.

[37] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[38] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), may 2008.

[39] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Google. Tcmalloc overview. https://google.github.io/tcmalloc/overview.html, 2023.

[41] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 21–40, New York, NY, USA, 2012. Association for Computing Machinery.

[42] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[43] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[44] Z. Guo, Z. He, and Y. Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[46] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 290–299, New York, NY, USA, 2007. Association for Computing Machinery.

[47] C. Hu, C. Wang, S. Wang, N. Sun, Y. Bao, J. Zhao, S. Kashyap, P. Zuo, X. Chen, L. Xu, Q. Zhang, H. Feng, and Y. Shan. Skadi: Building a distributed runtime for

data systems in disaggregated data centers. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 94–102, New York, NY, USA, 2023. Association for Computing Machinery.

[48] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. A case against (most) context switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 17–25, New York, NY, USA, 2021. Association for Computing Machinery.

[49] R. Iyer, M. Unal, M. Kogias, and G. Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 466–481, 2023.

[50] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[51] D. Joseph and D. Grunwald. Prefetching using markov predictors. *SIGARCH Comput. Archit. News*, 25(2):252–263, may 1997.

[52] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, Feb. 2019. USENIX Association.

[53] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, Feb. 2019. USENIX Association.

[54] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201, 2016.

[55] A. Karpathy. llama2.c. https://github.com/karpathy/llama2.c, 2024.

[56] M. Karsten and S. Barghi. User-level threading: Have your cake and eat it too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), may 2020.

[57] Y. Kim, H. Kim, and W. J. Song. Nomad: Enabling non-blocking os-managed dram cache via tag-data decoupling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 193–205, 2023.

[58] D. Lea. Dlmalloc: A memory allocator. https://gee.cs.oswego.edu/dl/html/malloc.html, 1996.

[59] D. Leijen, B. Zorn, and L. de Moura. Mimalloc: Free list sharding in action. In A. W. Lin, editor, *Programming Languages and Systems*, pages 244–265, Cham, 2019. Springer International Publishing.

[60] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[61] H. Li, K. Liu, T. Liang, Z. Li, T. Lu, H. Yuan, Y. Xia, Y. Bao, M. Chen, and Y. Shan. Hopp: Hardware-software co-designed page prefetching for disaggregated memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1168–1181, 2023.

[62] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[63] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[64] S. Li, K. Chen, J. B. Brockman, and N. P. Jouppi. *Performance Impacts of Non-blocking Caches in Out-of-order Processors*. HP Laboratories, 2011.

[65] J. Löff, D. Griebler, G. Mencagli, G. A. de Araujo, M. Torquati, M. Danelutto, and L. G. Fernandes. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Gener. Comput. Syst.*, 125:743–757, 2021.

[66] H. Ma, S. Liu, C. Wang, Y. Qiao, M. D. Bond, S. M. Blackburn, M. Kim, and G. H. Xu. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 92–107, 2022.

[67] H. Ma, Y. Qiao, S. Liu, S. Yu, Y. Ni, Q. Lu, J. Wu, Y. Zhang, M. Kim, and H. Xu. DRust: Language-Guided distributed shared memory with fine granularity, full transparency, and ultra efficiency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 97–115, Santa Clara, CA, July 2024. USENIX Association.

[68] H. A. Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *USENIX ATC*, pages 843–857, 2020.

[69] N. D. Matsakis and F. S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

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

[71] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, aug 1984.

[72] J. Meng, J. W. Sheaffer, and K. Skadron. Exploiting inter-thread temporal locality for chip multithreading. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010.

[73] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 271–283, 1996.

[74] H. Moein. C++ dataframe for statistical, financial, and ml analysis. https://github.com/hosseinmoein/DataFrame, 2020.

[75] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[76] J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 32*, pages 176–187, 1999.

[77] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The case for rackout: Scalable data serving using rack-scale systems. In *SoCC*, pages 182–195, 2016.

[78] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir, and M. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery.

[79] I. NumFOCUS. Pandas. https://pandas.pydata.org/, 2022.

[80] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In *International Conference on Software Technology: Methods and Tools*, 2008.

[81] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 12–pp. IEEE, 2005.

[82] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, pages 361–378, 2019.

[83] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango thread context swtiching. https://github.com/shenango/shenango/blob/master/runtime/sched.c#L386, 2023.

[84] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Technical report, USENIX Annual Technical Conference, 1996.

[85] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[86] V. S. Pai. Flash: An efficient and portable web server. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association.

[87] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[88] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony. In *NSDI*, 2023.

[89] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[90] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, 2008.

[91] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[92] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–332, 2020.

[93] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, USA, 2014. USENIX Association.

[94] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, page 84–94, New York, NY, USA, 2006. Association for Computing Machinery.

[95] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[96] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 42–53, 2000.

[97] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery.

[98] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232, 2013.

[99] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2l: software coherent shared memory on a clustered remote-write network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 170–183, New York, NY, USA, 1997. Association for Computing Machinery.

[100] S. Stork, K. Naden, J. Sunshine, M. Mohr, A. Fonseca, P. Marques, and J. Aldrich. æminium: A permission-based concurrent-by-default programming language approach. *ACM Trans. Program. Lang. Syst.*, 36(1), mar 2014.

[101] A. S. Tanenbaum and H. Bos. Modern operating systems. 2015.

[102] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[103] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *SOSP*, pages 306–324, 2017.

[104] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 409–422, 2006.

[105] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for High-Concurrency servers). In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, May 2003. USENIX Association.

[106] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A memory-disaggregated managed runtime. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–280. USENIX Association, Nov. 2020.

[107] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu. MemLiner: Lining up tracing and application for a Far-Memory-Friendly runtime. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 35–53, 2022.

[108] C. Wang, Y. Qiao, H. Ma, S. Liu, Y. Zhang, W. Chen, R. Netravali, M. Kim, and G. H. Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *NSDI*, 2023.

[109] C. Wang, Y. Shan, P. Zuo, and H. Cui. Reinvent cloud software stacks for resource disaggregation. *Journal of Computer Science and Technology*, 38(5):949–969, 2023.

[110] S. Westrick, M. Fluet, M. Rainey, and U. A. Acar. Automatic parallelism management. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.

[111] J. Yang, Z. Qiu, Y. Zhang, Y. Yue, and K. V. Rashmi. Fifo can be better than lru: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 70–79, New York, NY, USA, 2023. Association for Computing Machinery.

[112] W. Yoon, J. Ok, J. Oh, S. Moon, and Y. Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 266–282, New York, NY, USA, 2023. Association for Computing Machinery.

[113] H. Zhu, Y. Chen, and X.-H. Sun. Timing local streams: improving timeliness in data prefetching. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 169–178, New York, NY, USA, 2010. Association for Computing Machinery.

[114] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen. Livegraph: a transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.*, 13(7):1020–1034, mar 2020.

Making Serverless Pay-For-Use a Reality with Leopard

Tingjia Cao Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau Tyler Caraza-Harter
University of Wisconsin–Madison

Abstract

Serverless computing has gained traction due to its event-driven architecture and "pay for use" (PFU) billing model. However, our analysis reveals that current billing practices do not align with true resource consumption. This paper challenges the prevailing SLIM (static, linear, interactive-only model) assumptions that underpin existing billing models, demonstrating that current billing does not realize PFU for realistic workloads. We introduce the *Nearly Pay-for-Use (NPFU)* billing model, which accommodates varying CPU and memory demands, spot cores, and preemptible memory. We also introduce *Leopard*, an NPFU-based serverless platform that integrates billing awareness into several major subsystems: CPU scheduler, OOM killer, admission controller, and cluster scheduler. Experimental results indicate that Leopard benefits both providers and users, increasing throughput by more than 2x and enabling cost reductions.

1 Introduction

Serverless computing, also known as Function-as-a-Service, is increasingly popular in cloud computing environments [43, 51, 64]. This paradigm enables users to build event-driven applications through a collection of short-lived functions [6, 15, 18, 22, 39, 62]. It lets users define the events that trigger these functions; the platform provisions resources for function execution when said events occur [55, 64, 66]. Serverless computing is economically appealing as it has "pay-as-you-go" billing: developers are only charged when their function invocations are active [5, 13, 28]. Therefore, most cloud providers have a serverless offering, including AWS [13], Azure [5], and Google Cloud [28], and a majority of organizations running cloud workloads utilize serverless in some capacity [54, 63].

However, our examination of the details of serverless billing models (*i.e.*, how much a user is charged for an invocation, based on function configuration and resource usage), reveals more nuance to what providers mean when they advertise a "pay-per-use model" (Azure [5]) or claim that you only pay "for what you use" (GCP [28] and AWS [13]). To providers, these terms apparently mean something akin to "fine-grained billing" with respect to wall-clock time. An ideal pay-for-use platform for customers, in contrast, would charge based on actual resource consumption.

Characterizing these platforms as pay-for-use is justifiable if one makes four strong assumptions about function resource usage. The assumptions are: (1) resource usage is constant across time; (2) resource usage is similar across invocations; (3) CPU/memory usage is proportional; (4) all invocations are interactive. When these hold for a workload, most popular serverless platforms charge in proportion to resource usage; otherwise, invocations are charged for unused resources. We define a billing model, called SLIM (static, linear, interactive-only model) that captures these common billing practices.

Our first contribution is a detailed empirical evaluation of a suite of serverless functions that we have assembled (§2). The suite consists of a wide range of functions from different domains (developer, machine learning, and database workloads) and runs on varied input payloads. Our analysis shows that serverless workloads do not match SLIM assumptions. CPU and memory usage often fluctuate during a single invocation (it is not constant); resource consumption depends on input size, which varies across invocations; memory usage is rarely proportional to CPU usage; and most functions have non-interactive use cases. Put simply, "pay-for-use" models often charge for both used and unused resources.

Our second contribution is a new billing model, *Nearly Pay-for-Use (NPFU)* (§3), that can approximate pay-for-use efficiently. NPFU enables users to better express actual resource demands and eliminate SLIM assumptions. Specifically, NPFU separates memory and CPU demands and adds spot cores and preemptible memory for less urgent (non-interactive) work. Note that this approach does not depend on provider subsidization to reduce customer cost; NPFU is more flexible than current serverless approaches and can generate a win-win scenario for *both* providers and users. Users (nearly) pay for what they use and can better reach their objectives while providers can allocate their resources more effectively.

Our third contribution is an exploration of how billing models affect system design. We introduce *Leopard* (§4), a serverless system based on OpenLambda [34] that supports NPFU. Leopard builds billing awareness into several major subsystems. At the kernel level, we add support to the isolation API (cgroups), CPU scheduler (CFS), and preemption mechanism (OOM killer). These improvements enable efficient co-location of interactive and batch functions, which use reservation and spot billing, respectively. The new admission control logic leverages this support to aggressively run more functions concurrently, without sacrificing QoS. Finally, the load balancer creates more opportunities to resell unused resource reservations as spot cores and preemptible memory. Leopard demonstrates that billing must be considered as part of the system design process, rather than a late-stage addition.

Finally, we demonstrate the benefits of NPFU through rigorous experimentation (§5). We first illustrate a new methodology for comparing billing models. We show that Leopard's billing awareness improves worker throughput by 2.3x (on average). This efficiency may be delivered to customers in the form of price reductions (while keeping provider revenue per machine constant): 34% for interactive functions and 59% for batch functions for the average invocation.

2 Pay-for-Use Serverless Billing Models

When customers pay cloud providers for services, the *billing model* describes the cost of those services. Most services are highly configurable with various knobs (*e.g.*, the number of threads to use, or amount of memory), and those knobs may impact cost. Thus, a billing model consists of those knobs as well as a *billing function* that computes the cost based on those knobs and the actual usage of the service.

There are two families of billing models: pay-for-use (PFU), designed to charge based on actual resource consumption, and provisioned. Increasingly, modern services offer dual billing options: AWS EC2 allows customers to pay for instances on-demand (PFU), or commit to long-term reservations (provisioned); GCP BigQuery offers on-demand pricing (PFU) and capacity pricing (provisioned); AWS Lambda offers its original model, which charges for memory time used by invocations (PFU), as well as a newer provisioned concurrency option. Our focus is on PFU billing models for FaaS.

Pay-for-use vocabulary is frequently used to market popular serverless platforms, yet this term is subject to interpretation. We formalize a strict interpretation of PFU that we call strict PFU (SPFU) (§2.1). Real serverless billing models are usually an approximation of SPFU where the quality depends on the workload. We define another billing model in the PFU family called SLIM: a static, linear, interactive-only model (§2.2). SLIM resembles many billing models in use today, but it poorly approximates SPFU for realistic workloads (§2.3).

2.1 Idealized Billing: SPFU

We formalize a strict pay-for-use model (SPFU). In SPFU, the cost for each function invocation is computed as a constant factor multiplied by the used resources (*e.g.*, CPU time and bytes of memory) measured at arbitrarily fine granularity. The values for those constants are determined by *pricing*, a concept related to, but distinct from, *billing*. If the usage of each resource was plotted over time, the cost would be proportional to the area under each curve, where each resource is billed independently. For simplicity, we focus on CPU and memory as the billable resources, but the model is extensible to others (*e.g.*, network throughput, disk utilization). Given that a "CPU minute now" can be argued to be more valuable than "a CPU minute whenever it becomes available", SPFU may charge more for functions marked interactive (*i.e.*, urgent).

SPFU avoids billing for anything that is not a computing resource, such as function invocations, wall-clock time, or resource limits. Not charging for invocations implies that SPFU cannot have minimum charges per invocation. CPU time is a billed resource, but wall-clock time is not; a function that blocks mid-invocation can only be billed for memory during this idle time. A service offering SPFU can allow customers to configure resource limits, but these limits must be excluded from the billing function; customers might find resource caps useful to bound costs, but SPFU does not allow providers to charge more if resource caps exceed actual consumption.

SPFU billing has advantages for customers: not paying for unused resources can reduce costs, and decoupling billing from resource limits simplifies configuration. However, implementing SPFU presents challenges for providers; in particular, SPFU complicates admission control, as customers have less incentive to specify accurate resource limits. Lacking accurate limits, providers may concurrently execute too many functions, forcing functions to be terminated if a memory spike occurs; alternatively, providers may concurrently execute too few functions, leading to low utilization. A billing model that is bad for providers is usually bad for customers: providers are likely to compensate for inefficiencies with higher pricing.

2.2 Practical Billing: SLIM

We now define SLIM billing (static, linear, interactive-only model), another billing model in the pay-for-use family. SLIM is a generic billing model resembling real billing models used by popular platforms and can be viewed as an approximation of SPFU. With SLIM, users can specify a memory limit for each function, where the limit applies to all its invocations; a CPU size is calculated that is proportional to this memory limit (*i.e.*, the memory and CPU caps cannot be adjusted independently). The memory and CPU sizes are guaranteed to be available during execution. The cost of an invocation is proportional to the execution time multiplied by the memory limit, regardless of how much memory is actually used.

SLIM may still be considered a pay-for-use billing model given a weak definition of "use", *i.e.*, execution time of a function multiplied by an upper bound on resource consumption. SLIM's main advantage is that it may be better for admission control than SPFU, as customers have a strong incentive to set resource limits as low as possible. We suspect SLIM is common because it lends itself to FaaS implementations with greater profit potential than SPFU.

2.3 Does SLIM Approximate SPFU?

We now consider to what extent SLIM approximates SPFU. *For realistic workloads, are "pay-for-use" serverless platforms strictly pay-for-use?* For some workloads, SLIM and SPFU may indeed produce similar billing outcomes. We enumerate four workload characteristics that must apply for SLIM and SPFU to produce similar bills (§2.3.1), collect a suite of realistic serverless functions (§2.3.2), and evaluate whether the SLIM assumptions hold for our workload (§2.3.3).

2.3.1 Workload Assumptions

Given the following four assumptions about the workload, we can expect SLIM to approximate SPFU:

Assumption SI (Static Internal): Resource usage is constant throughout the invocation of a function. SLIM invocation cost is proportional to a resource cap for an entire invocation. If usage is close to the cap, the cap properly approximates usage; however, if resource limits are selected to provision for short bursts of peak usage, tenants will pay for unused resources.

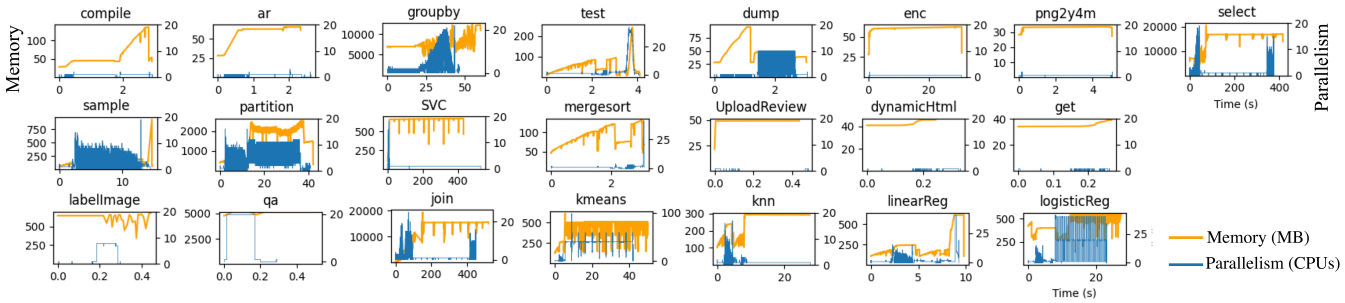


Figure 1: The diversity of parallelism behavior and memory usage over time within an invocation with medium payload.

Assumption SE (Static External): Resource usage is similar across invocations. SLIM uses same-size instances for all invocations of a given function. Resource limits are presumably chosen to be sufficient for the largest expected invocations so that those invocations will not fail. If many invocations have computational needs significantly lower than the large invocations, they will incur excess charges for unused resources.

Assumption L (Linearity): CPU and memory usage are linearly proportional. SLIM services offer instances with fixed compute-to-memory ratios (*e.g.*, 1 vCPUs to 1,769 MB of memory in AWS Lambda). As a result, CPU-dominant functions pay for unused memory resources (and vice versa). In contrast, functions that are relatively balanced across CPU and memory pay approximately for what they use.

Assumption I (Interactive): The QoS for all invocations are interactive. When invocations are triggered, SLIM-based platforms immediately provision resources. Although some triggers enable batching (*e.g.*, event hubs in Azure), when a batch arrives, the platform still immediately provisions instances for the batch. To support interactivity, providers maintain idle instances [51], although they could save memory by shutting down non-interactive instances. Under SLIM, all functions pay for low latency, even when it is not needed.

2.3.2 A Serverless Function Suite

Based on realistic workloads from a range of serverless research papers [21,23,27,39,42,47,48,70,72], we collected 22 serverless functions in 7 domains, representing applications in scientific, multimedia, engineering, and database usage scenarios, as listed in Table 1. For each function, we provide small, medium, and large input sizes. When possible, we try to anchor the sizes to actual inputs to make the workload more realistic (*e.g.*, the large input of the Compile function compiles the largest C file in the Linux kernel).

The function suite enables an analysis of SLIM assumptions across a variety of resource usage patterns. The average CPU and memory usage for each invocation varies significantly, from 0.04 CPU to 35 CPUs and from 28 MB memory to 28 GB memory, respectively. Additionally, the functions stress possible QoS: those that must complete urgently ("interactive") or those that prioritize cost and throughput ("batch"). Both types are desirable for the elasticity and pay-for-use characteristics of serverless computing [40–42, 62, 70, 73].

Application	Function	Possible QoS	
		Interactive	Batch
Software Compilation [21]	Compile	✓	
	Archive	✓	
	Testing	✓	
Video Analysis [23]	xc_dump	✓	✓
	xc_enc	✓	✓
	png2y4m	✓	✓
Batch Analysis [62]	Sample		✓
	Partition		✓
	Mergesort		✓
DB Client [39]	Upload	✓	
	DynamicHtml	✓	
	Get	✓	
DB server operations [62,74]	Groupby	✓	✓
	Join	✓	✓
	query	✓	✓
ML Inference [21,36]	ImageLabel	✓	
	Q and A	✓	
ML Training [40]	Kmeans		✓
	KNN		✓
	LinearReg		✓
	LogisticReg		✓
	SVC		✓

Table 1: A serverless function suite.

2.3.3 Workload Analysis

We now revisit those assumptions and evaluate how well they hold for realistic workloads.

Assumption SI: To evaluate whether resource usage is constant throughout a single invocation, we measure the parallelism of each function and resident memory size over time; for parallelism we trace the number of runnable tasks throughout an invocation by recording when tasks join a CPU's runqueue within Linux. As shown in Figure 1 for medium-sized functions, CPU and memory consumption varies dramatically during the execution of a single invocation.

Assumption SE: Figure 2 plots the memory and parallelism under different payloads for three representative functions. We observe that resource usage can be quite different across invocations: larger input usually corresponds to greater resource consumption. On average, CPU usage for large sizes is 1.5x higher than medium and 3x higher than small sizes, while memory usage is 8x and 16x greater, respectively.

Assumption L: Figure 3 plots the memory-to-CPU ratio for average and max usage for all functions; the line shows the memory-to-CPU ratio of AWS Lambda and extrapolated past the max support size (Google Cloud Functions and Azure Functions use similar ratios). We observe a departure from the fixed AWS Lambda ratio for most functions; 91% could use more CPU than memory relative to the AWS Lambda ratio, suggesting that CPU may be more scarce than memory.

Platform	Knobs	Instance Size r: memory to cpu ratio	Concurrent Invocations per instance	Bill per Execution T: Execution time CP: CPU unit price *MP*: memory unit price	Resource Limitation
AWS Lambda	Memory (M)	Memory: M, CPU: M*r	1	M * T * MP + M * r * T * CP	10GB Mem, 6vcpu
GCP	Memory (M)	Memory: M, CPU: M*r	1	M * T * MP + M * r * T * CP	32GB Mem, 8vcpu
Azure (premium)	Memory (M), Target	Memory: M, CPU: M*r	< target	M * T * MP + M * r * T * CP	14GB Mem, 4vcpu
Azure (consumption)	Target	Memory: 1.5GB, CPU: 1vcpu	< target	AvgMemory * T * MP	1.5GB Mem, 1vcpu
CloudFlare	Memory (M)	Memory: 128MB CPU: No dedicate CPU	1	AvgCPU * T * MP	128MB Mem, 10ms cputime

Table 2: The resource allocation policy and billing model of mainstream commercial serverless platforms.

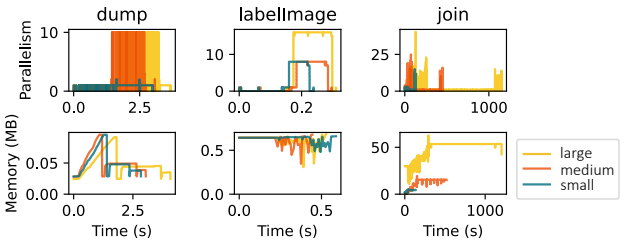


Figure 2: The resource demands with different payloads.

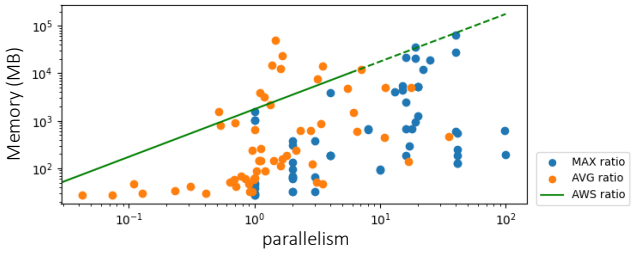


Figure 3: The ratio of cpu/memory demands.

Assumption I: In Table 1 the "possible QoS" column indicates whether a function must be urgently completed upon invocation, *i.e.*, is it latency critical (interactive) or not (batch). For example, model training is typically done offline; video analytics are used in both real-time stream processing systems and as background jobs. Thus, not all functions require interactivity all of the time.

Conclusion: Our analysis shows that serverless workloads challenge all four SLIM assumptions. CPU and memory usage fluctuate during a single invocation (assumption SI); resource consumption depends on input size, which varies across invocations (SE); memory usage is rarely proportional to CPU usage (L), and most functions have some non-interactive use cases (assumption I). Therefore, SLIM is a poor approximation of SPFU.

2.4 Serverless Billing in Practice

We now summarize the billing models in practice, as shown in Table 2. AWS Lambda, GCP, and Azure Functions (Premium Plan) closely resemble SLIM, the weakest interpretation of PFU billing. While Azure Functions (Consumption Plan) and Cloudflare align with SPFU, they enforce small, untunable resource limits (*e.g.*, Azure allocates 1.5 GB of memory and 1 vCPU and Cloudflare Workers are limited to 128 MB of memory and 10 ms of CPU time); these limits prevent customers from over-provisioning resources, but restrict the platforms' generality, prohibiting functions with larger resource requirements. Existing billing models either resemble SLIM, or SPFU with restrictive resource limit options.

3 NPFU: Nearly Pay-for-Use Billing

SPFU billing offers the most natural interpretation of "pay-for-use", but is rarely used in practice because providers must provision for resources that are unused (and unbilled). Instead, most providers adopt a billing model that resembles SLIM. We propose an alternative billing model that more closely approximates SPFU while exceeding SLIM's support for efficient and profitable FaaS implementations. We call this new model NPFU, Nearly Pay-for-Use. NPFU has two goals:

Profitable: The billing model should lend itself to efficient FaaS implementations. The drawback of SPFU is that it generally requires implementations to provision resources that may go unbilled. An ideal billing model will support greater utilization, leading to greater profits for providers (and perhaps lower costs for consumers).

Closely Approximates SPFU: SPFU offers the strongest interpretation of "pay-for-use" terminology, while SLIM only approximates SPFU for unrealistic workload assumptions. A good PFU billing model should better approximate SPFU for a wide range of workloads. Therefore, it requires adding proper knobs that enable users to describe resource requirements more accurately for workloads that breaks SLIM assumptions.

We introduce NPFU, defining the knobs for billing (§3.1) and the billing function that calculates cost per invocation (§3.2); we then compare NPFU to other billing models (§3.3).

3.1 NPFU Knobs

NPFU provides four knobs for customers to describe their resource demands: *cpu-cap*, *spot-cores*, *mem-cap*, and *preemptible-mem*. These options allow users to independently specify their CPU and memory demands instead of using the strict memory-to-CPU ratio of SLIM.

CPU: Cap and Spot. NPFU requires serverless users to specify *cpu-cap*, which is the maximum number of CPUs it can use; this bounds their cost. Given that many serverless functions are not interactive, NPFU introduces *spot-cores*: a subset of *cpu-cap* that a function does not need immediately, but could use if available. NPFU guarantees that *cpu-cap* minus *spot-cores* is available; this number of *reserved-cores* number is crucial for ensuring the function meets its QoS. Spot cores improve CPU usage: if instances underutilize their reserved cores, other instances can use them as spot cores.

Figure 4 shows an example of a server with 4 physical CPUs, where at t_0, the server hosts function invocations F1 and F2. F1 sets *cpu-cap* to 4 and *spot-cores* to 1 (*reserved-cores* is 3); F2 sets its *cpu-cap* as 3 and *spot-cores* as 2

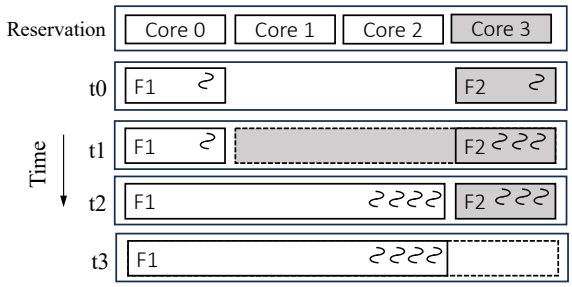


Figure 4: NPFU stresses allocated but unused cputime.

(*reserved-cores* is 1). The system ensures that each invocation can expand to its full set of reserved cores. At t_1, F2 forks two children; since there are idle CPUs from F1's reserved cores, F2 expands to use 3 CPUs. However, at t_2 when F1 forks 3 children and needs more cores, F2 immediately shrink to its reserved allocation. By t_3, F2 terminates, at which point F1, which has pending tasks, expands to all cores.

Decoupling *cpu-cap* and *spot-cores* helps batch functions describe low CPU urgency. It also benefits some interactive functions: although these functions have strict deadlines, their *reserved-cores* do not need to match maximum parallelism if deadlines can be met with fewer cores. Adding *spot-cores* creates opportunities to further reduce latency and costs.

Memory: Cap and Preemptible. Like the CPU knob, NPFU requires users to specify *mem-cap*, which is its maximum amount of memory; however, NPFU does not expose knobs for spot memory because reclaiming memory that has been borrowed by other functions can impose unacceptable overhead for interactive functions. Instead, NPFU allows users to simply specify if instances can be preempted (*preemptible-mem* as true). Preemptible instances can use the allocated-but-idle memory of non-preemptible instances; if memory usage exceeds physical memory, the preemptible instances are killed, allowing non-preemptible instances to access the reserved memory with minimal overhead. NPFU ensures that the total mem-cap of non-preemptible invocations does not exceed the available memory.

3.2 NPFU Billing Function

NPFU provides a flexible billing model that generates a win-win scenario for providers and users. NPFU leverages a used/lent model that enables users to buy/sell CPU and memory capacity from/to each other.

To calculate CPU cost, we define three kinds of cputime: *reserved-cputime*, the total reserved cputime (*execution_time* $\times$*reserved-cores*); *lent-cputime*, the idle reserved cputime used by other functions; *borrowed-cputime*, the used cputime of spot CPUs. The CPU cost, where C_r is the unit price for *reserved-cores*, and C_s for *spot-cores*, with $C_r > C_s$, is:

$$\begin{aligned} & C_r \times \textit{reserved-cputime} \\ + & \ C_s \times \textit{borrowed-cputime} \\ - & \ C_s \times \textit{lent-cputime} \end{aligned} \tag{1}$$

To calculate memory cost, non-preemptible memory acts as the lender, while preemptible instances are borrowers. For non-preemptible instances, the memory cost is:

$$\begin{aligned} & M_r \times \textit{mem-cap} \times \textit{execution_time} \\ - & \ M_p \times \textit{avg-lentMem} \times \textit{execution_time} \end{aligned} \tag{2}$$

$M_r > M_p$ since reserved memory is more valuable than preemptible memory. The cost for preemptible instances is:

$$M_p \times \textit{avg-memory} \times \textit{execution_time} \tag{3}$$

If a preemptible instance is preempted mid-invocation, the customer is not charged, and its invocation is simply requeued.

3.3 Comparison with existing PFU family

NPFU can reproduce many of the advantages of SLIM. Specifically, NPFU offers both bounded performance and bounded cost by allowing users to specify minimal resources (with *reserved-cores* and non-preemptible *mem-cap*); and maximal resource boundaries (using *cpu-cap* and *preemptible* option). Although the cost can be less predictable when functions lend or borrow CPU, it ensures a clear upper bound: for functions with *reserved-cores*, the upper bound occurs when there are no available *spot-cores* and no CPU time is lent to other invocations; for functions without *reserved-cores*, the cost is determined by the used CPU time. Furthermore, the used/lent billing function ensures that providers can maintain the same revenue as SLIM.

In terms of simplicity, NPFU introduces three more knobs than SLIM, which is acceptable as simplicity is not merely a knob count; a single knob that must be tuned with high precision is arguably more complex to tune than a handful of less sensitive knobs. SLIM offers a single "size" knob that controls CPU and memory for all function invocations, making four unrealistic assumptions and requiring precise tuning to estimate the worst-case scenario: overestimating results in an overpayment, while underestimating causes the largest invocations to fail. NPFU can be configured to behave identically to SLIM by setting *cpu-cap* to be linear with *mem-cap*, *spot-cores* to 0, and *preemptible* to false; customers who prioritize simplicity could use these default values.

NPFU takes significant steps toward approximating SPFU. NPFU addresses SLIM's workload assumptions: it enables functions to sell their "allocated but unused" CPU time and memory (removing Assumption-SI and SE); decouples CPU and memory (resolving Assumption-L); and enables users to specify their urgency on requested resource by setting reserved (or spot) CPUs and non-preemptible (or preemptible) memory (removing Assumption I). NPFU allows batch function to set *reserved-cores* to zero and use preemptible memory, aligning their billing with SPFU; for interactive functions, NPFU offers a discount on wasted resources, making it more cost-effective than SLIM.

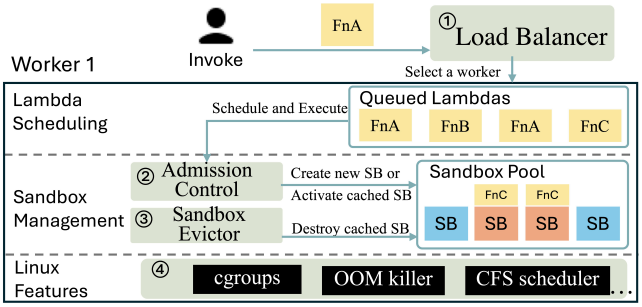


Figure 5: Overview of popular serverless platforms.

4 Billing-Aware Serverless

FaaS platforms are a composition of many subsystems, including load balancers, admission controllers, and kernel-level schedulers and resource controllers; efficiently supporting new billing models, such as NPFU, requires support from each of these subsystems. We give background for each subsystem (§4.1), describe the limitations of each for billing (§4.2), and propose Leapord (§4.3), a new FaaS platform with billing support built into every layer of the stack.

4.1 Background: Serverless Subsystems

Figure 5 illustrates the key components in a serverless platform. ① *Load Balancer:* dispatches incoming invocations to workers using a load-balancing strategy. ② *Admission Control:* selects the next request for execution and creates, or activates, a sandbox when sufficient resources are available. ③ *Sandbox Evictor:* removes cached sandboxes when too many are in memory ④ *Resource Isolation:* manages CPU and memory allocations across sandboxes. We now describe the typical implementations for each.

Load Balancing: In mainstream serverless platforms, after reaching the global controller, invocations are immediately dispatched to workers, eliminating memory usage for arguments and enabling scalability [4,34,45]. To avoid cold starts, the load balancer aims to minimize the number of workers handling invocations from the same function. For example, OpenWhisk [4] uses consistent hashing by mapping invocations to "home" workers; if a home worker is overloaded, the function is forwarded along the ring until a non-overloaded worker is found. Systems [9] built on Kubernetes [1] use greedy load balancing, sending requests to the first worker until it reaches capacity. Hermod [45] uses a greedy algorithm under low load and a least-loaded algorithm during high load.

Admission Control: The selected worker determines when to execute the incoming invocations, deciding how many invocations can run concurrently while ensuring that reserved resources are maintained for each invocation. For example, in OpenLambda [34], the provider manages the number of active sandboxes to ensure that total CPU/memory allocation across all active sandboxes does not exceed the physical resources available on each worker. Platforms typically use a priority queue; to improve average slowdown, many use Shortest Job First [16], estimating runtime based on historical data [45].

Sandbox Evictor: When a function completes, its sandbox is kept alive for future use. Cached sandboxes reduce latency but consume memory, so deciding when to evict cached sandboxes is critical. For instance, OpenLambda evicts the LRU sandbox when memory usage exceeds a threshold. Some approaches retain idle sandboxes for a specified duration [5,13,64]. FaasCache [26] evicts sandboxes only when memory is insufficient, evicting sandboxes with lower usage frequency, longer runtime, and larger memory footprints.

Resource Isolation: Resource isolation is crucial for maintaining performance in multi-tenant environments. To isolate requested CPU and memory resources, most serverless platforms use containers [4,9,34] and microVMs [13], such as Docker [3], Kubernetes pods [10], OpenLambda's SOCK [59], and AWS Lambda's Firecracker [11]. These solutions all rely on Linux cgroups. For memory management, cgroups provide the memory.max option, which enforces a memory limit for all tasks in the group; if a task exceeds this limit, the Out-of-Memory (OOM) killer is triggered and terminates one or more tasks based on metrics like memory consumption and age. Linux cgroups offer two mechanisms to manage the CPU: *CPU pinning* restricts tasks to specific CPUs, supporting both minimum and maximum CPU limits; *weighted sharing* relies on Linux's Completely Fair Scheduler (CFS) to allocate CPU time based on proportional shares. CFS divides available CPU cycles among threads in proportion to their weights. In multicore systems, each physical CPU has its own runqueue; load balancing periodically adjusts task distribution, moving tasks from burdened CPUs to lightly-loaded CPUs. With these mechanisms, containers can implement their own resource models. For instance, Kubernetes's "request" interface uses CPU sharing to reserve at least the requested amount of CPUs for a specific container [2]. The correctness and efficiency of its reservation depend on the implementation of cgroup.

4.2 Subsystem Support for NPFU: Challenges

Given that these subsystems were built without considering billing, each has limitations that make a correct and efficient implementation of NPFU difficult.

Challenge 1: cgroup API does not support efficient CPU reservations. NPFU uses *reserved-cores* to maintain performance and *spot-cores* to allow CPU sharing. The CPU scheduler should support two requirements. First, cgroups must be assured immediate access to their *reserved-cores* when needed (correctness); second, cgroups should be allowed to use idle *reserved-cores* of other co-located cgroups (efficiency).

The cgroup API and underlying CFS scheduler support both CPU pinning (via CPU sets) and weighted sharing (via CPU shares), but neither meets our requirements. To demonstrate the issue, we perform an experiment: two functions run concurrently, each of which has "paid" to reserve 16 of the machine's 32 cores. Function F1 runs 32 tasks continuously, and is willing to pay for additional spot cores. F2 starts with one task, fans out to 16 tasks, and does not use spot cores.

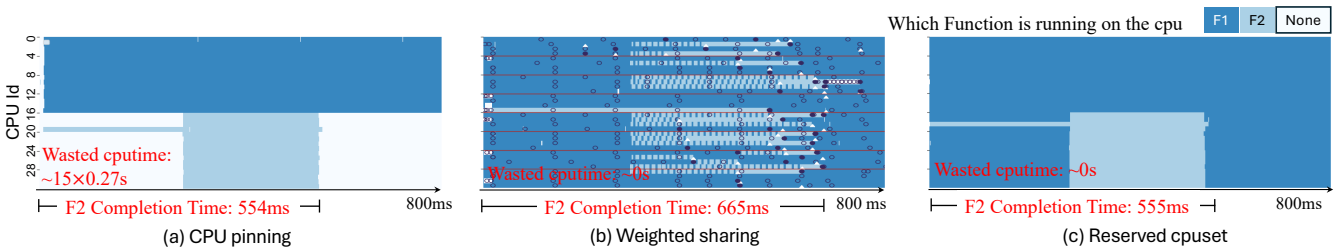


Figure 6: Colocation of Function 1 (32 processes) and Function 2 (1 process forks 15 processes at 270ms).

First, we evaluate CPU pinning via CPU sets in Figure 6(a). Reservations are implemented by applying a CPU set to all functions: the CPU set for F1 is cores 0-15, and for F2, it is cores 16-31. The figure shows that each function has sole use of its reserved cores, but the strictness of CPU sets wastes ∼4 seconds of CPU time on cores 16-31 that F1 would have been willing to purchase (at a reduced rate). Thus, CPU pinning only support inefficient reservations.

We evaluate weighted sharing with cpu.share in Figure 6(b); cpu.share the task to run on any core. We give both functions an equal share and expect each to be allocated approximately its reservation of 16 cores. The results show that no CPU is wasted because F1 uses whatever is available, but F2 is denied full access to its reserved CPUs and runs 20% slower. We conclude that cpu.share cannot correctly provide reservations.

CFS load balancing is resource-intensive, involving computational costs and cache misses, so CFS uses many heuristics to reduce overhead. For example, when CFS finds no imbalance, it doubles the interval before its next load balancing attempt and keeps the interval until the next successful migration. These heuristics undermine CFS's ability to promptly split CPU time among cgroups according to their cpu.shares. Figure 6(b) illustrates when CFS load balancing occurs: empty circles represent load balancing without migration, black circles represent migration, and white triangles show the source CPU of the migration. The figure shows that the interval between load balancing is around 100 ms. The infrequent rebalancing of cpu.shares is not sufficient for serverless workloads, where billing occurs at 1 ms granularity.

An alternative approach is to adjust the CPU set or the share weight of serverless functions over time dynamically, pushing their actual CPU usage closer to their proportional shares. However, it introduces overhead (*e.g.* adding a core to a VM takes approximately 0.1 milliseconds [14]), impacting millisecond-scale serverless invocations. The high overhead also limits the frequency of adjustments [32, 35, 46].

Challenge 2: sandbox evictor does not support preemptible memory. NPFU introduces a preemptible memory knob to allow efficient memory usage, and, to utilize idle reserved memory, NPFU allows preemptible sandboxes to be admitted even when all of memory is allocated. However, if memory is exhausted, the Linux OOM handler takes over before the sandbox evictor can react. The Linux OOM killer lacks access to sandbox-specific properties and may kill non-preemptible instances, violating the guarantees of NPFU. It also prevents the implementation of more sophisticated eviction strategies; for example, FaasCache [26] needs statistics for each sandbox such as execution recency, invocation frequency, and termination costs. Thus, NPFU requires improvements to the sandbox evictor and Linux OOM handler.

Challenge 3: NPFU complicates load balancing and admission control. By decoupling CPU and memory, NPFU introduces complexity for load balancing, leading to a multidimensional bin-packing problem and increasing the likelihood of stranded resources. This issue is worsened by locality-aware load balancing, which may assign memory- or CPU-heavy functions to the same "home" worker. Additionally, current loaded balancers rely on simple metrics like the number of invocations; consequently, interactive tasks may experience degraded performance even when another worker has sufficient unreserved resources but appears loaded due to batch tasks. Similarly, admission control that does not distinguish between spot CPUs and reserved CPUs will not know when it is safe to admit a given function invocation.

4.3 Leopard: Billing-Aware FaaS

As existing subsystems cannot handle novel billing models, we introduce a new billing-aware FaaS platform, Leopard, based on OpenLambda. Leopard implements spot CPUs and other NPFU features with several new subsystems. It introduces a new resource isolation mechanism for task groups, including CPU scheduler support for reservations (§4.3.1) and a billing-aware out-of-memory (OOM) killer (§4.3.2). It also presents a serverless-specific admission control policy at the worker level that accounts for spot CPUs (§4.3.3). Additionally, it introduces a serverless-specific cluster-level load balancer that considers both alignment and locality (§4.3.4).

4.3.1 Efficient CPU Reservations

Linux cgroups and CFS cannot support CPU reservations, so Leopard provides a new cgroup interface, *cpu.resv_cpuset*. Tasks in this new type of cgroup are guaranteed reserved CPUs on demand, but other cgroups may use the reserved CPUs if the reserver does not need them. To borrow a legal term, a cgroup reserving some cores has the *right of first refusal* for using them. Unlike the prior CPU-set interface, cpu.resv_cpuset does not prevent a cgroup from running on CPUs outside of the set; non-reserved CPUs may be used if computational needs exceed the reservation and there is a willingness to pay for additional spare compute.

Leopard modifies the CFS scheduler to support the new cgroup reservation API. The new CFS mode has two goals:

(1) Do not rely on fairness to achieve isolation. The previous approach of weighted sharing in cgroups relies on fairness, which current CFS implementations do not enforce effectively due to the high overhead of monitoring other cores for load-imbalance. In contrast, a new approach that ensures isolation for reserved cores is simpler and less resource-demanding, as it can be achieved locally without inter-core checks.

(2) Allow tasks to use flexible policies on different cores. CPU pinning allowed different policies for different cores (*e.g.*, a sandbox was permitted or forbidden on each core), but this policy was too blunt. We propose an enhanced model, where the scheduler differentiates more refined policies. Specifically, on reserved cores, a task can have the highest priority allowing immediate preemption; conversely, on other cores, the same task can have a lower priority, following spot tasks rules. This approach ensures that the new scheduler achieves both reservations and flexible sharing.

Based on these goals, Leopard adds reservation support in CFS as follows. When CPUs are added to *cpu.resv_cpuset*, the core marks the tasks as *reserving tasks*; other tasks are treated as *spot tasks*. When a core scheduler picks a task, it always picks a reserving task if any are runnable; otherwise, it pulls a runnable reserving task from the busiest runqueue in its cgroup; if that fails, the scheduler picks a spot task on the core's runqueue or makes an *idle-balance* CFS call to find a spot task elsewhere. When a reserving task wakes up (*e.g.*, after an I/O completion) and a spot task is running, the reserving task preempts the core immediately.

To maintain fairness within a cgroup, Leopard keeps the CFS design for managing tasks inside cgroups, but divides the load metric of a runqueue to track classes of tasks. When one CPU triggers load balancing, if the current task is a reserving task, the scheduler only balances the reserving cgroup load; if the current task is a spot task, the scheduler balances the load of all spot groups.

Figure 6(c) shows that Leopard's new reservation API behaves as desired: there are no wasted cores, and each task receives its reserved allocations. Specifically, F1 expands to more cores when they are available, and F2's performance is not harmed because it has top priority on cores 0-15.

In addition, to compute CPU billing in Equation 1, Leopard tracks reserved CPUtime, spot CPUtime, and lent CPUtime of the invocations by modifying the cgroup CPU Accounting (*cpuacct*) subsystem, which reports the used CPUtime of a cgroup. Linux updates the used CPUtime of a cgroup incrementally by calling *cgroup_account_cputime(Task p, int delta_exec)* when a task's state changes.

We add three interfaces in *cpuacct*: *resv_usage*, *spot_usage*, and *lent_usage*. Each time *cgroup_account_cputime* is called for a task p, (1) if the current CPU is p's reserved CPUs, add *delta_exec* to *resv_usage* of p*'s* cgroup; (2) otherwise, add *delta_exec* to the *spot_usage* of p's cgroup, and to the *lent_usage* of current CPU's reserving cgroup. The overhead is negligible, as it involves simple arithmetic operations within *cgroup_account_cputime* when the task state changes. Once the invocation finishes, the CPU billing can be computed by reading these metrics from the cpuacct interface, with less than 1 ms per invocation.

4.3.2 Sandbox Evictor

NPFU allows preemptible sandboxes to be admitted even when all memory is allocated to non-preemptible sandboxes, so memory exhaustion may occur under heavy load. However, Linux's original OOM handler sometimes takes over sandbox eviction before the userspace sandbox evictor can react. To support NPFU, Leopard contains a mechanism that gives control to the sandbox evictor during OOM events.

Leopard enables customizable OOM victim selection for different cgroups, allowing evictors to use rich user-space information. Specifically, Leopard contains two new Linux cgroup APIs for handling out-of-memory scenarios: *memory.oom.listener* specifies a user-space process to perform victim selection; *memory.oom.victim* sends the selected victim's PID to the kernel. For example, in user-space, Leopard can estimate the revenue lost for preempting different sandboxes (since preempted sandboxes are not charged) and preempt the sandbox that loses the least revenue; this information is unknown in the kernel. Specifically, when receiving an OOM signal, the evictor first selects cached sandboxes (*i.e.*, the same policy as FaasCache [26]), but if none are available, chooses the active preemptible sandbox with the shortest runtime and using the fewest resources, in order to lose the least revenue.

With this mechanism, Leopard can easily support sophisticated preemption policies. The communication overhead of victim selection in userspace is minimal ($\sim$0.4 ms), and is mitigated by proactive eviction and admission control of preemptible sandboxes. In summary, these enhancements allow Leopard's evictor to make informed decisions during OOM events caused by including preemptible sandboxes.

4.3.3 Admission Control

Leopard uses the following criteria to determine whether to admit an invocation into a sandbox: interactive invocations with *resv-cores* or non-preemptible memory are admitted if sufficient unreserved resources are available; batch invocations with preemptible memory and no *resv-cores* are admitted if currently idle memory and CPU resources are sufficient.

To determine whether or not preemption is likely, Leopard collects historical CPU and memory usage from past invocations and monitors the total of these averages for current active invocations. If the sum of historical memory usage exceeds the total memory, there is a high likelihood of preemption. In such cases, to avoid the risk of OOM events and unnecessary preemptions, Leopard does not admit the next batch function. Overcommitting CPU is less harmful, as it does not trigger preemption; therefore, Leopard admits new batch functions as long as the total historical CPU usage remains less than twice the total CPU resources.

When reserved resources are released, Leopard admits queued interactive invocations using the Shortest Job First. When the utilization of a worker falls below a threshold, Leopard admits the batch request with the shortest runtime that does not request reserved capacity. Thus, Leopard ensures high utilization for each worker while ensuring that reserved resources for active sandboxes are available when needed.

4.3.4 Load Balancer

Leopard provides a new load balancer for invocations. At low load, function placement does not significantly impact performance, so Leopard prioritizes alignment (to strand fewer resources) and locality (to improve warm-up). For alignment, we follow Grandl *et al.* [29] and define the alignment score of a task relative to a worker, *score-align*, as the dot product between the vector of task's reserved resources and the worker's available resources; a higher score indicates a better fit. To preserve locality, Leopard calculates a *score-locality* for each worker based on its priority scheduling with the traditional hashing sequence [4]. At low load, Leopard places tasks based on a combined score for each worker as $\textit{score-align} + \alpha\ \textit{score-locality}$.

At high load, the queueing delay is the most critical factor. Leopard balances the load while accounting for the QoS requirements of each function. Specifically, for interactive invocations, Leopard chooses the worker with the lowest *reserved-load*, which is the amount of reserved resources needed to complete all queued interactive jobs; $\textit{resv-load} = \sum_r r.avgRT \times (r.resv\text{-}cores + mem\text{-}cap \times !preemptible\text{-}mem)$, where r is the queued request in this worker. In contrast, for batch invocations, Leopard chooses the worker with the lowest *usage-load*, which is the actual resources used to complete all queued jobs; $\textit{usage-load} = \sum_r r.avgRT \times (r.avg\text{-}CPU + r.avg\text{-}Mem)$, where *avg-CPU* and *avg-Mem* are the historical CPU and memory utilization for the function. Thus, the load balancer ranks workers differently for interactive versus batch tasks.

4.3.5 Limitations

Further improvements on Leopard are worth considering. First, resv_cpuset does not consider placement: whether the reserved CPUs should reside on the same NUMA node and hyperthreads from the same core. Second, resv_cpuset cannot support the reservation of a fraction of a core, which may be desirable for some applications [25, 37]. Finally, conflict between alignment and locality goals can make the combined load balancer score ineffective [52]; deployment in a disaggregated cluster could be a potential solution [30].

5 Evaluation

We describe a new methodology for evaluating billing (§5.1). Then, we explore four questions: *How does Leopard with NPFU perform relative to other models* (§5.2)? *Can Leopard adapt to workload changes* (§5.3)? *Does Leopard scale effectively* (§5.4)? *How does building a serverless platform from billing-aware components improve efficiency* (§5.5)?

5.1 Methodology for Billing Model Evaluation

We discuss the limitations of existing methods when applied to billing and present a billing-oriented benchmark.

5.1.1 Existing Workloads

The Microsoft Azure dataset [64] provides the most complete details on invocations in a major serverless platform, including 52,000 functions invoked 8.8 billion times over 14 days. The dataset is divided into three major components: *Invocations*, the number of function invocations per minute; *Execution Time*, average and percentile execution-times per function; *Memory Usage*, average and percentile memory usage per application. However, the Azure dataset does not contain CPU usage data or details on resource fluctuations over time. Thus, researchers often use synthetic workloads and make some assumptions about unknown information [26, 45, 65] (*e.g.*, Hermod [45] and Orcbench [31] simulate spin loops for each execution, which assumes that the CPU utilization for each invocation is 100%).

Existing serverless benchmarks measure aspects of performance and cost [20, 31, 68, 71], but rarely focus on billing, and their function suites exhibit simplistic resource usage patterns. For example, SeBS [20] contains functions with a maximum CPU usage of one. Moreover, prior work [20, 71] considering customer cost does so only for SLIM-like billing.

5.1.2 BilliBench

We introduce a new billing-oriented benchmark, BilliBench (BB). We describe BB's essential components and how it can be used to evaluate billing models.

BB Functions: We utilize the function suites collected earlier (§2.3.3) as BB Functions. Unlike previous serverless benchmarks, our function suite exhibits varied resource usage behaviors (as shown in Figures 1, 2, and 3).

BB Trace: To realistically compare different billing models, invocation traces must contain detailed resource usage fluctuations. Our key insight is that, given summarized coarse-grained trace data from major cloud providers (such as Azure), we can supplement incomplete information with detailed fine-grained data from real-world serverless functions (*e.g.*, BB functions). By combining these two sources of information, we can generate realistic traces of function invocations.

The BB Trace includes three components: functions, invocations, and resource-usage phases. A function contains a list of invocations, and each invocation contains a list of resource-usage phases; each phase specifies a duration and the resource utilization in that interval (*i.e.*, active thread number and memory size). Each function in the Azure dataset, F_{Azure}, is represented as a function in the BB Trace; for each F_{Azure}, we randomly select a function F_{BB} from the BB function suite (shown in Table 1) to provide detailed data on resource usage.

We then translate the Azure dataset into a list of function invocations, following previous work [26, 45, 64]. After that, BB represents each invocation as a sequence of resource-usage phases: for each phase, we randomly choose the number

of active threads below F_{BB}'s maximum parallelism, ensuring the invocation's average parallelism across all phases matches that of F_{BB}; next, we randomly sample the memory size from the distribution of F_{Azure} in the Azure dataset; finally, we randomly select the runtime of the entire invocation from F_{Azure}'s runtime percentile in the Azure dataset. The number of resource-usage phases is based on F_{BB}, and the duration of each phase is the total runtime divided equally among the phases. The QoS of each invocation is based on whether the corresponding F_{BB} is interactive or batch.

Given the constructed dataset, we synthesize a function F_S with Python code for the function F_{Azure} that imitates its specific resource usage phases during invocation.

The above default BB Trace leverages CPU usage behavior and QoS of BB functions. To mitigate bias from the limited function suite, BB allows users to configure the trace parameters. Users can generate a trace with a configured average CPU utilization across all functions; BB randomly selects the active thread number for each phase to maintain the overall average. Users can also configure batch task proportions. These configurable synthetic traces offer a method for evaluating billing models across diverse workloads. In our experiments, we evaluated NPFU with the default BB Trace (§5.2) and BB Traces with varying average CPU utilization (ranging from 10% to 90%) and batch task proportions (ranging from 10% to 60%) (§5.3). Each combination of these two parameters generates a unique trace.

Knob Tuning: To compare billing models with different configuration options, we configure the functions in the BB Trace with four base limits to imitate real users. We set *cpu-cap* to the function's maximum parallelism and *mem-cap* to its maximum memory usage; *preemptible-memory* is set to false for interactive functions and true for batch functions; for interactive functions (*e.g.*, ML inference and database clients), *spot-cores* is set to zero, while for other interactive functions, *spot-cores* is set to *large_parallelism* minus *medium_parallelism* for performance variability.

BB can run with simple and complex models. Models that enforce strict memory-to-CPU ratios can round-up either *mem-cap* or *cpu-cap*. Platforms that do not differentiate between interactive and batch jobs can ignore *spot-cores* and *preemptible-memory*, assuming that all cores are reserved.

BB Metrics: We compare billing models using five metrics across providers and customers. For performance issues, we measure resource utilization and throughput for providers and job completion time for customers. Customer cost (per invocation) and provider revenue (per machine) are two other interrelated metrics. We explicitly separate billing from pricing, since pricing merely shifts value between producers and consumers (*e.g.*, reducing prices by 10% lowers revenue by 10%), while better billing models can create additional value. To compare billing models, we fix provider revenue per machine per time unit by adjusting the pricing constants (*e.g.*, M and C in Table 3); keeping provider revenue constant lets us compare models using a single metric: user cost. When models allow tenants to tune both memory and CPU, we fix the C/M ratio at 9.6 to match Google Cloud Functions [8].

Baselines	Tunable Knobs				Billing Function	
	cpu cap	spot core	mem cap	preemptible	CPU Bill (per time unit)	Memory Bill (per time unit)
SLIM			✓		mem-cap / $r \times C$	mem-cap$\times M$
SIM	✓		✓		cpu-cap$\times C$	mem-cap$\times M$
SPFU	✓		✓		avg-cpu$\times C$	avg-mem$\times M$
NPFU	✓	✓	✓	✓	resv-cores$\times C_r$ +avg-spot-cpu$\times C_s$ −avg-lent-cpu$\times C_s$	mem-cap$\times M_r \times \neg P$ +avg-mem$\times M_p \times P$ −avg-lent-mem$\times M_p$

Table 3: The tunnable knobs and billing functions of baselines. Billing functions contain the price constants that providers can adjust: C for per cpu-second, M for per GB-second. NPFU includes two CPU prices for reserved CPU (C_r) and spot CPU (C_s) (Similarly for M_r and M_p) and resv-cores = cpu-cap − spot-cores. r indicates the fixed memory-to-CPU ratio in SLIM. P indicates whether the user set preemptible-memory or not.

Billing Models: We have implemented the following mainstream billing models in Leopard, as summarized in Table 3. *SLIM:* AWS Lambda, GCP, and Azure's premium plan use SLIM-like billing models, where users specify a memory limit for each function, and CPU is proportionally allocated; these limits serve as both minimal and maximal thresholds. *SIM:* SLIM without Assumption L provides independent knobs for memory and CPU limits. *SPFU*[1]: Strict PFU has no knobs, since cost is a function of usage and not knob settings; however, a platform using SPFU may still expose resource limit settings (*e.g.*, Azure's consumption plan and Cloudflare use mostly fixed resource limits), and so we optimistically assume customers set these limits as restrictively as possible, despite having little billing incentive to do so. *NPFU:* NPFU contains four knobs (*cpu-cap*, *spot-cores*, *mem-cap*, and *preemptible-mem*) and supports paying more or less when resources are borrowed or lent, respectively. NPFU aims to closely approximate SPFU while remaining profitable in practice.

5.1.3 Experiment Setup

We conduct experiments on machines with two Intel Xeon Silvers, each with 10 physical cores. We disable hyperthreading to eliminate hardware-level interference, allowing us to focus on CPU isolation mechanisms in the kernel. We use Ubuntu 22.04 with the latest Linux kernel v6.7-rc2 using cgroups V1.

We build support for each of the above billing models in Leopard. We use BB Traces based on a random sample of 1,000 functions in the BB dataset. The client initiates a task to send the invocation at the designated timestamp and wait for the response. We use one load-balancing node to route invocations to one of eight worker nodes. For each worker node, we assigned 16 CPUs and 8 GB of DRAM to serve invocations and 1 CPU to handle worker logic, including admission control and communication with the controller.

[1] In §2, we claim that SPFU is an idealized billing model, but it's not very profitable in practice as it usually provisions resources that may go unbilled.

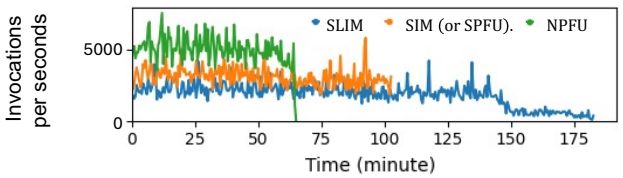


Figure 7: The throughput vary based on different billing knobs.

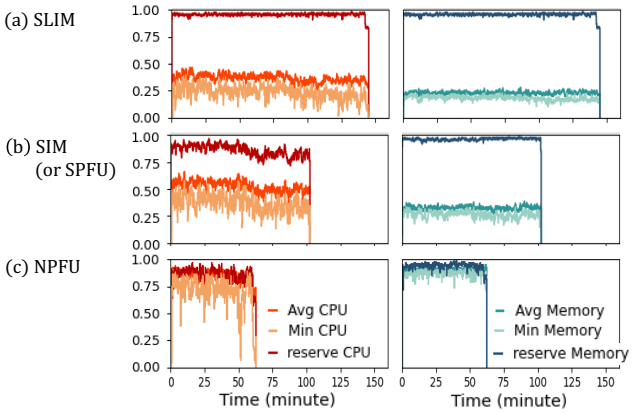


Figure 8: The utilization vary based on different billing knobs.

5.2 Evaluation of NPFU

We show that NPFU benefits both providers and users: providers obtain greater efficiency while users pay lower costs and still meet their QoS needs.

5.2.1 Provider Side

Figure 7 shows the benefits of NPFU for providers: the throughput for SLIM, SIM, and NPFU billing models in the first hour of invocations from the BB Trace. We observe that decoupling the linear memory-CPU relationship (*i.e.*, going from SLIM to SIM) leads to a 1.3x increase in throughput. Switching from SIM to NPFU provides an additional 1.6x improvement because Leopard NPFU can admit invocations when all resources are reserved but happen to be idle.

Figure 8 shows the average and minimum resource utilization in the cluster, explaining the higher throughput of NPFU. With SLIM, over 50% of CPU and 75% of memory are wasted, while SIM improves resource utilization by decoupling CPU and memory, and NPFU further increases memory utilization to 90% and CPU utilization to 80%, thanks to spot cores and preemptible memory. We observe that all billing models maintain high reservation rates: SLIM achieves near-100% CPU and memory reservation, while SIM and NPFU have lower CPU reservations due to CPU-memory decoupling. In summary, Leopard's load balancer and admission control ensure full resource rental under high load. These benefits come from the more expressive knobs of NPFU, allowing users to specify which portions of allocated resources can be borrowed without compromising their objectives. In contrast, SLIM lacks such knobs, as it makes four assumptions about resource allocation, preventing resource schedulers from sharing unused resources across invocations and leading to low utilization.

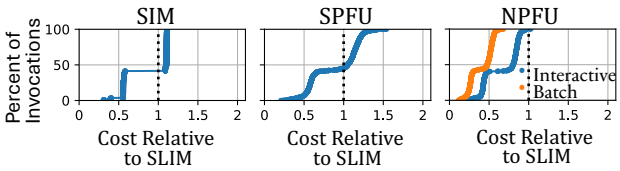


Figure 9: User cost relative to SLIM with different billing models.

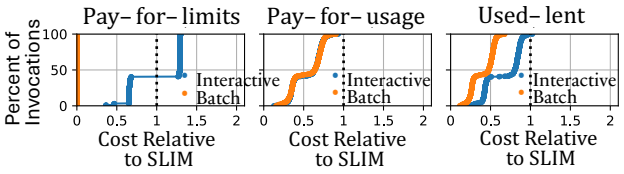


Figure 10: User cost of NPFU with different billing functions.

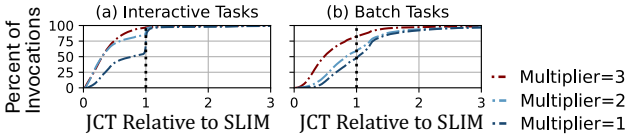


Figure 11: CDF of Job Completion Time (JCT) of invocations when running with NPFU (relative to that when running with SLIM).

5.2.2 User Side

For users, we examine both cost and performance.

User cost: To perform a fair comparison for each billing model, we adjust the unit prices of CPU and memory to ensure that the provider revenue remains the same as the SLIM. Figure 9 shows the CDF of invocation cost relative to the cost of running with SLIM. With SIM, approximately 50% of invocations save money since they no longer over-allocate CPU to meet the memory requirements of the linear model; however, other functions become slightly more expensive. For SPFU, maintaining provider revenue necessitates increasing SPFU's C and M price parameters by 1.6x; as a result, some functions cost more than 50% because SPFU distributes the cost of resource wastage across all users. Most importantly, NPFU reduces the cost of nearly every invocation, often significantly: about 40% of functions cut costs by at least half. Batch tasks save more than interactive tasks, as they pay lower rates for spare resources that interactive instances reserved. In total, the costs for NPFU clients are 66% (for interactive functions) and 41% (for batch functions) of those under SLIM.

In Figure 10, we examine the benefits of used-lent billing by comparing NPFU with pay-for-limits (like SLIM) and pay-for-usage (like SPFU). Pay-for-limits lets batch tasks run at no cost, as they did not require any reserved resources, making interactive tasks more expensive. Pay-for-usage charges interactive and batch jobs similarly and lacks incentives for users to relax their QoS requirements; it also does not charge for limits, giving users no incentive to set realistic restrictions.

Job Completion Time: We now examine whether interactive functions meet their QoS. To simulate different load levels, we repeat each invocation a *multiplier* number of times. Figure 11(a) compares job completion time (JCT), including queuing and execution time, with NPFU versus SLIM. Under

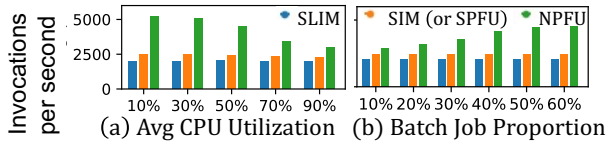


Figure 12: The throughput as the workload characteristics change.

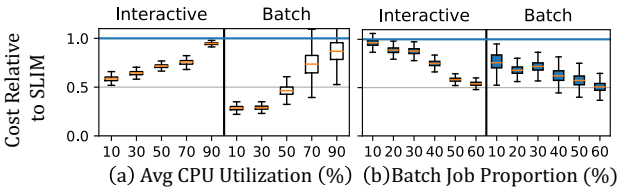


Figure 13: The cost of NPFU as the workload characteristics change.

high system load, when most invocations experience queuing, NPFU significantly reduces JCT for nearly all interactive functions. As the load decreases, the JCT for NPFU and SLIM becomes similar. A comparison with SIM (or SPFU) produced similar results. Batch tasks do not have strict latency requirements. As shown in Figure 11(b), batch tasks may take up to three times longer with NPFU compared to SLIM; however, some batch tasks can achieve shorter JCT as NPFU allows them to run when resources are not fully available.

5.3 Robustness to Different Workloads

We now analyze billing models for a range of BB Traces; we adjust CPU utilization and the proportion of batch tasks.

CPU Distributions: In a BB Trace, each invocation derives its parallelism from BB functions. In Figure 12(a), we vary the CPU utilization for each function following a normalized distribution with a given average. As CPU utilization increases for each invocation, NPFU delivers reduced throughput for the provider and smaller savings for customers, as Leopard admits extra batch tasks only when CPU utilization is below a threshold. Other billing models have unchanged throughput, as they admit tasks based on reserved resources, not usage. Still, the throughput of NPFU always remains higher than that of the other models. Figure 13(a) shows the user cost with varying CPU utilization; when CPU utilization exceeds 70%, a small number of batch invocations become slightly more expensive for customers with NPFU.

Batch Task Proportions: Previously, we assigned functions as interactive or batch based on BB functions. In Figure 12(b) we analyze how Leopard's performance changes with different batch task proportions. NPFU achieves higher throughput for the provider as the batch task proportion increases; for other models, the throughput stays constant. Figure 13(b) shows that NPFU offers greater financial benefits for users when there are more batch tasks; with more batch tasks, they can better use idle but reserved resources and boost throughput. Compared to SLIM, NPFU may not reduce the costs for a small portion of interactive tasks, as idle resources remain underutilized without sufficient batch tasks.

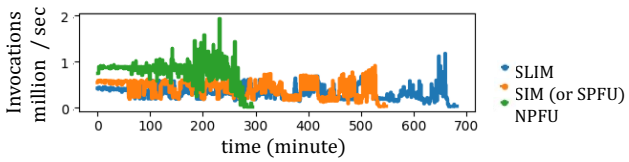


Figure 14: The provider throughput running large-scale BB Trace on different billing knobs.

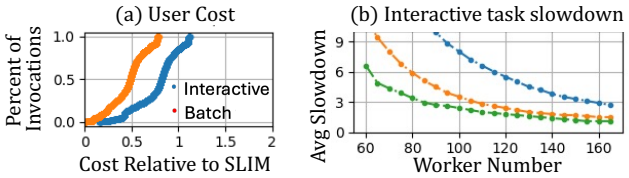


Figure 15: The NPFU's (a) user cost (b) JCT of interactive jobs running large-scale BB Trace.

In conclusion, NPFU outperforms SLIM for every point we explored in the space of CPU utilization and batch-job proportion. The benefits are greatest when interactive jobs have low CPU utilization relative to resource caps and there are many batch jobs to take advantage of those leftover resources.

5.4 Large-Cluster Simulations

To study a larger cluster, we build a simulator to evaluate different billing models on a cluster of 160 workers and use the entire first hour of the complete BB Trace (instead of only a random sample of 1,000 functions). Figure 14 shows that the simulated results on a 160 workers obtain the same benefits as a real 8-worker cluster (Figure 7).

Figure 15(a) shows that improvements in user cost with NPFU relative to SLIM are similar to that shown for small-scale experiments (Figure 9); however, about 5% of interactive functions are now up to 15% more expensive than with SLIM. Given that NPFU differentiates between interactive and batch functions, it is natural that some revenue collection shifts from batch to interactive. Figure 15(b) shows that the QoS of interactive functions is still satisfied under NPFU as the number of workers changes, with consistently lower slowdown than in other billing models.

5.5 Evaluation of Leopard Components

CPU Reservations: We evaluate whether the cgroup interface cpu.resv_cpuset and modifications to CFS of Leopard efficiently support CPU reservations. We examine three settings: (1) SLIM with CPU pinning: each sandbox is assigned a fixed CPU set and exclusively uses its allocated CPUs. (2) NPFU with weighted sharing: all CPUs are shared among active sandboxes, where interactive sandboxes with reserved CPUs have a weight of $1024 \times resv_cpu$ and non-interactive sandboxes have a weight of 1. (3) NPFU with reserved CPU set: sandboxes with reserved CPUs are assigned to a reserved CPU set; sandboxes can access their reserved CPUs when needed but still lend idled CPUs.

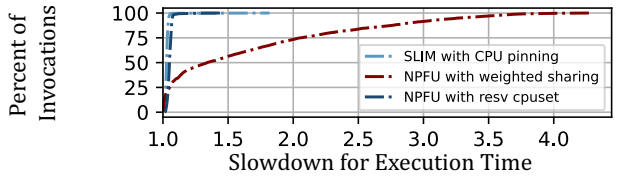


Figure 16: The CDF for the invocation execution time slowdown relative to running without contention, excluding queuing delay.

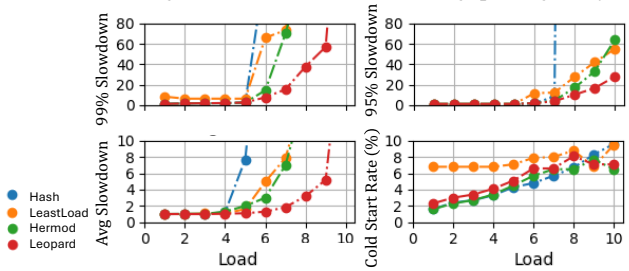


Figure 17: The slowdown and cold start rate when implementing different load balancers on Leopard.

Figure 16 shows the CDF of execution time slowdowns for interactive invocations in the BB Trace. Execution time excludes queue delay, so the slowdown is primarily due to CPU contention among co-located invocations. We observe that CPU pinning cannot support spot-cores; fair-share scheduling cannot correctly provide performance isolation for interactive jobs; and Leopard's reserved cpuset enables the spot-core while maintaining performance isolation for interactive jobs.

Load Balancer: With NPFU, we implement the following load balancers on Leopard: (i) Consistent Hashing scheduler from OpenWhisk [4], (ii) Least-Loaded Scheduler as in Kogias *et al.* [49], and (iii) Hermod [45] scheduler, which uses a greedy approach under low load and the least-loaded scheduler during high load. In Figure 17, we present the slowdown in job completion time (including queue delay) for all interactive functions in Leopard (with NPFU enabled) under different load balancers. A few key observations: First, under low load, the 99% slowdown of Least-Loaded balancer is higher compared to other load balancers since it does not account for locality (the cold start rate is depicted in Figure 17). Second, the performance of the Consistent Hashing deteriorates significantly as the load increases, as it is load-unaware. Finally, Leopard performs as well as, or better than, these common serverless schedulers across all load conditions.

6 Related Work

Improving Serverless Utilization: Existing research has explored ways to increase the resource utilization of serverless platforms. Owl [67], Golgi [50], and Jiagu [53] stress overcommitment of function instances. However, these approaches still use SLIM billing models. Overcommitment breaks the promise made by the SLIM. They rely on predicting user performance to avoid breaking the QoS of user requests, which is difficult for real workloads [17]. Other works [58,75] search for a near-optimal configuration for functions under SLIM or SIM. However, our evaluation shows that even if users choose precise resource limits, the cluster still suffers from low resource utilization. This problem arises because actual usage fluctuates both within individual invocations and across different invocations. We introduce a new perspective by formalizing existing billing models and proposing a novel model aimed at improving utilization.

Resource Harvesting: Cloud services (*e.g.*, EC2 [12]) offer on-demand and spot instances, while Harvest VMs [14] improve cost-efficiency by adjusting VM cores over its lifetime. Harvest VMs adjust CPU affinity with interprocessor interrupts, introducing high overhead (~0.1 milliseconds). These approaches are cost-effective, but VM size changes are infrequent compared to serverless workloads; supporting QoS-aware sandboxes on serverless platforms presents additional challenges. As shown in Figure 1, CPU demand fluctuates frequently, and adjusting CPU affinity with every change would incur unacceptable overhead. BigQuery [7] offers capacity and on-demand pricing, but the resources are static in a sandbox in both pricing models. Running interactive tasks alongside batch tasks with efficient resource schedulers [19,61] has been explored in multi-tenant datacenters. However, serverless platforms do not effectively support co-location, as their billing models assume that all functions are interactive.

CPU Reservation: The user-space schedulers [24,38,44,57,60] support CPU reservation by adjusting CPU affinity over time, using a centralized CPU for allocation and ensuring sub-microsecond tail latency. However, they are not well-suited for serverless environments. Their simple load balancing methods inside a task group (*e.g.*, work stealing) are difficult to maintain fairness within a group. Additionally, some require users to modify their code for process creation.

Serverless Trace Analysis: Many studies have offered comprehensive overviews of the serverless computing [33,43,51,56,69]. For example, Microsoft researchers [64] characterize serverless workloads in their production platform, revealing a clear gap between maximum and average memory usage, indicating dynamic memory usage. Similarly, FirePlace [17] shows the fluctuating CPU and memory usage of 200K micro VMs on AWS Lambda.

7 Conclusion

We introduced the NPFU billing model and built Leopard to support NPFU with kernel-level and cluster-level changes. NPFU improves throughput by more than 2x, and reduces customer costs while maintaining provider profitability. Our work demonstrates that billing models should be considered not as an afterthought, but as a central part of system design.

8 Acknowledgment

We thank Dong Du, the anonymous reviewers, and students in ADSL for their feedback. This work was supported by the NSF under the award number CNS-2402859 and the taxpayers of Wisconsin and the USA. Opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and may not reflect the views of these institutions.

9 References

[1] Kubernetes. http://kubernetes.io, August 2014.

[2] Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, August 2014.

[3] Docker Project. https://www.docker.io/, 2015.

[4] IBM OpenWhisk. https://developer.ibm.com/openwhisk/, May 2016.

[5] Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/, May 2016.

[6] Common Serverless Applications Scenarios. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/scenarios.html, May 2023.

[7] Google BigQuery. https://cloud.google.com/bigquery/docs/introduction, May 2023.

[8] Google Cloud Functions pricing. https://cloud.google.com/functions/pricing, May 2023.

[9] KNIX Serverless. https://github.com/knix-microfunctions/knix/, May 2023.

[10] Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, May 2023.

[11] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[12] Amazon. Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing/, November 2023.

[13] Amazon. AWS Lambda. https://aws.amazon.com/lambda/, November 2023.

[14] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 735–751. USENIX Association, November 2020.

[15] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A Serverless Video Processing Framework. In Proceedings of the ACM Symposium on Cloud Computing, page 263–274, 2018.

[16] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books, 1.10 edition, November 2023.

[17] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan (Murali) Narayanaswamy. FirePlace: Placing FireCracker virtual machines with hindsight imitation. In MLSys 2021, NeurIPS 2020 Workshop on Machine Learning for Systems, 2020.

[18] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, page 1–15, 2020.

[19] Shuang Chen, Christina Delimitrou, and José F. Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In Proceedings of the 22nd International Middleware Conference, page 64–78, 2021.

[21] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 475–488, Renton, WA, July 2019. USENIX Association.

[22] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 363–376, Boston, MA, March 2017. USENIX Association.

[23] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 363–376, Boston, MA, March 2017. USENIX Association.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 281–297. USENIX Association, November 2020.

[25] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. SFS: Smart OS Scheduling for Serverless Functions. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '22. IEEE Press, 2022.

[26] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 3–18, 2019.

[28] Google. Google Cloud Functions. https://cloud.google.com/functions, November 2023.

[29] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. SIGCOMM Comput. Commun. Rev., 44(4):455–466, aug 2014.

[30] Zhiyuan Guo, Zachary Blanco, Junda Chen, Jinmou Li, Zerui Wei, Bili Dong, Ishaan Pota, Mohammad Shahrad, Harry Xu, and Yiying Zhang. Zenix: Efficient Execution of Bulky Serverless Applications, 2024.

[31] Ryan Hancock, Sreeharsha Udayashankar, Ali José Mashtizadeh, and Samer Al-Kiswany. OrcBench: A Representative Serverless Benchmark. In 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), pages 103–108, 2022.

[32] J.L. Hellerstein. Achieving service rate objectives with decay usage scheduling. IEEE Transactions on Software Engineering, 19(8):813–825, 1993.

[33] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back, 2018.

[34] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openLambda. In Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing, HotCloud'16, page 33–39, USA, 2016. USENIX Association.

[35] G. J. Henry. The UNIX system: The fair share scheduler. ATT Bell Laboratories Technical Journal, 63(8):1845–1857, 1984.

[36] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24, page 1039–1053, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Al Amjad Tawfiq Isstaif and Richard Mortier. Towards Latency-Aware Linux Scheduling for Serverless Workloads. In Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies, SESAME '23, page 19–26, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23, page 466–481, 2023.

[39] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 152–166, 2021.

[40] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards Demystifying Serverless Machine Learning Training. In Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 406–419, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, 2019.

[44] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for second-scale Tail Latency. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 345–360, Boston, MA, February 2019. USENIX Association.

[45] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In Proceedings of the 13th Symposium on Cloud Computing, SoCC '22, page 289–305, New York, NY, USA, 2022. Association for Computing Machinery.

[46] J. Kay and P. Lauder. A fair share scheduler. Commun. ACM, 31(1):44–55, January 1988.

[47] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[49] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 863–880, Renton, WA, July 2019. USENIX Association.

[50] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing. page 32–47, 2023.

[51] Liang Wang and Mengyuan Li and Yinqian Zhang and Thomas Ristenpart and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 133–146, 2018.

[52] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The Gap Between Serverless Research and Real-world Systems. In Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC '23, page 475–485, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 1–17, Santa Clara, CA, July 2024. USENIX Association.

[54] Renato Losio. State of Serverless 2023 Report Suggests Increasing Serverless Adoption. https://www.infoq.com/news/2023/09/state-serverless-report/, September 2023.

[55] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 303–320, 2022.

[56] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions. ACM Comput. Surv., 54(11s), sep 2022.

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

[58] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. Parrotfish: Parametric Regression for Optimizing Serverless Functions. In Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC '23, page 177–192, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In USENIX Annual Technical Conference (USENIX ATC '18), Boston, MA, 2018.

[60] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 361–378, Boston, MA, February 2019. USENIX Association.

[61] Tirthak Patel and Devesh Tiwari. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 193–206, 2020.

[62] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 193–206, Boston, MA, February 2019. USENIX Association.

[63] Sambhav Satija, Chenhao Ye, Ranjitha Kosgi, Aditya Jain, Romit Kankaria, Yiwei Chen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kiran Srinivasan. Cloudscape: A Study of Storage Services in Modern Cloud Architectures. In 23rd USENIX Conference on File and Storage Technologies (FAST 25), pages 103–121, Santa Clara, CA, February 2025. USENIX Association.

[64] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference, 2020.

[65] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A Scalable Low-Latency Serverless Platform. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Prasoon Sinha, Kostis Kaffes, and Neeraja J. Yadwadkar. Online Learning for Right-Sizing Serverless Functions. In Architecture and System Support for Transformer Models (ASSYST @ISCA 2023), 2023.

[67] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud. In Proceedings of the 13th Symposium on Cloud Computing, page 78–93. Association for Computing Machinery, 2022.

[68] Erwin van Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, and Alexandru Iosup. Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark. In Companion of the ACM/SPEC International Conference on Performance Engineering, page 26–31, 2020.

[69] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 443–457, 2021.

[70] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 1489–1504, Boston, MA, April 2023. USENIX Association.

[71] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing Serverless Platforms with ServerlessBench. In Proceedings of the ACM Symposium on Cloud Computing, 2020.

[72] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 1187–1204. USENIX Association, November 2020.

[73] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 653–669. USENIX Association, April 2021.

[74] Zili Zhang, Chao Jin, and Xin Jin. Jolteon: Unleashing the Promise of Serverless for Serverless Workflows. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 167–183, Santa Clara, CA, April 2024. USENIX Association.

[75] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, page 1–14, New York, NY, USA, 2022. Association for Computing Machinery.

GRANNY: Granular Management of Compute-Intensive Applications in the Cloud

Carlos Segarra[†] Simon Shillaker[†] Guo Li[†] Eleftheria Mappoura[†]
Rodrigo Bruno[⋆] Lluís Vilanova[†] Peter Pietzuch[†]
[†]*Imperial College London* [⋆]*INESC-ID, Instituto Superior Técnico, University of Lisbon*

Abstract

Parallel applications are typically implemented using multi-threading (with shared memory, e.g., OpenMP) or multi-processing (with message passing, e.g., MPI). While it seems attractive to deploy such applications in cloud virtual machines (VMs), existing cloud schedulers fail to manage such applications efficiently: they cannot scale multi-threaded applications dynamically when more CPU cores in a VM become available, and they cause fragmentation over time due to the static allocation of multi-process applications to VMs.

We describe GRANNY, a new distributed runtime that enables the fine-granular management of multi-threaded/process applications in cloud environments. GRANNY supports the vertical scaling of multi-threaded applications within a VM and the horizontal migration of multi-process applications between VMs. GRANNY achieves both through a single WebAssembly-based execution abstraction: *Granules* can execute application code with thread or process semantics and allow for efficient snapshotting. GRANNY scales up applications by adding more Granules, and de-fragments applications by migrating Granules between VMs. In both cases, it launches new Granules from snapshots efficiently. We evaluate GRANNY with dynamic scheduling policies and show that, compared to current schedulers, it reduces the makespan for OpenMP workloads by up to 60% and the fragmentation for MPI workloads by up to 25%.

1 Introduction

Compute-intensive parallel applications are common in many domains including machine learning [23], weather forecasting [44], hydrodynamics [35], genomics [5], simulation, and modeling [43]. These applications exploit massive parallelism and are built using multi-threaded or multi-process programming models, such as OpenMP [36] and MPI [33] that utilize the CPU cores within a node and across nodes.

To satisfy their resource needs, such applications are increasingly deployed on cloud-hosted clusters with a high total virtual CPU (vCPU) core count made up of virtual machines (VMs). These cloud clusters are managed by cloud resource schedulers, e.g., Slurm [48] or Azure/AWS/Google Batch [2, 16, 27], which monitor a queue of submitted applications and allocate them to VMs and vCPUs.

Existing cloud resource schedulers, however, cannot change an application's resource allocation after it has started executing. This lack of *elasticity* prevents multi-threaded applications from leveraging new cloud resources as they become available, e.g., when other applications have released vCPUs. This reduces utilization: our experiments show that, when deploying multi-threaded (OpenMP) applications, Azure Batch and Slurm consistently leave 60% and 40% of vCPUs idle, respectively, even when there are pending applications to be scheduled (§6.3). No matter how effective a resource manager's bin-packing approach is, it cannot completely avoid idle resources.

For multi-process applications, high utilization can be achieved by allocating resources at a fine granularity (e.g., allocating vCPUs to applications as soon as they become available), but this leads to *resource fragmentation*. If multi-process applications become fragmented across many VMs, it increases their network communication, thus reducing overall application performance.

Cloud resource schedulers handle this tension between utilisation and compute/data locality differently: when executing distributed multi-process applications, the Azure Batch [27] scheduler allocates resources at a VM granularity. This achieves good locality at the cost of resource utilization, because idle vCPUs in VMs cannot be used to execute other applications; in contrast, Slurm [48], another popular scheduler, allocates resources at vCPU granularity. It exhibits high utilization, but incurs fragmentation.

While high utilization is desirable for cloud providers, low fragmentation improves application performance. We observe that an ideal cloud resource scheduler for parallel applications must navigate this trade-off by *elastically scaling* and *migrating* applications at a fine granularity (i.e., individual vCPUs) at runtime. Existing runtimes for multi-threaded and multi-process applications in cloud environments, however, do not support such fine-granular management.

We describe **GRANNY**, a distributed runtime for executing unmodified multi-threaded/multi-processing (OpenMP/MPI) applications in cloud environments, which supports *vertical scaling* of multi-threaded application and *horizontal migration* of distributed multi-process applications. GRANNY achieves both with the help of a new execution abstraction called *Granules*. It makes the following contributions:

(1) Granules for thread/process execution. GRANNY executes each application as a set of Granules (§3.2). Each Granule contains application code with a single execution thread, capable of executing with thread (i.e., for multi-threaded applications) or process (i.e., for multi-process applications) semantics. A Granule executes a sandboxed WebAssembly [17] module, so that all Granules on a VM execute safely within a single host process. This allows Granules to implement an efficient snapshotting operation, share memory directly by mapping pages, and exchange messages with low overhead.

A Granule's execution state can be captured completely as part of a *snapshot* (§3.3). A Granule's snapshot contains its WebAssembly linear memory, with its static data sections, stack and heap, as well as other execution state, such as stack pointers, function tables, messaging queues, and file descriptors. Since their execution state is self-contained, Granules, unlike processes or containers [60], can be snapshotted and migrated robustly and with low performance impact.

(2) Granule elasticity and migration. GRANNY leverages Granules to perform management actions: it can (a) spawn a new Granule with thread semantics to scale a multi-threaded application (§4.2); or (b) change the distribution of a multi-process application by migrating the snapshot of a Granule with process semantics to a different VM (§4.3).

When performing the above management actions, GRANNY must not break the consistency of a multi-threaded/process application. GRANNY therefore only controls Granules when their execution reaches well-defined *control points* (§4.1), such as certain system calls or OpenMP/MPI calls. At these control points, a Granule is guaranteed to have a consistent state with respect to its shared data and messages, and it is safe to spawn new Granules, or snapshot and migrate them.

(3) Granule-aware scheduling policies. Using Granule's support for vertical scaling and horizontal migration, GRANNY implements dynamic scheduling policies for fine-granular management of compute-intensive applications in the cloud (§5). The policies enable (i) multi-process (MPI) applications to migrate processes between VMs to reduce inter-VM communication and thus improve application performance (§5.1); (ii) multi-threaded (OpenMP) applications to launch extra threads, increasing their parallelism when further vCPUs become available at runtime on a VM (§5.2); and (iii) checkpoint applications on spot VMs [7] before they are evicted and migrate them to replacement VMs (§5.3).

In our experiments on a 32-VM cluster, we execute a workload of existing OpenMP/MPI applications based on a 100 job trace. We compare GRANNY's execution to that with Azure Batch [27] and Slurm [48]. Azure Batch and Slurm use the OpenMPI [39] and LLVM OpenMP (`libomp`) [26] runtimes to execute MPI and OpenMP jobs, respectively.

We show that, by defragmenting multi-process applications using horizontal migration, GRANNY reduces end-to-end execution time (makespan) by 20% and fragmentation by 25% (§6.2). By elastically scaling multi-threaded applications to use idle CPU cores, GRANNY reduces makespan by 60% and the tail job completion time by 50% (§6.3). Finally, when using spot VMs, GRANNY reduces makespan by 50% (§6.4).

2 Compute-Intensive Workloads in Cloud

Next, we discuss compute-intensive parallel applications (§2.1), their runtime support (§2.2), describe associated schedulers (§2.3), and explain why existing schedulers fail to manage these applications efficiently in cloud settings (§2.4).

2.1 Compute-intensive applications

Compute-intensive applications include large-scale data analytics [61], video processing [4], and deep learning training [52], and also typical high performance computing (HPC) workloads, such as fluid dynamics [35], molecular simulation [47], and weather forecasting [44]. These applications require plentiful hardware resources, because they parallelize computation by distributing it across many CPU cores, both within nodes and across nodes.

To handle large problem sizes without increasing execution time or exhausting memory, compute-intensive applications make use of vertical *scale-up* and horizontal *scale-out* patterns. These are typically implemented using multi-threading (with shared memory among threads) and/or multi-processing (with distributed message passing between processes located on different nodes), as offered by programming models such as OpenMP [36] and MPI [33]. In this work, we focus on OpenMP and MPI, as representative examples of APIs and runtimes for parallel applications. They also benefit from long-standing popularity and widespread adoption.

2.2 Runtimes for multi-threading/processing

Multi-threaded/process applications typically require *runtime* support. Multi-threaded runtimes such as LLVM's OpenMP runtime (`libomp`) [26] manage threads, coordinate access to shared memory, and provide synchronization primitives; multi-processing runtimes such as OpenMPI [39] provide APIs for inter-node message passing, message synchronization, and data partitioning.

Internally, these runtimes use OS primitives to implement their APIs. In Linux, most OpenMP implementations use POSIX threads and standard synchronization primitives, such as mutexes and locks. MPI implementations use the OS kernel's TCP/IP stack to send and receive messages between processes reliably. Runtimes typically do not restrict other OS system calls e.g., for file system access. In this case, however,

it becomes the responsibility of the developer to access the underlying resources in a race-free manner, as the runtime's coordination and synchronization guarantees do not apply.

Depending on whether an application uses multi-threading or multi-processing, application state is handled differently. For multi-threaded applications, application state resides in the process' address space, but may be divided in different thread's stacks and local storage (TLS). The state associated with synchronization primitives, e.g., mutexes and locks, resides in the OS kernel. For multi-process application, the state is partitioned across all processes. In addition, the state of the network stack is spread across all VMs involved in the distributed computation.

2.3 Resource scheduling for applications

Compute-intensive applications are deployed on large shared clusters (either on-premise or in the cloud) with high CPU core and node counts. Users submit applications as jobs to a work queue managed by a resource *scheduler*. The scheduler allocates applications to compute resources (i.e., CPU cores) according to an application's demand for parallelism. Multi-threaded/process applications specify their resource needs at deployment time, e.g., using `mpirun`'s `np` flag [40], or the `OMP_NUM_THREADS` environment variable [37].

Cloud providers use a range of resource schedulers to execute compute-intensive parallel applications, such as AWS Batch [2], Azure Batch [27], and Google Batch [16]. These schedulers control a pool of cloud VMs that can be scaled up or down on-demand. They schedule jobs once sufficient VMs are available, and idle vCPU cores in VMs cannot be allocated to other pending jobs.

In addition, some cloud providers also support general-purpose schedulers such as Slurm [48], KubeBatch [21], or Volcano [53], which are often used for on-premise clusters. These schedulers support more fine-grained policies in which they allocate applications to individual vCPU cores.

2.4 Challenges in cloud scheduling

While cloud providers strive for high utilization, cloud users want to combine high parallelism with locality, as this reduces avoidable inter-VM communication. Cloud schedulers thus try to optimize both for resource utilization and compute/data locality [20, 49], but they struggle to achieve this goal: the allocation of a multi-threaded application to a VM determines its parallelism based on the number of available vCPUs. If the scheduler cannot perfectly bin-pack multi-threaded applications to VMs, vCPUs remain idle, decreasing utilization.

Similarly, a scheduler must allocate the processes of a distributed multi-process application to sufficiently many VMs that offer the required total number of vCPUs. This potentially reduces locality, as it may spread the allocation across multiple VMs, depending on their vCPU availability.

Ideally, the scheduler should be able to *change* this initial allocation of a parallel application at the granularity of individual vCPUs. In particular, it should (a) add more threads to a multi-threaded application when vCPUs become available, increasing utilization; and (b) move processes to VMs that already contains other processes of the same application, reducing fragmentation and improving locality.

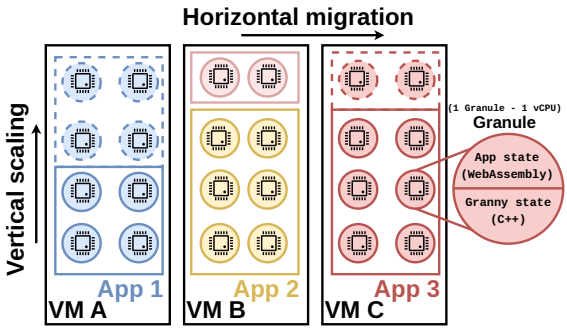


Figure 1: GRANNY executes applications as sets of Granules (GRANNY supports elasticity by vertically scaling multi-threaded applications, adding more Granules, or horizontally migrating Granules of multi-process applications to consolidate them across VMs.)

Such fine granular resource management, however, is not possible due to the limitations of today's runtimes for multi-threaded/process applications: elastically scaling threads requires runtime support, otherwise it may violate consistency with concurrent memory accesses by threads. For example, threads may have shared variables pending synchronization; similarly, migrating processes between VMs requires support for *process-level checkpointing*, e.g., CRIU [11], which incurs significant overheads [18, 60]. Without runtime support, it also cannot ensure consistent checkpoints across all distributed processes when messages are in-flight. Applying existing migration techniques [54] thus would require changes to applications, runtimes, and OS kernels.

3 GRANNY Design

We describe the design of GRANNY, a distributed runtime for multi-threaded/processing applications (§3.1). At its core lies the abstraction of a *Granule* (§3.2), which is the building block for the execution of both thread- and process-based applications. The state of Granules can be captured efficiently using *snapshots* (§3.3), thus enabling elasticity and migration.

3.1 Overview

GRANNY executes multi-threaded and multi-process applications as a set of Granules (shown as circles in Fig. 1). A Granule represents a single thread of execution and has its own mappings for code and data: a multi-threaded application (OpenMP) is therefore a set of Granules with shared-memory code and data mappings; a multi-process application (MPI) is a set of Granules with non-shared data mappings.

Each Granule runs in a single WebAssembly [17] module with its application code and executes on a vCPU in a VM. The GRANNY runtime decouples Granule state from OS ker-

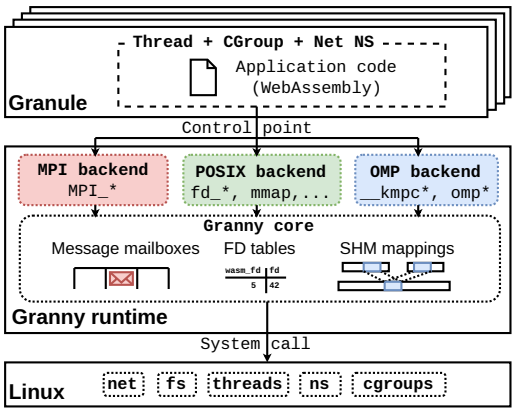


Figure 2: GRANNY architecture (Application code, cross-compiled to WebAssembly, can make calls, via control points, to different backends. The GRANNY core maintains relevant runtime system state and defers to OS system calls when necessary.)

nel state, which makes the snapshotting of Granules simpler and faster than traditional process checkpoints [11]. It also allows GRANNY to execute multiple Granules within the same process (shown as different colors in Fig. 1), because WebAssembly modules are isolated from each other and the runtime. Each VM runs a single instance of the GRANNY runtime, and there is also one cluster-wide GRANNY scheduler.

GRANNY uses the Granule abstraction (§3.2) and snapshots (§3.3) to implement management actions (§4): it can spawn Granules with thread semantics to vertically scale multi-threaded applications (§4.2), and it can horizontally migrate the snapshot of a Granule with process semantics (§4.3). The GRANNY scheduler uses these management actions to implement dynamic scheduling policies (§5): as shown in Fig. 1, it can speed up App 1 with the newly released vCPUs of VM A, improving utilization, or consolidate App 3 to VM C, improving locality.

3.2 Granule abstraction

Fig. 2 shows the GRANNY architecture. A Granule executes as an OS thread inside the VM in which it is spawned. Application code is deployed in an isolated WebAssembly module that only has external access to pre-defined OpenMP, MPI, and POSIX operations implemented by the GRANNY runtime (the various backends in Fig. 2). The GRANNY backends are built on top of a shared GRANNY runtime core that keeps the per-Granule state and decouples backends from the OS.

Applications must be cross-compiled to WebAssembly [17], a memory-safe and platform-independent binary instruction format that supports a wide range of programming languages. The use of WebAssembly allows Granules to execute side-by-side within one virtual address space, together with the GRANNY runtime, while enforcing memory safety [19] and reducing interaction with the privileged OS kernel. In addition to the file system and network operations necessary for multi-process applications, GRANNY only relies on the OS kernel to schedule threads and guarantee resource (CPU and network) fairness.

WebAssembly code cannot, in general, interact with its host environment. In GRANNY, application code can use *control points* to interact with the runtime (§4.1). A control point is triggered by a call to one of the supported APIs, POSIX, MPI, and OpenMP, implemented in the backends. Control points come at no cost for developers and at little cost for the runtime: they are injected by leaving the corresponding API symbol (e.g., `MPI_Barrier`) as undefined during cross-compilation and marked as a function import [13]. The symbol is resolved at runtime and triggers a WebAssembly context switch executed in tenths of cycles, similar to a function call [34].
The GRANNY runtime makes, whenever necessary, system calls to the underlying OS, e.g., to send cross-VM messages or write to a file descriptor. It records these interactions to ensure that Granule state can be encapsulated in a snapshot.

3.3 Granule implementation

Control points. Our GRANNY prototype has three backends:

The **MPI** backend implements the standard `MPI_*` APIs (e.g., `MPI_Reduce` [33]) and provides reliable Granule-to-Granule messaging. It uses in-memory message mailboxes for message reception, and the host network stack for cross-VM messaging. The GRANNY runtime maintains consistent Granule addressing tables across Granule migrations.

The **OpenMP** (OMP) backend implements the interface exposed by LLVM's OpenMP runtime (`libomp`) after OpenMP pragmas have been expanded (e.g., `__kmpc_fork_call` [26]). For correct and safe OpenMP execution, the GRANNY runtime must carefully manage the shared and private memory regions of different Granules (see §3.3).

The **POSIX** backend implements WebAssembly's standard system interface (WASI) [55]. The use of WASI simplifies the cross-compilation of large codebases, because it enables the static linking of applications with WASI-aware libraries, such as `wasi-libc` [57]. The backend only implements the symbols needed by compute-intensive applications, which are mostly filesystem APIs. To maintain a consistent Granule state and facilitate snapshots and migration, the GRANNY runtime maps the file descriptors used internally in WebAssembly code, to the OS file descriptors used in practice.

Memory layout. Due to WebAssembly's linear memory model [17], Granules have a simple memory layout, shown in Fig. 3, which facilitates spawning and snapshotting. A Granule occupies a contiguous region of virtual memory with the code, data, a stack, and a heap. Spatial isolation comes from a combination of WebAssembly's memory safety and guard pages, and is it enforced by the WebAssembly runtime [10].

The main difference between a Granule executing with process semantics and with thread semantics is their memory layout (red and blue Granules in Fig. 3, respectively): Granules

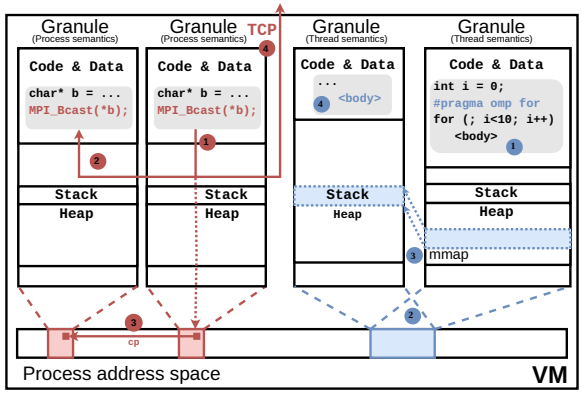


Figure 3: Granules memory layout in a single GRANNY instance (Granules execute side-by-side in an address space, and rely on WebAssembly's sandboxing for spatial isolation. Granules executing with processes semantics execute in different sandboxes, whereas Granules executing with thread semantics share the same sandbox.)

with processes semantics have separate linear memories, with the same application mapped into them. To send messages, the sender Granule indicates the buffer to be sent (Fig. 3, ❶), and the MPI backend in the GRANNY runtime decides if it is a local or a remote message. For a local message, the runtime can directly enqueue the message metadata (❷) and copy the buffer contents into the reception buffer (❸). For a remote message, the runtime will send the whole payload over TCP to the appropriate GRANNY runtime instance (❹).

Granules with thread semantics share a linear memory and have separate stacks. When a Granule creates new threads, such as with OpenMP `#pragma omp for` (Fig. 3, ❶), the GRANNY runtime spawns a new Granule mapped to the same linear memory (❷). To give each thread a separate execution context, but maintain WebAssembly's memory sandboxing, GRANNY allocates an area in the parent's heap for the child's stack (❸), and sets the child's code entrypoint to the corresponding OpenMP task (❹). Additional care goes into maintaining shared variable visibility, as well as their consistency.

Snapshots. Granules thus have a simple memory layout, which means that their execution state can be captured combining the span of the linear memory, together with the Granule state in the GRANNY runtime (Fig. 2). The combined linear memory and runtime state is what we call a Granule *snapshot*. Having a concise snapshot representation for Granules is essential for migration. Since all state is contained in a snapshot, GRANNY does not require host kernel modifications to obtain a Granule's full execution state. This is in contrast to general process checkpointing [11], which must also extract process state from the host kernel.

4 Granule Management

GRANNY performs management actions, vertical scaling (§4.2), and horizontal migration (§4.3) by interrupting application execution at well-defined *control points* (§4.1). This ensures GRANNY always operates with a consistent application state. The following sections describe these operations in more detail.

4.1 Interrupting Granules at control points

GRANNY takes control over Granule execution at control points, i.e., at every call to one of the supported runtime backend APIs (see Fig. 2). GRANNY defines two types of control points, *regular* and *barrier* control points, that enable the runtime core to perform different operations. At regular control points, such as calls to the POSIX API, the application state is not guaranteed to be consistent, but this still allows GRANNY to send point-to-point messages or operate on shared memory. In contrast, barrier control points guarantee that the application state is consistent, i.e., no messages are in-flight and no modifications to shared variables are pending to be synchronized. Therefore, GRANNY can only perform management operations at barrier control points.

Differentiating between regular and barrier control points requires semantic knowledge of the shared memory/message passing API, as well as control over its implementation. For example, GRANNY's MPI backend implementation of `MPI_Barrier` has a reduce phase in which all Granules send messages to the Granule with the lowest MPI rank, and a broadcast phase in which all Granules are notified that the barrier has completed. After the reduce phase, the Granule with the lowest MPI rank can rely on the fact that there are no outstanding messages and has thus reached a consistent state. A similar explanation, with shared memory synchronization instead of messages, applies to GRANNY's OpenMP backend implementation of `#pragma omp barrier`.

As a consequence, vertical scaling and horizontal migration in GRANNY is a co-operative process. When a Granule triggers a barrier control point, the runtime interacts with the scheduler and adds, removes, or migrates Granules as indicated. In practice, barrier control points are frequent enough, so that their co-operative nature does not hinder the benefits in resource management, as we can see later in §6.

4.2 Vertical scaling

Implementing vertical scaling in GRANNY is straightforward if we take into account the memory layout presented in Fig. 3. To create a child Granule from a parent Granule, the runtime allocates the child's stack in the parent's heap, and instantiates a new Granule on top of the same WebAssembly linear memory, changing only the stack pointer. The child's entrypoint is indicated as an index in the WebAssembly module's function table. We also add guard pages around each child's stack to mitigate potential stack overflows (typical of various threading implementations in WebAssembly [58]).

Since vertical scaling happens during barrier control points in OpenMP, the runtime can easily distribute the work across any Granules by simply setting the appropriate per-thread and

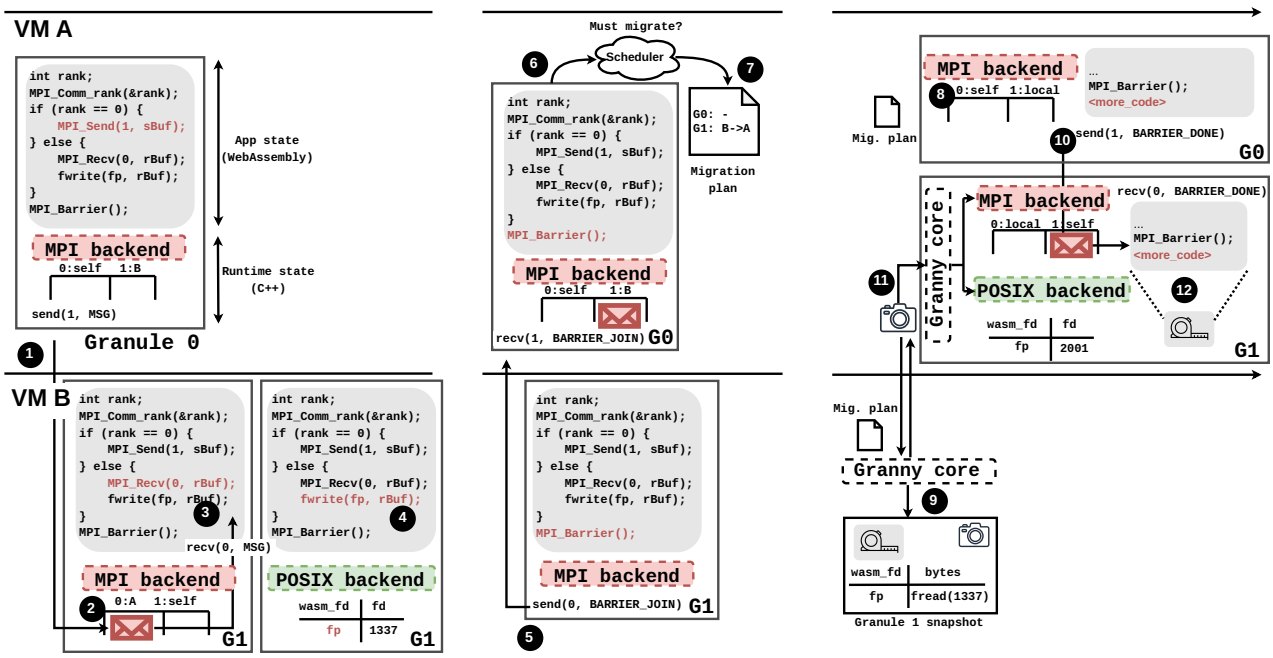


Figure 4: Horizontal migration overview (When Granules reach a barrier control point, such as `MPI_Barrier`, the application's main Granule queries the resource scheduler. If triggered, the runtime performs a horizontal migration by taking a snapshot of the migrated Granule and transferring it to the destination VM. Runtime state, such as open file descriptors, is also snapshotted and reconstructed at the destination.)

global variables in the OpenMP specification. We design a scheduling policy using vertical scaling in §5.2.

4.3 Horizontal migration

Fig. 4 illustrates horizontal migration. The MPI backend gives each Granule a unique integer identifier, akin to the MPI rank, and implements messaging as a three-step process: it captures the send operation as a call to a backend operation such as `MPI_Send` (❶); it copies the message into the receiving Granule's mailbox within the target GRANNY runtime core (❷); and it delivers the message to the application by capturing (❸), and potentially blocking, calls to a receive operation (which the MPI backend matches to the corresponding mailbox, according to the MPI specification). Note that Granules may also call other API backends, such as file writes via the POSIX backend (❹). As previously introduced, the POSIX backend serves the request and keeps track of the mapping between WebAssembly file descriptors and OS ones.

When Granules reach a barrier control point (such as `MPI_Barrier` in Fig. 4b), the runtime can perform horizontal migrations safely. For example, in the case of `MPI_Barrier`, all Granules send a `BARRIER_JOIN` message (❺) to the zero-th MPI rank (Granule 0 in VM A in the example), and wait for a `BARRIER_DONE` message before continuing. When the runtime for Granule 0 has received all messages, it queries the scheduler for any horizontal migrations (❻). The resource scheduler then applies the scheduling policy and returns a migration plan (❼). After all migrations have been performed (if any), the runtime broadcasts a `BARRIER_DONE` message so all blocked Granules resume execution. In our example, Granule 1 must be migrated from VM B to VM A.

To perform the migration, shown in Fig. 4c, the runtime in VM A distributes the migration plan to the other runtimes involved in the execution (in this case the runtime in VM B), and all proceed to individually prepare for the migration. For VM A, this means the runtime must create a new, empty Granule 1, setup its mailbox mappings, and update the entry for Granule 1 on the mailbox of Granule 0 (❽). For the runtime in VM B, this means creating a snapshot of Granule 1, including any runtime state, and terminating the old Granule 1 (❾).

Once the migration has been prepared, the runtime in VM A can resume execution of Granule 0, which broadcasts the `BARRIER_DONE` message to the *updated* mailboxes (❿). The runtime in VM A can also restore Granule 1 from the received snapshot, re-construct the file descriptor tables, and resume execution where it left off, i.e., blocked waiting for a `BARRIER_DONE` (⓫). Since the restored Granule 1 uses the updated mailboxes, it has the message and resumes execution (⓬). We design two scheduling policies using horizontal migration in §5.1 and §5.3.

5 Granular Management Policies

We implement three scheduling policies, described in the following sections, to improve the utilization and performance

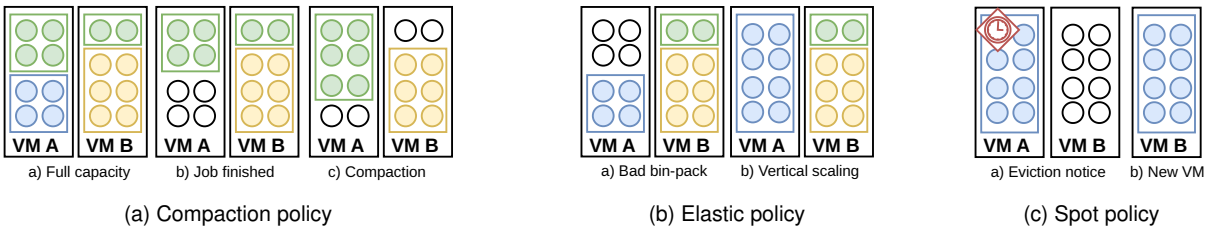


Figure 5: Dynamic scheduling policies implemented for their use with GRANNY (Different colors indicate different jobs.)

of cloud-based multi-process and multi-thread execution. The policies use GRANNY's support for vertical scaling and horizontal migration, and treat scheduled the applications as black boxes, i.e., without knowledge of future applications, their distribution of sizes, or their expected duration.

5.1 Improving locality with compaction policy

When scheduling MPI applications, existing cloud schedulers face a utilization/locality trade-off: they can either assign available vCPUs to an MPI application to achieve high utilization, or assign an entire VM to an MPI application to achieve high locality at the expense of having unused vCPUs in those VMs. The former reduces queue wait times for applications, but it may increase execution time, because applications may have to communicate across different VMs; the latter optimizes execution time at the cost of increasing queuing time due to the wait for an available VM of the required size.

Instead, we adopt a *compaction* policy (see Fig. 5a) that uses GRANNY to combine the benefits of both approaches. The scheduler first eagerly deploys MPI applications on any available vCPUs, which results in the highly fragmented green application in our example. When vCPUs are later released (e.g., the blue application terminates), the scheduler performs horizontal migration of Granules to increase the locality of executing applications. This results in less cross-VM communication, which speeds up the green application.

We define our fragmentation metric as the total number of Granule-to-Granule connections that cross a VM boundary in an application. When the application reaches a barrier control point, the scheduler checks if it can reduce its cross-VM links by performing any set of horizontal migrations. We show experimentally in §6.2 that, using this policy, GRANNY maintains high cluster utilization while keeping fragmentation low, thus improving performance.

5.2 Improving utilization with elastic policy

Even with a compaction policy, some cluster resources remain idle due to the nature of bin-packing scheduling. VMs may have some spare capacity that is insufficient to deploy the next application in the queue. This is particularly true with OpenMP applications that cannot be distributed across VMs.

In response, we introduce an *elastic* policy (see Fig. 5b). When there is spare capacity in a VM (Fig. 5b-a) and an OpenMP application reaches a barrier control point, the scheduler triggers a vertical scale-up to utilize these idle resources (Fig. 5b-b). This improves cluster utilization and application performance by exploiting extra parallelism. Our policy is careful to not mistake fork-join patterns of co-located OpenMP application with truly available resources by keeping track of the `OMP_NUM_THREADS` environment variable.

In §6.3, we show that the elastic policy reduces end-to-end execution time while decreasing idle vCPUs.

5.3 Ephemeral VMs with spot policy

Compute-intensive applications may be deployed on a pool of spot VMs [7] to benefit from lower costs. This introduces the challenge of handling partial failures, because spot VMs are withdrawn by the cloud provider after a short grace period.

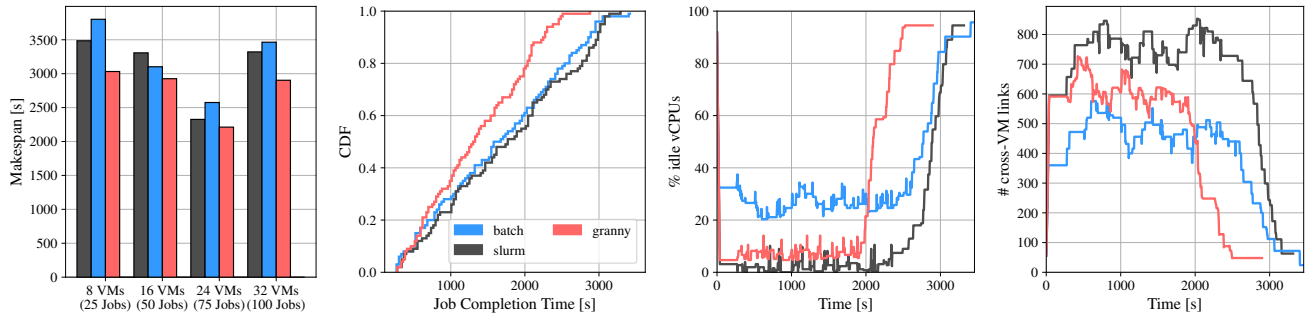
Fig. 5c shows our *spot* policy. We follow a two-part horizontal migration approach: (1) when the cloud provider notifies the scheduler of an upcoming spot VM eviction (Fig. 5c-a), the scheduler stops scheduling Granules to that VM; (2) when a Granule running on the to-be-evicted VM reaches a barrier control point (e.g., `MPI_Barrier` or `#pragma omp barrier`), the resource manger tries to re-schedule it on the remaining VMs (Fig. 5c-b). If there are insufficient resources, snapshots are taken of all Granules in the application, and they are terminated. Interrupted applications are then added to the beginning of the scheduler's queue.

We show experimentally in §6.4 that, using a spot policy, GRANNY reduces end-to-end execution time when deploying compute-intensive applications on spot VMs.

6 Evaluation

Our evaluation explores the benefits of using GRANNY to run MPI and OpenMP applications. We show how, using GRANNY's dynamic scheduling polices (§5), we can: (i) improve performance and locality of MPI applications while maintaining a target utilization (§6.2); (ii) improve performance and utilization of OpenMP applications by allocating extra CPU cores (§6.3); and (iii) ensure efficient fault-tolerant execution with ephemeral spot VMs (§6.4).

In our microbenchmarks (§6.5), we analyze the baseline overheads of: (i) executing MPI applications with GRANNY instead of OpenMPI (§6.5.1); (ii) executing OpenMP applications with GRANNY instead of LLVM's `libomp` (§6.5.2); (iii) the horizontal migration of Granules (§6.5.3); and (iv) the vertical scale-up of Granules (§6.5.4).



(a) Makespan (b) CDF JCT (32 VMs, 100 Jobs) (c) Idle vCPUs (32 VMs, 100 Jobs) (d) xVM Links (32 VMs, 100 Jobs)

Figure 6: Improving locality with compaction policy (We compare with Azure Batch and Slurm's default policies.)

6.1 Experimental setup

Implementation. GRANNY is implemented in 24,000 lines of C++20 code (compiled with clang-17) on top of the Faasm runtime [51]. GRANNY extends Faasm by implementing the MPI/OpenMP backends, a centralized scheduler, and migration/scale-up, but it re-uses the underlying sandbox abstraction (using WAMR [10] as the WebAssembly runtime). GRANNY is available as open-source.[1] We compile all deployed applications and their library dependencies, such as *libc*, to WebAssembly [17] using clang-18 [45], as part of GRANNY's CPP toolchain.

As baselines, we compare against two cloud resource schedulers, batch and slurm, that mirror the behaviour of Azure Batch [27] and Slurm [48], respectively. Unless otherwise stated, batch and slurm execute MPI applications with OpenMPI v4.1 [39] and OpenMP ones with `libomp` v4.5 [25].

Testbed. We deploy GRANNY and any baselines on a Kubernetes cluster [22] on Azure [28]. The cluster consists of *Standard_D8_v5* VMs [29] with 8 vCPU cores and 32 GB of memory. We deploy the resource scheduler in a separate VM in the same cluster.

Workloads. We evaluate GRANNY with unmodified applications that use OpenMP [36] for multi-threaded shared memory computation, and MPI [33] for multi-process message passing. Applications require no changes to source code, just re-compilation with the GRANNY toolchain.

6.2 Improving locality with compaction policy

This experiment explores the benefits of using a *compaction* policy (§5.1) to navigate the utilization/locality trade-off: it horizontally migrates Granules executing with process semantics. We execute a trace of MPI applications as jobs, and each application executes the LAMMPS [43] molecule dynamics simulator, running the Lennard-Jones (LJ) benchmark with a varying number of MPI processes. Applications are executed in order and may wait in a queue until sufficient resources become available in the cluster. To measure only the benefits of the policy and not differences in the MPI implementation, we configure our two baselines, batch and slurm, to use GRANNY's MPI backend.

Fig. 6 shows various performance metrics, as we increase the number of VMs and the number of applications in the trace. Fig. 6a shows that the compaction policy improves end-to-end execution time (makespan) by up to 20%. Using compaction, GRANNY always improves makespan across all baselines and cluster sizes.

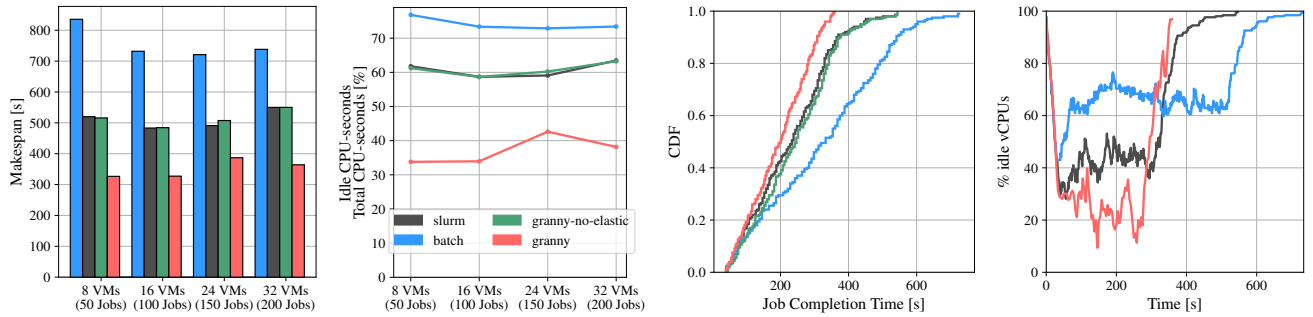
To show the compaction policy in action, Fig. 6c and Fig. 6d plot the time-series of idle vCPUs (as a proxy for cluster utilization) and cross-VM network links (as a proxy for locality) for the (32 VMs, 100 jobs) execution. We see that, differently to Slurm, GRANNY deliberately leaves a percentage of vCPUs idle corresponding to a target utilization (5% in this experiment). GRANNY can use these spare vCPUs to defragment applications at runtime by performing horizontal migrations, achieving consistently 25% less fragmentation than Slurm. In fact, GRANNY, with only 5% idle vCPUs, is closer in terms of fragmentation to Azure Batch, which behaves optimally with respect to this metric but leaves 30% of vCPUs unused.

Fig. 6b shows that this reduction in fragmentation at high cluster utilization has a direct impact on job completion time (JCT). For the (32 VMs, 100 jobs) execution, GRANNY improves median and tail JCT by up to 20%. We conclude that the compaction policy enables GRANNY to exploit the utilization/locality trade-off more effectively compared to Azure Batch and Slurm.

6.3 Improving utilization with elastic policy

This experiment explores the benefits of using an *elastic* policy (§5.2) to vertically scale OpenMP applications by adding Granules with thread semantics (§5.2). We execute a trace of OpenMP applications as jobs, and each application executes a large-scale version of the `p2p` ParRes OpenMP kernel [41], which performs a compute-intensive pipelined parallel algorithm on a large matrix, and requires a different number of

[1] https://github.com/faasm/faasm/



(a) Makespan (b) Aggregate idle vCPUs (c) JCT (32 VMs, 200 Jobs) (d) Idle vCPUs (32 VMs, 200 Jobs)

Figure 7: Improving utilization with elastic policy (We compare to Azure Batch and Slurm with `libomp` as OpenMP runtime, and Slurm with GRANNY's OpenMP backend, granny-no-elastic.)

OpenMP threads. In this experiment, the baselines (batch and slurm) execute applications using `libomp` [25]. We include an additional baseline, granny-no-elastic, which corresponds to slurm using GRANNY' OpenMP backend.

Fig. 7a shows that GRANNY improves makespan by up to 60% compared to the native baselines and GRANNY without the elastic policy (granny-no-elastic). This confirms that the performance improvements come from the policy. Indeed, we can assert that GRANNY reduces the idle CPU cores in the cluster by up to 30% (Fig. 7b), and it uses these extra cores to improve median and tail JCT by up to 50% (Fig. 7c).

This large performance gap can be understood when considering how many compute resources are left idle by the native baselines. Fig. 7d shows a time series of the percentage of idle vCPUs when running 200 jobs on a 32-VM cluster. Azure Batch and Slurm, even when the job queue is not empty, consistently leave 60% and 40% of vCPUs idle. This is due to multiple reasons: (i) OpenMP applications have fixed parallelism; (ii) it is not possible to distribute them across VMs; and (iii) in the case of Azure Batch, it is not possible to run different applications on the same VM concurrently. By vertically scaling-up, GRANNY maintains the fraction of idle vCPUs at around 20% while there are still pending jobs.

GRANNY's improvements in terms of JCT are partly due to the fact that our OpenMP workload always benefits from increasing its parallelism (§6.5.4). In a real deployment, an elastic policy should be accompanied by runtime profiling to determine when a workload exhausts its parallelism.

6.4 Ephemeral VMs with spot policy

This experiment explores the performance benefits of using a *spot* policy (§5.3) to execute compute-intensive applications on a cluster with ephemeral spot VMs. The submitted applications and the native baselines are the same as in §6.2. In this experiment, the native baselines execute applications using the OpenMPI [39] runtime. To emulate the behaviour of spot VMs while making our findings reproducible, we withdraw VMs at a pre-defined rate with a 1-min grace period.

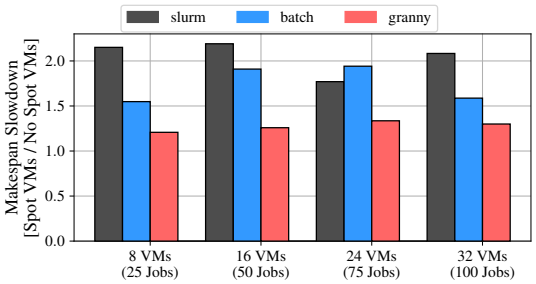
The eviction rate for this experiment is 25% of the VMs, selected at random, each minute. We choose this eviction rate after measuring the eviction rate for `Standard_D8_v5` spot VMs using Azure's Resource Graph Analyzer [8] (25% per hour) and scaling it to our MPI applications' length (minutes, instead of hours, to make the experiments reproducible).

Fig. 8a shows the reduction in makespan when comparing each baseline to itself without evictions. We see that, across cluster and batch sizes, all native baselines experience a minimum of a 50% slowdown, and a maximum of a 2× slowdown. This is because batch and slurm are unaware of evictions and must restart jobs each time they fail due to an evicted VM. Instead, GRANNY uses the spot policy and, as a consequence, its slowdown is at most 25%, consistent with the eviction rate.

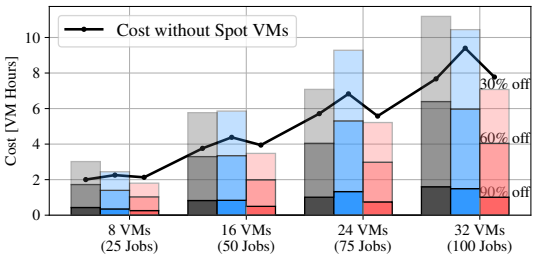
The native slowdowns of 50%–100% can potentially thwart the cost benefits of using spot VMs. Indeed, Fig. 8b shows the normalized cost of each execution for the range of discounts based on Azure's spot VM price list [6]: 30%, 60%, and 90%. We calculate the cost by assuming a unit price per VM-hour and applying the corresponding discount. We also overlay the cost of not using spot VMs.

We observe that, for the native baselines, the effectiveness of spot VMs in terms of cost savings depends on the discount rate at which they are offered. Counterintuitively, for many discount ranges, it is not cost-effective to run applications using OpenMPI on spot VMs, because the re-execution costs outweigh the price discount. In contrast, with GRANNY, it is always cost-effective to use spot VMs, because the eviction slowdown is lower than the smallest cost discount.

Discussion. Our experiments are conducted on a 32-VM cluster with job traces that contain one type of application and do not capture inter-arrival times. Given the co-operative and on-demand nature of vertical scale-up and horizontal migration, however, we expect our results to generalize to larger cluster sizes. We also believe that GRANNY can improve performance and utilization for other workloads, but this may require the design of further scheduling policies.

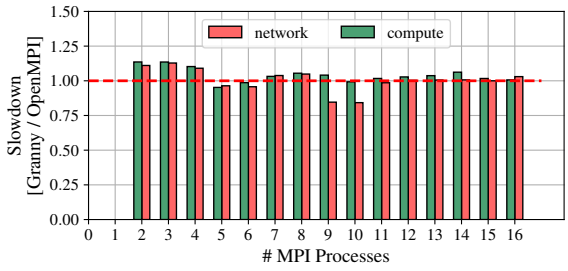


(a) Makespan

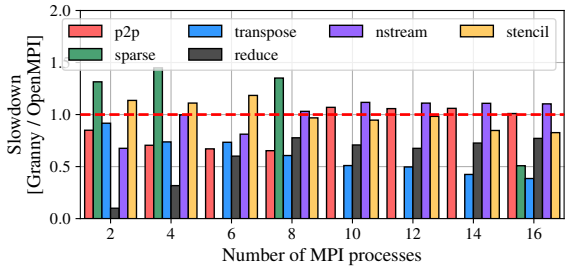


(b) Cost (Assuming a unit cost per VM hour.)

Figure 8: Ephemeral VMs with spot policy (We compare to Azure Batch and Slurm with their default policies and native runtimes.)



(a) LAMMPS



(b) MPI ParRes Kernels

Figure 9: MPI backend performance

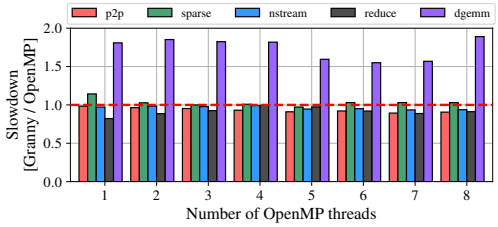


Figure 10: OpenMP backend performance

6.5 Microbenchmarks

In this section, we isolate different components of GRANNY and evaluate overheads. We evaluate GRANNY's MPI backend (§6.5.1), OpenMP backend (§6.5.2), as well as horizontal migration (§6.5.3) and vertical scale-up (§6.5.4).

6.5.1 MPI backend

This experiment investigates the overhead of GRANNY's MPI backend implementation compared to OpenMPI [39]. We run the same MPI application as in §6.2. To stress GRANNY's communication layer, we update the benchmark and increase the synchronisation steps, resulting in three orders of magnitude more cross-VM messages. We refer to the unmodified LJ benchmark as compute, and the modified one as network. We also execute a subset of the ParRes kernels [41] to evaluate specific parts of GRANNY's MPI implementation.

Fig. 9a shows the slowdown in execution time of GRANNY compared to OpenMPI when executing the two LAMMPS simulations with different levels of parallelism on two VMs. We observe that the overhead introduced by GRANNY is within 10% and often negligible. GRANNY occasionally introduces minor performance gains due to the benefits of intra-process co-location of Granules.

Fig. 9b shows the slowdown when executing the ParRes kernels with different levels of parallelism. For this workload, the performance of GRANNY and OpenMPI varies more than for LAMMPS, because these kernels execute MPI operations in a tight-loop. As a consequence, the performance benefit of intra-process co-location of Granules becomes more significant, leading to a performance benefit for GRANNY. In these cases, GRANNY can replace the sending of messages (reduce) with reducing shared memory variables.

6.5.2 OpenMP backend

Next, we investigate the overhead of GRANNY's OpenMP backend compared to LLVM's `libomp`. We execute the ParRes kernels [41] in their OpenMP implementation. We execute each kernel with a varying number of threads and take the average execution time over 5 runs.

Fig. 10 shows the slowdown in execution time of GRANNY compared to `libomp` for a variety of kernels. We observe that, for most kernels, GRANNY's performance matches the native baseline. This is because GRANNY's OpenMP backend adds minimal book-keeping at runtime and relies on the OS' synchronization primitives. For the `dgemm` kernel, GRANNY introduces an 80% slowdown. This kernel performs a dense matrix multiplication, and the overheads come from WebAssembly's less efficient floating-point operations [51]. We expect future WebAssembly releases and compilers to improve floating-point performance.

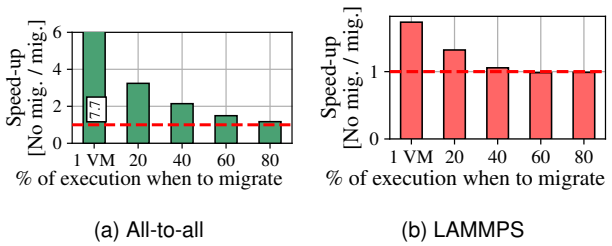


Figure 11: Speedup when migrating Granules (We deploy 8 Granules with process semantics across two VMs, and migrate 4 at runtime. We report the speedup over no migration.)

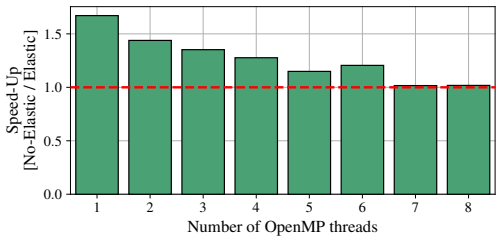


Figure 12: Speedup when elastically scaling to more vCPUs (We deploy a varying number of Granules with thread semantics, and elastically scale up to all CPU cores after 50% of execution.)

6.5.3 Granule migration

This experiment explores GRANNY's performance overhead when doing a horizontal migration of Granules at runtime, and the benefits of improved co-location. As workloads, we use the compute-bound LAMMPS simulation, and a network-bound *all-to-all* kernel, which performs synchronisation over a vector in a loop. For each experiment, we artificially fragment the 8 Granules with process semantics across two VMs. We then trigger a horizontal migration of half of them to the other VM at 20%, 40%, 60%, or 80% of execution time.

Fig. 11a shows the speedup when migrating a network-bound application compared to no migration. For such an application, fragmentation has a high cost: the speedup for running in one VM (1 VM) is 7.7×. By migrating after 20%, 40%, 60%, and 80%, of execution, GRANNY achieves speedups of 3.5×, 2.2×, 1.5×, and 1.1×, respectively. For network-bound applications, the overheads of horizontal migration are thus outweighed by the benefits of co-location.

Fig. 11b shows the speedup when migrating a compute-bound application. Here, fragmentation is less of an issue: the speedup for running in one VM is 1.7×. By migrating after 20% and 40% of execution, we achieve speedups of 1.3× and 1.1×, respectively. We observe no benefit when migrating later during execution. There exists a trade-off between the frequency and cost of migration, however, GRANNY's horizontal migration mechanism introduces a negligible overhead, thus enabling cloud providers to optimize for locality.

Migration time is dominated by the time to transfer the snapshot from one VM to another. For a 4 MB snapshot, corresponding to the *all-to-all* kernel, Granule migration is on the order of 30 ms. From this, only 3 ms correspond to creating the snapshot, which is almost an order of magnitude faster than a highly-optimized version of CRIU [60].

6.5.4 Elastic scale-up

Finally, we investigate GRANNY's performance overhead when doing vertical scaling of a multi-threaded application, and the potential benefits of increased parallelism. As a workload, we deploy the same OpenMP application as in §6.3. We initially use a varying number of Granules, and scale up to all available CPU cores (8) after 50% of execution.

Fig. 12 shows that, by scaling-up to use more CPU cores, GRANNY achieves a speedup of up to 60% when scaling from 1 to 6 threads. With more than 7 initial threads, we do not observe a benefit, because we have exhausted the application's parallelism. We conclude that elastically scaling-up does not slow down execution, because elastic scaling naturally fits with OpenMP's fork-join semantics.

7 Discussion

We now discuss some of GRANNY's design decisions and how its insights could be transferred to other cloud systems.

WebAssembly. Despite offering an efficient sandboxed environment where Granules execute, WebAssembly is still an evolving technology. The performance of some WebAssembly operations, particularly floating point operations, still remains below natively compiled code [14] and sandboxes that exceed 4 GB incur a higher sandboxing overhead [56]. We expect these issues to be addressed as compiler support for WebAssembly matures.

Programming models. GRANNY supports OpenMP and MPI and is therefore capable of running a large set of compute-intensive applications. GRANNY's design, however, is not tied to any programming model, and other programming models could be supported after identification of barrier control points for elastic scaling and migration.

A limitation of GRANNY is that it offers a CPU execution model only and cannot support GPU-based computation. This is due to its reliance on WebAssembly as an execution format. We plan on exploring GPU-based Granules with APIs such `wasi-nn` [59] in future work.

Resource schedulers and policies. GRANNY can integrate with other schedulers. For this, a resource scheduler needs to expose an endpoint to receive notifications from running applications and implement dynamic scheduling policies.

8 Related Work

Compute-intensive cloud applications. All major cloud providers offer solutions to support compute-intensive applications in the cloud [5, 30, 31]. To execute such applications, providers deploy batch scheduling solutions [2, 27] inspired

by HPC batch schedulers. Much previous work focuses on making scheduling decisions more effective [24].

Complementary to this, our work demonstrates that vertical scaling of multi-threaded applications and horizontal migration of multi-process ones is a fundamental missing piece for granular management of compute-intensive applications in cloud environments. GRANNY enables a number of scheduling policies that can be implemented to jointly optimize resource utilization and compute and data locality.

Checkpointing and migration. Nu [46] is a distributed computing platform that supports resource *fungibility* through migration. It uses multi-threaded *Proclets*, which can only communicate with each other by sending messages. This means that Nu's execution model is not transparent to existing multi-threaded applications, but rather requires a complete application re-write to partition shared application state and use its C++ messaging API. In contrast, GRANNY transparently executes and migrates existing multi-threaded and multi-process applications with no extra source code modifications required (with only recompilation to WebAssembly).

CloudScale [50] automates fine-grained elastic resource scaling in a shared cluster. It uses migration to handle scheduling or scaling issues. GRANNY is complementary to CloudScale in that it focuses on fine granular (thread/process) resource management.

There are various approaches to migrate parts of running applications transparently without violating integrity. MigrOS [42] checkpoints the state of RDMA-enabled applications executing in containers, but requires expensive modifications to the underlying network protocol. Other work [54] has performed live migration of MPI applications with process-level migration, which incurs high bandwidth consumption and necessitates OS kernel modifications. General-purpose process (e.g., CRIU [11] or DMTCP [15]) or VM migration techniques have been shown [18] to be poorly suited for efficient shared memory and message passing applications.

In contrast, GRANNY executes entirely in user-space and exploits lightweight sandboxing and traps using WebAssembly. Through its semantic knowledge of shared memory and message passing APIs, GRANNY achieves the same migration support but at a lower performance cost, independently of the employed VMs or host OS kernel.

Shared memory and message passing runtimes. Hoplite [62] uses collective communication primitives for building fault-tolerant task-based distributed applications. Ray [32] unifies task-parallel and actor-based computation using a single interface. It offers transparent state and message passing, irrespective of distribution, together with transparent scaling.

GRANNY could be extended to support these programming models. Our initial prototype implementation of GRANNY focuses on two existing programming models, namely MPI and OpenMP, which constitute opposite ends of the spectrum when writing parallel applications.

GRANNY's threading implementation is similar to that of `wasi-threads` [58]: both rely on WebAssembly's atomic intrinsics and have similar memory layouts, but GRANNY delegates control of shared memory synchronization (e.g., mutexes and locks) to the host runtime, whereas `wasi-threads` handles it in WebAssembly.

Elastic scaling. Elasticity for cloud-based execution has been traditionally implemented at the programming model level. MapReduce [12], actor-based models such as Akka [1] or Orleans [9], or serverless system such as AWS Lambda [3] or Faasm [51] implement elasticity at the worker/actor/function levels. In this work, we enable elasticity at the finer granularity of threads in multi-threaded applications. OpenMP has some elasticity support via `omp_set_dynamic`, but such support is only used to scale *down* in case of contention [38].

9 Conclusions

Existing resource schedulers fail to jointly optimize resource utilization and application performance, because application resource allocation cannot be modified after an application begin execution. To address this issue, we presented GRANNY, a new distributed runtime for unmodified multi-threaded (OpenMP) and multi-processing (MPI) applications. Through its Granule abstraction, GRANNY is able to spawn and migrate Granules at run-time. With this mechanism, we implement dynamic scheduling policies that improve both resource utilization and application performance when compared to existing cloud resource schedulers.

Acknowledgments. This work has been partially funded by the European Union through the Horizon Europe projects CloudStars (101086248) and CloudSkin (101092646), and by FCT (UIDB/50021/2020). We thank our shepherd, Xin Jin, and the anonymous reviewers for their helpful feedback.

References

[1] Akka. Build, Operate, and Secure Distributed Applications. https://akka.io/, 2024.

[2] Amazon Web Services. AWS Batch. https://aws.amazon.com/batch/, 2021.

[3] Amazon Web Services. AWS Lambda. https://aws.amazon.com/lambda/, 2021.

[4] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[5] AWS. Genomics in the cloud. https://aws.amazon.com/health/genomics/, 2022.

[6] Azure. Pricing - Linux Spot VMs. https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux/#pricing, 2024.

[7] Azure. Use Spot VM Instances. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms, 2024.

[8] Azure. Use Spot VM Instances - Eviction Rate and Pricing History. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms#pricing-and-eviction-history, 2024.

[9] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[10] ByteCode Alliance. WebAssembly Micro Runtime. https://bytecodealliance.github.io/wamr.dev/, 2024.

[11] CRIU. Checkpoint-Restore in Userspace. https://www.criu.org/Main_Page, 2021.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[13] FFmpeg Contributors. Webassembly lld port. https://lld.llvm.org/WebAssembly.htmlimports, 2022.

[14] Frank Denis Blog. Performance of WebAssembly runtimes in 2023. https://00f.net/2023/01/04/webassembly-benchmark-2023/, 2024.

[15] Google. Github - DeepVariant. https://github.com/google/deepvariant, 2022.

[16] Google. Google Cloud Batch. https://cloud.google.com/batch, 2023.

[17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[18] Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[19] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Veriwasm: Sfi safety for native-compiled wasm. NDSS'21.

[20] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, 1992.

[21] kube batch. A batch scheduler of kubernetes for high performance workload, e.g. AI/ML, BigData, HPC. https://github.com/kubernetes-retired/kube-batch, 2024.

[22] Kubernetes. Kubernetes- Production-Grade Container Orchestration. https://kubernetes.io/, 2022.

[23] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. Ribbon: Cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[24] Bing Lin, Wenzhong Guo, Xianghan Zheng, Hong Zhang, Chunming Rong, and Guolong Chen. Optimization scheduling for scientific applications with different priorities across multiple clouds. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, CLOUDCOM '14, 2014.

[25] LLVM Project. LLVM/OpenMP documentation. https://openmp.llvm.org/, 2022.

[26] LLVM Project. LLVM OpenMP Runtime Library Interface. https://openmp.llvm.org/doxygen/index.html, 2024.

[27] Microsoft. Azure Batch. https://azure.microsoft.com/en-us/services/batch/, 2021.

[28] Microsoft. Azure: Cloud Computing Services. https://azure.microsoft.com/en-us/, 2021.

[29] Microsoft. Azure Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series, 2021.

[30] Microsoft. Azure Genomics. https://azure.microsoft.com/en-us/services/genomics/, 2022.

[31] Microsoft. Azure HPC. https://azure.microsoft.com/en-gb/solutions/high-performance-computing, 2022.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI'18, 2017.

[33] MPI. MPI Forum. https://www.mpi-forum.org/, 2022.

[34] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, 2023.

[35] OpenFOAM. Github - OpenFOAM. https://github.com/OpenFOAM/OpenFOAM-dev, 2022.

[36] OpenMP. The OpenMP API specification for parallel programming. https://www.openmp.org/specifications/, 2021.

[37] OpenMP Api Specification. OMP_NUM_THREADS. https://www.openmp.org/spec-html/5.0/openmpse50.html, 2022.

[38] OpenMP Architecture Review Board. OpenMP application program interface version 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[39] OpenMPI. OpenMPI: Open Source High Performance Computing. https://www.open-mpi.org/, 2021.

[40] OpenMPI. mpirun - Man Page. https://www.open-mpi.org/doc/v3.0/man1/mpirun.1.php, 2024.

[41] ParResKernels Team. Parallel Research Kernels. https://github.com/ParRes/Kernels, 2021.

[42] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. MigrOS: Transparent Live-Migration support for containerised RDMA applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX ATC'21, 2021.

[43] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 1993.

[44] Press Office. Up to 1.2 billion for weather and climate supercomputer. https://www.metoffice.gov.uk/about-us/press-office/news/corporate/2020/supercomputer-funding-2020, 2022.

[45] LLVM Project. Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2022.

[46] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving Microsecond-Scale resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, NSDI'23, 2023.

[47] Sandia National Laboratories. Github - LAMMPS. https://github.com/lammps/lammps, 2020.

[48] SchedMD. Slurm Workload Manager. https://slurm.schedmd.com/overview.html, 2024.

[49] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

[50] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[51] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *USENIX Annual Technical Conference (USENIX ATC)*, USENIX ATC'20, 2020.

[52] TensorFlow. Github - TensorFlow. https://github.com/tensorflow/tensorflow, 2022.

[53] Volcano. Cloud native batch scheduling system for compute-intensive workloads. https://volcano.sh/en/, 2024.

[54] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in hpc environments. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[55] WASI. WebASsembly System Interface. https://wasi.dev/, 2024.

[56] WebAssembly. Memory with 64 bit indexes. https://github.com/WebAssembly/memory64, 2024.

[57] WebAssembly. WASI libc implementation for WebAssembly. https://github.com/WebAssembly/wasi-libc, 2024.

[58] WebAssembly. WASI Threads. https://github.com/WebAssembly/wasi-threads, 2024.

[59] WebAssembly. WASI Threads. https://github.com/WebAssembly/wasi-nn, 2024.

[60] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, OSDI'23, 2023.

[61] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2010.

[62] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, 2021.

On Temporal Verification of Stateful P4 Programs

Delong Zhang* Chong Ye* Fei He†
School of Software, BNRist, Tsinghua University, Beijing 100084, China
Key Laboratory for Information System Security, MoE, China

Abstract

Stateful P4 programs offload network states from the control plane to the data plane, enabling unprecedented network programmability. However, existing P4 verifiers overapproximate the stateful nature of P4 programs and are inherently inadequate for verifying network functions that require stateful decision-making.

To overcome this limitation, this paper introduces an innovative approach to verify P4 programs while accounting for their stateful feature. We propose a specification language named P4LTL, tailored for describing temporal properties of stateful P4 programs at the packet processing level. Additionally, we introduce a novel concept called the *Büchi transaction*, representing the product of the P4 program and the P4LTL specification. The P4 program verification problem can be reduced to determining the existence of any fair and feasible trace within the Büchi transaction. To the best of our knowledge, our approach represents the first endeavor in temporal verification of stateful P4 programs at the packet processing level. We implemented a prototype tool called p4tv. Evaluation results demonstrate p4tv's effectiveness and efficiency in temporal verification of stateful P4 programs.

1 Introduction

Software Defined Networking (SDN) [22] is an innovative network architecture that offers unprecedented network flexibility and programmability by decoupling the control plane and data plane. In traditional SDN architecture, the data plane's sole responsibility is forwarding packets [30]. Unfortunately, this configuration often results in frequent communication between the control and data planes, leading to significant network delay and overhead [55].

Recently, a more intelligent data plane has emerged, capable of maintaining network states locally to determine forwarding behaviors [9, 32, 38, 43, 45, 46, 49]. This novel architecture is known as the *stateful data plane* [55]. By updating the states of the data plane locally and adjusting forwarding behaviors based on these local states, the architecture reduces the excessive controller interaction. As a result, it offers faster response time and enhances network flexibility. In this paper, we focus on P4 [10], a widely adopted data plane language that supports the stateful data plane.

Formal methods play a pivotal role in ensuring the correctness of P4 programs. In recent years, numerous P4 program verifiers have been developed, including ASSERT-P4 [24], Aquila [50], bf4 [21], p4v [40], P4-NOD [41], and Vera [47]. However, these existing P4 program verifiers have primarily focused on per-packet processing. They treat the register states of P4 programs as fully nondeterministic at the beginning of each packet processing, simplifying the checks for continuous packet processing events into a single check. Consequently, they overlook the preservation of network states, treating each packet processing in isolation. As a result, these P4 program verifiers are inherently inadequate for verifying network functions that require stateful decision-making [31]. For example, a simple functionality like "if link failure is detected, then packets arriving *afterward* will all be forwarded to the backup port" [38, 45, 49] is not currently expressible and verifiable by existing verifiers.

Verifying P4 programs while accounting for their stateful nature presents significant challenges. The first challenge relates to environment modeling. It is important to note that P4 programs are deployed within switches, which continuously receive packets from the network and invoke the P4 programs to process these packets. The processing of packets by the P4 program can be influenced by the network states, and the behaviors of the P4 program for different packets can be interrelated. Therefore, when verifying the correctness of a P4 program, it is inadequate to consider its behaviors solely in the context of processing a single packet (as in the existing approach [24, 40, 47, 50]); instead, we should consider its behaviors in processing a series of packets. To address this challenge, we propose the use of an environment model (in Section 3) and verify the behaviors of P4 programs within the

*Both authors contribute equally to this work.
†Fei He is the corresponding author.

context of this environment model.

The second challenge pertains to the specification language used to describe the behaviors of a P4 program as it processes a series of packets. Unlike current P4 verifiers [21, 24, 40], which primarily focus on verifying properties for individual packet processing, our task now requires specifying properties that span multiple packet processings. This is considerably more challenging as it demands consideration of multiple executions of the P4 program. Existing data plane specification languages tend to either be too high-level, focusing on events of the entire network but not the executions of the P4 programs (e.g., NetKAT [3], SNAP [4], and NICE [11]), or too low-level, focusing solely on events within individual packet processing (e.g., NetCTL [47]). Although some simple safety properties (i.e., ensuring that something bad never happens [35]) can be reformulated into assertions for a single packet processing, the specification language for individual packet processing generally lacks the capacity to depict complex properties, especially those related to liveness (i.e., ensuring that something good eventually happens [35]). To tackle this challenge, we extend Linear Temporal Logic (LTL) [44] and introduce a specification language called P4LTL (in Section 4) that facilitates specifying temporal properties of P4 programs at the packet processing level.

Throughout the paper, we use the term *temporal verification of stateful P4 programs* to refer to the verification of temporal properties that span multiple packet processings. The third challenge lies in the development of algorithms for conducting such verification. It is important to note that the granularity of time steps plays a crucial role in temporal verification. In our specific context, we need to consider two distinct time steps: the time step that occurs after the execution of a P4 program statement, referred to as the *small time step*, and the time step that occurs after processing a packet, referred to as the *big time step*. Notably, a big time step comprises multiple small time steps. We introduce a novel concept known as a *Büchi transaction*, representing the product of the P4 program and the P4LTL specification. This concept is carefully defined to harmonize these two distinct time steps. We prove that the P4 program satisfies the P4LTL specification if and only if the Büchi transaction does not exhibit any fair and feasible trace. Building upon this observation, we develop an efficient verification algorithm (in Section 5) for temporal verification of stateful P4 programs.

With the above techniques, we develop p4tv, a temporal verifier tailored for stateful P4 programs. The framework of p4tv is illustrated in Figure 1. Given a P4 program and a P4 specification as inputs, it either returns "Verified" if the program meets the specified property or provides a counterexample that exposes a defect in the program.

We manually collected 9 benchmarks of stateful P4 programs from open-source communities and formulated 19 temporal verification tasks to evaluate p4tv. Experiment results demonstrate both effectiveness and efficiency of p4tv in verifying the temporal properties of stateful P4 programs. Furthermore, we conducted experiments to evaluate p4tv's efficiency in assertion verification and to assess its scalability concerning program complexity and specification complexity.

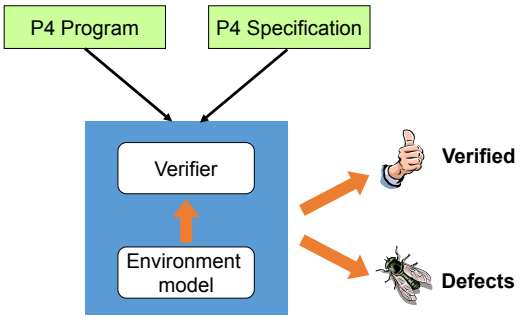


Figure 1: The workflow of p4tv .

Our contributions can be summarized as follows:

- We present, to the best of our knowledge, the first approach for temporal verification of stateful P4 programs.
- We design P4LTL, a specification language for describing temporal properties of stateful P4 programs at the level of packet processing.
- We propose a novel concept called the Büchi transaction, representing the product of a P4 program and a P4 specification. This concept allows us to reduce the P4 program verification problem to determining the existence of any fair and feasible trace within the Büchi transaction.
- We implement a prototype tool called p4tv and show its effectiveness on publicly-accessible benchmarks.

2 Motivations

In this section, we present the processors-based network immune scheme (P4NIS) mechanism [38] to provide motivation for our approach. P4NIS is designed to safeguard the network from packet eavesdropping within the programmable data plane. A snippet of P4NIS code (slightly modified for clarity) is depicted in Figure 2, where *count* serves as a register variable and *check_passed* validates whether all forwarding conditions are met. This validation includes a packet validity check (i.e., the ingress port and header validity check) and a range check on *count* (i.e., $0 \leq count \leq 2$). Upon the arrival of a packet that passes *check_passed*, the switch circularly distributes the packet traffic and executes various eavesdropping defense mechanisms.

One essential safety property for P4NIS states: *"The forwarding ports for valid incoming packets should always remain within the valid range (1 to 3 inclusively)."*

Nevertheless, existing P4 verifiers assess P4 programs on a single-packet basis, abstracting registers as fully nondeterministic. Consequently, these verifiers will report that the property

```
1  count.read(temp,(bit<32>)0);
2  if(check_passed()){
3      // send packages circularly
4      temp = temp + 1;
5      standard_metadata.egress_spec = temp;
6      if (temp == 3) temp = 0;
7      count.write((bit<32>)0,(bit<32>) temp);
8      // other operations
9      ...
10 }
```

Figure 2: The snippet of P4NIS code.

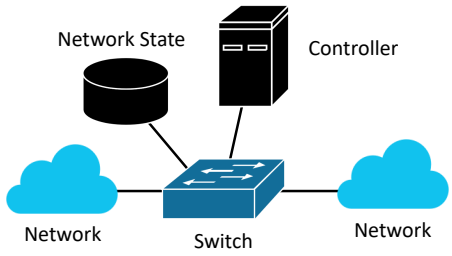


Figure 3: The environment model of p4tv.

does not hold for P4NIS. This is evidenced by a counterexample where the register *count* is initially set to a value of 4, bypassing the condition check (at line 2) and resulting in an implicit drop.

However, to reason about the precise behaviors of stateful P4 programs, it is necessary to analyze them across multiple packets. Considering the P4NIS example, if the register *count* is initialized with 0 during network setup, as per the code from lines 4 to 7, the value of *count* cannot exceed the valid range regardless of the number of packets processed by this P4 program, thereby proving the property. This underscores the importance of assessing P4 programs in the context of multiple packets.

One could suggest revising the property to: *"If the value of count is valid at the start of processing, the forwarding ports will remain valid."* While this simplifies verification into a single-packet check, identifying the specific precondition (i.e., the value of *count* falls within the valid range) requires substantial expertise and manual effort.

Furthermore, there exist properties that can hardly be simplified into a single-packet check. For instance, consider the liveness property for P4NIS that ensures the consistent activation of all defense mechanisms: *"When valid packets arrive infinitely often, they should be forwarded to every valid port infinitely often."* Expressing liveness concept like "infinitely often" in the context of a single-packet check is challenging.

These challenges motivate us to model the P4 execution process in a stateful manner (Section 3), introduce specification language P4LTL for stateful P4 programs (Section 4), and propose its corresponding automated verification algorithm (Section 5). Leveraging these techniques, p4tv is able to effectively and automatically verify complex temporal properties.

3 Environment Model

A P4 program specifies the behavior of a switch upon receiving a packet. To ensure the correctness of P4 programs, it is necessary to specify their environment model. As illustrated in Figure 3, the programmed switch receives packets from the network and processes them based on rules provided by the controller and local network states. In Section 3.1, we present our modeling mechanism for network environment and register states in comparison with existing P4 verifiers. In Section 3.2, we introduce our control plane interface which models the controller part.

3.1 The Basic Model

The modeling mechanism employed in current P4 verifiers can be seen as follows:

```
1  procedure main(){
2      nondet_register_variables();
3      call P4Program();
4  }
```

The model only considers the processing of a single packet and assumes that register variables are fully nondeterministic. However, this level of abstraction overlooks the stateful nature of packet processing, where register values in fact persist and are preserved for subsequent processing.

The real-world switch may receive arbitrarily many packets from the network environment. As a result, the P4 program needs to be executed arbitrarily many times. To model the stateful nature of packet processing, we regard a P4 program as an infinite while loop, similar to the idea presented in the probabilistic P4 test tool [31]. Since we consider the correctness of the P4 program, the network environment should not be constrained and the input packets should be nondeterministic. Therefore, we add an initialization block at the entry of the while-loop, in which we assign each packet field a nondeterministic value. The metadata fields in P4 programs are initialized in accordance with the P4-16 specification [16]. Considering that the control plane may change constantly, the table applications are modeled with a nondeterministic action choice, similar to the technique used in p4v [40] and bf4 [21]. This approach ensures that the control plane remains unconstrained. Notably, we declare the registers outside the loop to keep their value after each iteration (i.e., each packet processing). Our modeling mechanism is as follows:

```
1  procedure main(){
2      declare_registers();
3      while(true){
4        packet_initialization();
5        call P4Program();
6      }
7  }
```

$$
\begin{array}{lcl}
\text{P4LTL}\,\varphi & \rightarrow & \psi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \\
& & \mid \circ\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \varphi\, \mathcal{W}\, \varphi \\
\text{Pred}\,\psi & \rightarrow & t\,\text{comp}\,t \mid \text{fwd}(t) \mid \text{valid}(h) \mid \text{hit}(t) \mid \text{drop} \\
& & \mid \text{apply}(t,a) \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \psi \rightarrow \psi \\
\text{Term}\,t & \rightarrow & d \mid h \mid m \mid v \mid r[t] \mid \text{old}(t) \mid \text{key}(t,k) \\
& & \mid \text{para}(t,a,p) \mid t\ \text{op}\ t
\end{array}
$$

Figure 4: Syntax of P4LTL.

It is important to note that our modeling mechanism differs from previous P4 verification studies. Existing P4 verifiers only consider a single packet processing, assuming register states are fully nondeterministic at the start of each packet processing. While this technique simplifies the verification process, it hinders the ability to track changes of register states throughout packet processings. On the other hand, we consider the execution sequences in a continuous and correlated manner, reflecting their actual occurrence. This approach models the stateful nature of packet processing, which is crucial for stateful P4 program analysis and verification.

3.2 Control Plane Interface

Some P4 functionalities cannot be properly verified without additional information regarding the control plane. Therefore, we present *control plane interface* (CPI). which facilitates users specifying the control plane assumption.

For instance, consider a scenario where a rule exists in the table *rt* that triggers the action fwd with the parameter *pt* set to 1 when the key $m.d$ equals 2. In this case, the corresponding rule can be expressed in CPI as follows:

$$
\Box(\text{hit}(rt) \wedge \text{key}(rt, m.d) == 2 \rightarrow \\
\text{apply}(rt, fwd) \wedge \text{para}(fwd, pt) == 1).
$$

CPI provides a novel feature that assists developers by allowing not only concrete table entries but also symbolic ones. For instance, developers can modify the equality condition above to inequality, creating a symbolic CPI. This symbolic CPI encompasses the set of rules triggering fwd when the key $m.d$ is not equal to 2. This allows developers to verify P4LTL under control plane snapshot (i.e., table rules) sets instead of a single control plane snapshot. A blank CPI simply indicates that every table rule is possible.

4 Specification Language

To specify temporal properties spanning states of packet processing in stateful P4 programs, we introduce P4LTL, a language that extends the well-established formalism of Linear Temporal Logic (LTL) [44].

Among various temporal logics such as CTL, LTL, and CTL* [51], all capable of expressing temporal properties, we choose LTL as the foundational logic of P4LTL for the following reasons: (1) its linear time framework aligns more naturally with our problem domain, and (2) its rich expressiveness allows for specifying intricate properties along the packet processing traces.

4.1 P4LTL

The syntax of P4LTL is shown in Figure 4. In this section, we describe the P4LTL design in a bottom-up fashion.

Terms. Terms of P4LTL mainly consist of integer numbers d, packet headers h, metadata m, header field h, free variables v and register r. They represent the corresponding values *at the end of* packet processing. Furthermore, $key(t,k)$ denotes the value of key k when applying table t, while $para(t,a,p)$ represents the value of parameter p for action a when invoked by table t. Additionally, P4LTL provides a special construct $old(t)$ to refer to the value *at the start of* the ingress pipeline (i.e., at the end of the parsing pipeline). For instance, we can use $old(r[i])$ and $old(h)$ to refer to the register value of r at index i and the header value at the start of core packet processing. [1] Moreover, P4LTL allows the use of arithmetic operations between two terms with the arithmetic operator op that is supported by P4. Notably, P4LTL supports the use of free variables v, whose values are non-deterministic and can be further constrained by predicates.

Predicates. P4LTL provides several predicates for users to describe packet forwarding behaviors. To be specific, the semantics of predicates are defined as follows:

1. Predicate $t\ comp\ t$ is true if the corresponding terms meet the situation in the packet forwarding process, where *comp* denotes the comparison operators in P4.
2. Predicate $valid(h)$ is true if the corresponding header h is valid after the parsing progress.
3. Predicate *drop* is true if the packet is dropped at the end of packet processing.
4. Predicate $fwd(t)$ is true if the packet is forwarded to port t at the end of packet processing.
5. Predicate $hit(t)$ is true if the table t is hit during the packet processing.
6. Predicate $apply(t,a)$ is true if the action a is invoked by table t during the packet processing.

Note that $fwd(t)$ can be rewritten as a special case of the predicate $t\,\text{comp}\,t$. In P4 programs, the forwarding port can be

[1] For the sake of completeness, we consider the values of $old(key(t,k))$ and $old(para(t,a,p))$ to be arbitrary.

represented as *standard_metadata.egress_spec*, so the predicate *standard_metadata.egress_spec* $== t$ is equivalent to $fwd(t)$. We present it as a syntactic sugar for convenience.

Time step for stateful P4 programs. The LTL component is the key part of P4LTL. As the semantics of LTL hinge on the idea of *execution trace*, it is necessary to define what its *time step* is at the outset. An execution trace constitutes a sequence of valuations, which map P4LTL terms to their values at the end of every packet processing. A time step is performed every time the P4 program finishes processing a packet. It is important to note that the semantics of P4LTL are different from LTL in standard program verification [17, 18, 20] or NetCTL [47]. P4LTL concentrates on temporal properties across multiple packet processing, whereas other approaches primarily center around temporal properties across statements or pipelines in a single packet processing.

P4LTL. Based on the illustrations of P4LTL time steps, we define the semantics of P4LTL operators.

- Operator *next* $\circ$ is satisfied when the condition holds at the end of the next packet processing.
- Operator *eventually* $\Diamond$ is satisfied when the condition holds at the end of some packet processing eventually.
- Operator *always* $\Box$ is satisfied when the condition holds at the end of every packet processing.
- Operator *until* $\mathcal{U}$ is satisfied if the left-hand condition stays true until the right-hand condition becomes true, and the right-hand condition eventually holds true.

We further introduce syntactic sugar *weak-until* $\mathcal{W}$ in the following manner: $\varphi_1 \mathcal{W} \varphi_2$ is equivalent to $(\varphi_1 \mathcal{U} \varphi_2) \vee \Box \varphi_1$. A trace τ satisfies a P4LTL specification φ is written as $\tau \models \varphi$. For a P4 program $\mathcal{P}$ and a P4LTL specification φ, $\mathcal{P} \models \varphi$ if and only if all $\mathcal{P}$'s feasible execution traces $\tau \models \varphi$.

CPI. To specify the control plane assumption, developers can leverage terms $key(t,k)$, $para(t,a,p)$ and predicates $hit(t)$, $apply(t,a)$, as exemplified in Section 3.2. As per the semantics of P4LTL, alterations in the control plane amidst a packet sequence do not affect the validity of the verified P4LTL property, as long as the control plane still complies with the symbolic CPI. We integrate the control plane assumption ψ_C along with the property φ_{old} to form the new P4LTL formula $\varphi_{new} = \psi_C \rightarrow \varphi_{old}$ and proceed with the verification.

Example. Consider the liveness property, "Specific packets arriving infinitely often should be forwarded to every valid port infinitely often," introduced previously in Section 2. We can express the property with P4LTL as follows. The environment assumption $\varphi_E = \Box\Diamond(\text{valid}(hdr.label))$ states that we only consider the situation when specific packets come infinitely often. The temporal property $\varphi_P = \Box(\neg \text{valid}(hdr.label) \rightarrow \text{drop}) \wedge \Box\Diamond \text{fwd}(1) \wedge \Box\Diamond \text{fwd}(2) \wedge \Box\Diamond \text{fwd}(3)$ specifies that the specific packets should be forwarded infinitely often to valid ports, while other packets are dropped. In the end, the P4LTL formula $\varphi = \varphi_E \rightarrow \varphi_P$ depicts the desired property.

5 Verification

In this section, we present our approach for temporal verification of the stateful P4 programs. A fundamental challenge arises from the fact that the semantics of P4 programs and the semantics of the P4LTL specification attribute different meanings to the concept of a time step. Consequently, there arises a necessity to align these two distinct time steps within our verification algorithm. To address the issue, we propose a novel concept Büchi transaction, extending the classical program LTL verification approach [20].

5.1 Büchi Transaction

Our approach is rooted in the verification paradigm proposed in [20], which relies on its definition of the Büchi program product.

The concept Büchi program product extends from the concept of *Büchi automaton*, designed to accept infinite words. A Büchi automaton is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \mathcal{F})$, which consists of a finite alphabet Σ, a finite state sets Q with an initial state $q_0 \in Q$, a transition relation $\delta : Q \times \Sigma \rightarrow Q$ and a set of final states $\mathcal{F} \subseteq Q$. Büchi automaton $\mathcal{A}$ accepts a word $w \in \Sigma^\omega$ if its corresponding run on $\mathcal{A}$ contains any final states infinitely often. The word accepted by Büchi automaton can also be called a *fair* trace. In addition, each LTL formula can be transformed into an equivalent Büchi automaton following the well-established algorithm [25].

Definition. (Büchi program product) A program is a tuple $\mathcal{P} = (\mathcal{S}, \mathcal{L}, \ell_0, \delta_p)$ consisting of a statement set $\mathcal{S}$, a program location set $\mathcal{L}$ with program entry ℓ_0 and a transition relation $\delta_p : \mathcal{L} \times \mathcal{S} \rightarrow \mathcal{L}$. Given a specification Büchi automaton $\mathcal{A}_\varphi = (\Sigma, Q, q_0, \delta, \mathcal{F})$, the Büchi program product $\mathcal{P} \times \mathcal{A}_\varphi$ is a Büchi automaton $\mathcal{B} = \left(\widehat{\Sigma}, \widehat{Q}, \widehat{q_0}, \widehat{\delta}, \widehat{\mathcal{F}}\right)$ where:

- $\widehat{\Sigma} = \{s; \text{assume } (\phi) \mid s \in \mathcal{S} \wedge \phi \in \Sigma\}$
- $\widehat{Q} = \{(l,q) \mid l \in \mathcal{L} \wedge q \in Q\}$ and $\widehat{q_0} = (\ell_0, q_0)$
- $\widehat{\delta} = \{((l,q), s; \text{assume}(\phi), (l',q')) \mid (l,s,l') \in \delta_p \wedge (q,\phi,q') \in \delta\}$
- $\widehat{\mathcal{F}} = \{(\ell,q) \mid l \in \mathcal{L} \wedge q \in \mathcal{F}\}$

The definition of Büchi program product is actually the product of two automatons, the program $\mathcal{P}$ and the specification $\mathcal{A}_\varphi$. The product is also a Büchi automaton, which enables acceptance of infinite traces. Informally, one can view the Büchi program product of $\mathcal{P}$ and the specification negation $\mathcal{A}_{\neg\varphi}$ as the Büchi automaton that witnesses the violation, i.e., accepts counterexamples. It has been proven that every

fair and feasible trace of this Büchi program product is a counterexample demonstrating that $\mathcal{P}$ violates the specification φ in [20], where a corresponding checking algorithm is also proposed. A trace is *feasible* if the corresponding sequence of statements is executable. The notion executable means that the *assume* conditions along the execution are all satisfiable.

Yet by definition, Büchi program product only concerns traces at the *small time step* (i.e., at the level of statements), which can not be directly applied to verify P4LTL that concerns the *big time step* (i.e., at the level of packet processing). For brevity, we denote the execution of the P4 program in our execution model as a *transaction*. In P4LTL verification, we can not allow the LTL automaton to transition every time the control flow automaton transitions. The LTL automaton should only take its step to transition whenever the *transaction* of the control flow automaton finishes, in correspondence with the semantics of P4LTL. Thus, we extend Büchi automaton to *Büchi transaction*, aiming to check LTL properties at the packet processing level. Its formal definition is as follows:

Definition (Büchi Transaction) Consider a program $\mathcal{P} = (\mathcal{S}, \mathcal{L}, \ell_0, \delta_p)$. We call it $\mathrm{TE}(s)$/ $\mathrm{TR}(s)$ when $s \in \mathcal{S}$ is an entering/return statement to/from the transaction. Given a specification automaton $\mathcal{A}_\varphi = (\Sigma, Q, q_0, \delta, \mathcal{F})$, the Büchi transaction $\mathcal{P} \otimes \mathcal{A}_\varphi$ is a Büchi automaton $\mathcal{B} = \left(\widehat{\Sigma}, \widehat{Q}, \widehat{q_0}, \widehat{\delta}, \widehat{\mathcal{F}}\right)$ where:

- $\widehat{\Sigma} = \{s; \text{assume } (\phi) \mid s \in \mathcal{S} \wedge \phi \in \Sigma\}$
- $\widehat{Q} = \{(l,q) \mid l \in \mathcal{L} \wedge q \in Q\}$ and $\widehat{q_0} = (\ell_0, q_0)$
- $\widehat{\delta} = \{((l,q), s; \text{assume}\,(\phi), (l',q')) \mid$
  $(\mathrm{TR}(s) \wedge (l,s,l') \in \delta_p \wedge (q,\phi,q') \in \delta) \vee$
  $(\neg \mathrm{TR}(s) \wedge (l,s,l') \in \delta_p \wedge q = q' \wedge \phi = true)\}$
- $\widehat{\mathcal{F}} = \{(\ell,q) \mid q \in \mathcal{F} \wedge (\mathrm{TE}(\ell) \vee \mathrm{TR}(\ell))\}$, where
  - $\mathrm{TE}(\ell) : \exists \ell' \in \mathcal{L}, s \in \mathcal{S}.\ (\ell, s, \ell') \in \delta_p \wedge \mathrm{TE}(s)$,
  - $\mathrm{TR}(\ell) : \exists \ell' \in \mathcal{L}, s \in \mathcal{S}.\ (\ell', s, \ell) \in \delta_p \wedge \mathrm{TR}(s)$.

The Büchi program product and Büchi transaction are both products of a control flow automaton and a specification automaton. The key distinction lies in the asynchronous nature of defining the Büchi transaction product. One can conceptualize the products of two Büchi automatons as their concurrent executions. In the Büchi program product, this manifests as a synchronized execution, where each step taken by the control flow automaton corresponds to a step by the specification automaton, and vice versa.

In the context of Büchi transaction, the behavior is distinct. As per the transition definition $\widehat{\delta}$, the LTL automaton advances only when a *transaction step* occurs. A "transaction step" is based on the predicate $\mathrm{TR}(s)$, which indicates whether the statement s is a "transaction return" statement. The LTL automaton can only progress when $\mathrm{TR}(s)$ is satisfied, whereas the control flow automaton can advance without additional constraints. It is worth noting that while Büchi transaction resembles Büchi program product, they diverge in the definition of transition and final state. To be specific, since P4LTL is defined at the big time step, only a "transaction return" triggers the progression in the LTL automaton, and only states corresponding to a "transaction entering/return" are labeled as final states $\widehat{\mathcal{F}}$. This extension is of importance, for it enables verification at the transaction level semantics (i.e., P4LTL semantics) and substantially reduces the number of transitions and accepted traces that are semantically irrelevant to P4LTL.

Theorem 5.1. *The P4 program $\mathcal{P}$ satisfies the P4LTL specification φ if and only if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace.*

The proof of the theorem resembles the proof for Büchi program product discussed in [20]. Therefore, we refer the interested readers to the appendix for further details.

Example. To better illustrate the distinction between Büchi transaction and Büchi program product, consider the pseudocode and property shown in Figure 5. The snippet aims to decrease the packet field *hdr.cnt* by one and set the register *status* to zero if *hdr.cnt* is less than zero, or one otherwise at every packet processing. For brevity, we denote *hdr.cnt* as x and *status* as y. In the Büchi program product, each step in the control flow automaton corresponds to a step in the LTL automaton, as depicted in Figure 6. In contrast, for the Büchi transaction, edges $(l_0,q_0) \rightsquigarrow (l_1,q_1)$ and $(l_1,q_0) \rightsquigarrow (l_2,q_1)$ are removed, and assumptions not from $(l_2,*)$ to $(l_0,*)$ all become true. This is because only edges from l_2 to l_0 correspond to a step in the LTL automaton. The formula $\Diamond(x > 0) \rightarrow \Diamond(y = 0)$ does not hold under the definition of Büchi program product, as evidenced by a counterexample where x is consistently assigned a value of 1 at the loop front. That is, $x > 0$ eventually holds, but $y = 0$ never becomes true. However, the formula is satisfied under the notion of Büchi transaction, since only edges from l_2 to l_0 will trigger progression of the LTL automaton. As a result, the Büchi transaction has no fair and feasible trace, indicating the validity of the property. Intuitively, $x > 0$ is always accompanied by $y == 0$ at the end of every *transaction*, but not necessarily for every *statement*. Furthermore, the greater concision of Büchi transaction compared to the Büchi program product in Figure 6 and Figure 7 visually demonstrates that Büchi transaction reduces the number of accepted traces that are semantically irrelevant to P4LTL.

5.2 Verification Algorithm

Based on Theorem 5.1, the temporal verification of stateful P4 programs is reduced to checking whether the Büchi transaction has any fair and feasible trace. Note that the Büchi transaction is a special case of Büchi automaton. Therefore, we can leverage the existing verification algorithm for the Büchi automaton proposed in [20].

Algorithm 1 illustrates our verification framework. It takes a program $\mathcal{P}$ and a LTL formula φ as inputs, and outputs

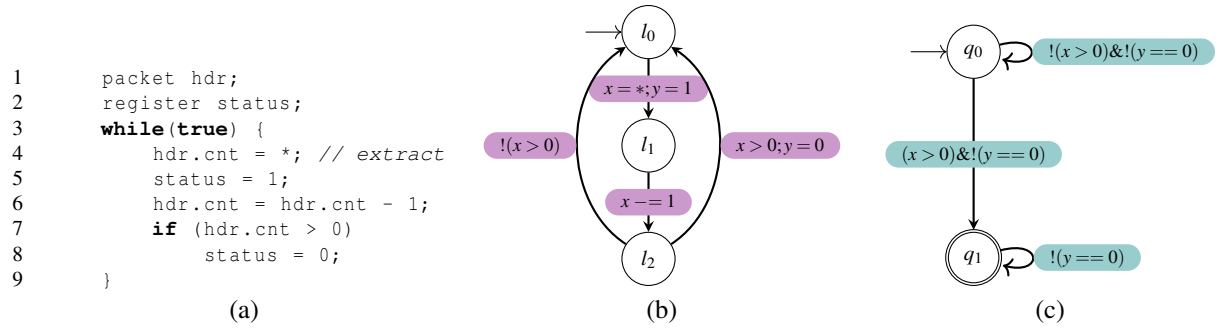


Figure 5: The program is presented in (a) as code and in (b) as control flow automaton. For brevity, we denote *hdr.cnt* as x and *status* as y. The Büchi automata representing the negation of $\Diamond(x > 0) \rightarrow \Diamond(y = 0)$ is presented in (c).

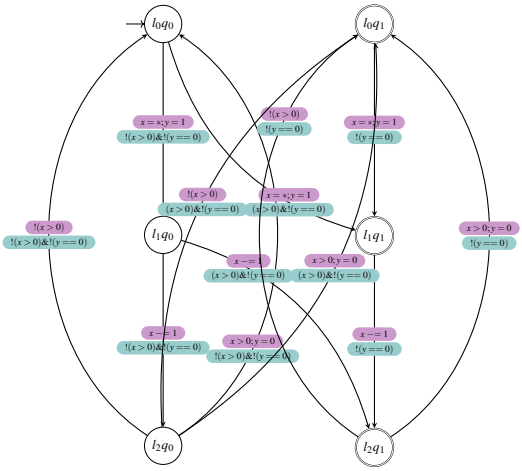


Figure 6: Büchi program product for Figure 5.

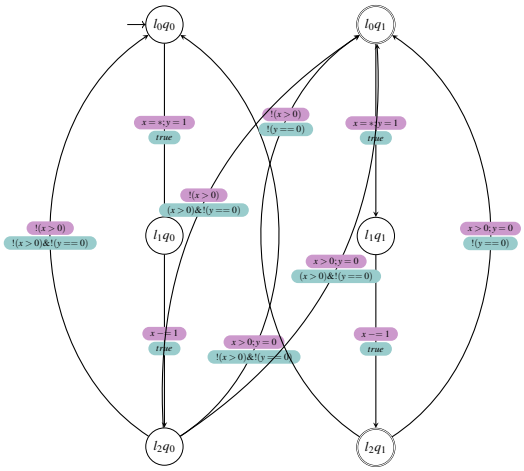


Figure 7: Büchi transaction for Figure 5.

Algorithm 1 The verification framework

Input: Program $\mathcal{P}$ and LTL property φ
Output: *Yes* if $\mathcal{P} \models \varphi$, and counterexample π otherwise

1: $\mathcal{A}_{\neg\varphi} \leftarrow$ LTL2Buchi$(\neg\varphi)$
2: $\mathcal{B} \leftarrow \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$
3: **while** $\mathcal{L}(\mathcal{B}) \neq \emptyset$ **do**
4: Let $\pi \in \mathcal{L}(\mathcal{B})$
5: **if** π is feasible **then**
6: **return** π
7: $\mathcal{B} \leftarrow$ refine$(\mathcal{B}, \pi)$
8: **return** *Yes*

the verification result. In the first place, the negation of the LTL formula $\neg\varphi$ is transformed into its corresponding Büchi automaton $\mathcal{A}_{\neg\varphi}$ (line 1). Next, the Büchi transaction is constructed to represent the counterexamples that witness the violation (line 2). The while loop (lines 4-7) aims to identify any fair and feasible trace that belongs to $\mathcal{B}$. First, it extracts a fair trace π that belongs to $\mathcal{B}$ (line 4). If π is feasible, then a counterexample is found according to Theorem 5.1. Otherwise, $\mathcal{B}$ is reduced to not accept π and other spurious traces that can be generalized from it (line 7). Finally, if there exists no fair trace for $\mathcal{B}$, the framework concludes that the property φ has been verified (line 8).

6 Implementation

The implementation of p4tv comprises approximately 10,000 lines of code in C++ and Java. As a common approach adopted by P4 verifiers [24, 40, 41, 47], we first translate P4 to an intermediate representation called Boogie [7] and proceed with the downstream verification. Next, we instrument the program with ghost variables to keep track of the terms and predicates introduced in P4LTL, similar to the approach used in p4v [40]. Our verification approach is implemented as an extension to Ultimate Automizer model checker [20]. Given a P4LTL specification, our implementation constructs the corresponding Büchi transaction introduced in Section 5 and checks for the existence of a fair and feasible trace. If the trace is discovered, the corresponding counterexample will be reported to the developer. Otherwise, p4tv will confirm that the P4LTL property is satisfied by the given P4 program.

Limitation. Currently, we note several limitations with

Name	Benchmark Description
Bfs [45]	The fast failover mechanism in a Bfs fashion to reroute packets without control plane interference.
Blink [28]	Detect characteristic failure based on information aggregated with registers and trigger rerouting.
CoDel [36]	An active queue implementation for RFC 8289.
Dfs [45]	A Dfs variation of the Bfs benchmark.
Ndn [46]	A named data networking paradigm where data are retrieved based on names rather than addresses.
P4NIS [38]	Several mechanisms for eavesdropping protection.
P4sp [37]	A sensor failure recovery system.
P4xos-acceptor [19]	P4 implementation of the acceptor in Paxos protocol.
P4xos-learner [19]	P4 implementation of the learner in Paxos protocol.

Table 1: P4 benchmark descriptions.

p4tv. Firstly, while it fully supports V1Model, it offers only limited support for TNA and PSA. Nonetheless, as demonstrated in Section 3.1, we assert that our modeling techniques remain applicable to these architectures, requiring only a more intricate design in the `P4program` process. We plan to support them in the future. Additionally, although the utilization of Büchi transactions can decrease the number of accepted traces, p4tv may still encounter the state explosion problem, which is an intrinsic challenge for model checking. Lastly, p4tv only permits developers to specify properties at the packet processing boundaries. Specifically, developers cannot directly specify or verify properties within packet processing. As per the P4LTL semantics, if a property violation occurs within packet processing but remains unobservable at the boundary, p4tv does not regard such violations as an issue.

7 Evaluation

Dataset. As p4tv pioneers temporal verification for stateful P4 programs, we introduce a dedicated dataset tailored for this purpose. This dataset comprises nine non-trivial stateful P4 programs sourced from both the open-source community and academic papers (further details provided in Table 1). For each program, we carefully analyze its design proposal and manually formulate temporal properties that we anticipate to be satisfied. Furthermore, besides several bugs that have been found in our evaluation, we intentionally inject additional artificial bugs into the control plane or code implementation to further test p4tv's bug-finding capabilities. In total, we formulate 19 stateful verification tasks.

P4 verifiers. We compare p4tv with bf4 [21], p4v [40], Vera [47], and Aquila [50], representing the state-of-the-art P4 verifiers in recent years.

All experiments were conducted on a machine with an Intel(R) Core(TM) i5-12400 2.50 GHz CPU and 32GB RAM.

7.1 Verifying Stateful Temporal Properties

Table 2 presents the evaluation results of our tool, including the specification size and detailed description of verified properties. The CPI column indicates whether the corresponding verification task requires additional control plane information. The number with an asterisk superscript denotes an artificial bug injection. Symbol ✓ represents the successful verification, and ✗ denotes a counterexample revealing a discrepancy between program behavior and the property.

Effectiveness. All of the compared verifiers (i.e., bf4 [21], p4v [40], Vera [47] and Aquila [50]) currently do not support the temporal verification of stateful P4 programs at the packet processing level. These verifiers assume the register values to be non-deterministic at the start of each packet processing, overlooking its stateful nature. This limitation restricts their ability to specify and verify temporal properties related to register state changes and history traces.

Of all the 19 stateful verification tasks, p4tv successfully verifies 14, while reporting counterexamples for the remaining. In the 9th verification task, we discover that insufficient consideration for register initialization in the P4 implementation leads to property failure. According to the P4-16 specification [16], the initial register value is non-deterministic without manual specification, potentially causing constant packet dropping. Concerning the 13th verification task, the improper implementation of the heartbeat check for the primary port results in property violation. Moreover, p4tv effectively reveals bugs through corresponding counterexamples for manual bug injections in both CPI (task 8) and codes (tasks 15 and 18). In task 8, swapping the matching action for data packet and interest packet in CPI causes a resource interest packet to reset, rather than set, the "can be forwarded" status for the corresponding data packet. P4tv identifies the violation and reports a packet processing sequence where the switch receives an interest packet followed by a data packet but fails to forward the latter. For tasks 15 and 18, p4tv helps pinpoint the injected bugs by revealing counterexamples where the port state and round registers are incorrectly updated.

Efficiency. The verification time of p4tv ranges from 10 seconds to 21 minutes, with median and average time costs of 51 seconds and 181 seconds, respectively. Memory usage ranges from 201MB to 6.1GB, with violation detection tasks tending to require more memory. Some verification tasks, such as task 8, require relatively longer verification time, attributed to numerous refinement iterations. Generally, the inherent complexity of verification tasks primarily determines the solving time. Moreover, while larger programs and specifications often lead to more complex verification tasks, it is not always the case, e.g., tasks 5 and 16.

7.2 Verifying Assertions

This experiment compares p4tv with prior tools for assertion verification. An assertion ψ is a Boolean predicate associated with a specific P4 program point, that should always evaluate to true at that point during code execution [27]. Each assertion corresponds to a particular case of P4LTL property described

Benchmark	No.	Property Description	CPI	Prog Size	Spec Size	p4tv
Bfs	1	Reroute packets are sent to parent ports only after the whole route process is complete.		355	34	✓ (51s, 1.6GB)
Blink	2	The packets are retransmitted only after the failure signal is detected over multiple flows.		745	44	✓ (154s, 1.8GB)
CoDel	3	If in normal state, the switch will not drop packets until the queuing delay exceeds the threshold.		285	40	✓ (51s, 497.9MB)
	4	If in drop state, packets will be dropped until the queuing delay resumes.			33	✓ (18s, 286.3MB)
Dfs	5	Packets are sent to parent ports only after the whole route process is complete.		306	34	✓ (47s, 201.3MB)
Ndn	6	Forwarding data packets occurs only if the switch has received corresponding interest packets before.	✓	470	50	✓ (895s, 1.4GB)
	7	After forwarding data packets, the arrival of an immediate data packet entails drop.	✓		34	✓ (274s, 1.8GB)
	8*	Bug inject: swapping matching action for data packet and interest packet in CPI. Check property 6.	✓		50	✗ (1260s, 6.1GB)
P4NIS	9	If user packets arrive infinitely often, then they will be forwarded to certain ports infinitely often.		303	82	✗ (185s, 2.7GB)
	10	Egress ports for packets from the same source are not fixed.			40	✓ (39s, 1.8GB)
P4sp	11	When timeout is detected, the switch will forward packets from the redundant port until the primary port recovers.		222	64	✓ (14s, 623.8MB)
	12	If a packet from the primary port is forwarded, packets below the threshold will be dropped next time.			38	✓ (22s, 836.0MB)
	13	Early packets from the secondary port are forwarded only if the time threshold has been exceeded before.			38	✗ (64s, 1.7GB)
	14	If a packet from the primary port arrives, the forwarding decision will depend on the corresponding timestamp until the arrival of the next primary port packet.			39	✓ (10s, 513.1MB)
	15*	Bug inject: The state update action is interchanged between primary pory and secondary port. Check property 12.		231	38	✗ (85s, 1.9GB)
Paxos-acceptor	16	After a p4xos packet is forwarded, the p4xos packets of earlier rounds will not be forwarded by the switch.	✓	355	66	✓ (45s, 857.9MB)
	17	The acceptor switch only votes for the value it stores until a new election packet arrives.			31	✓ (28s, 719.1MB)
	18*	Bug inject: Wrongly updating the value round instead of the current valid round for round register. Check property 17.			31	✗ (176s, 2.8GB)
Paxos-learner	19	The learner switch will not forward votes until the vote has reached majority agreements.		322	43	✓ (21s, 396.5MB)

Table 2: Results of stateful temporal verification.

by $\square\psi$, indicating that "the predicate ψ should hold at the end of every packet processing." We can use p4tv to verify assertion by instrumenting a shadow variable to track the assertion's value at that point and initializing it with *true* at the beginning of each packet processing stage.

Given Vera's limited support for user-defined assertions and Aquila's not publicly accessible, we contrast p4tv with bf4 and p4v, the current state-of-the-art P4 verifiers. We utilize the publicly available code for bf4 and employ the approximate implementation for p4v in [21], as p4v is not open-sourced. Furthermore, it's important to note that while assertions are supported by prior tools, they operate under a model that assumes all register values to be nondeterministic and only considers a single packet processing. In the subsequent discussion, an assertion is deemed *stateful* (or *stateless*) based on whether it is related to registers or not.

Table 3 presents the results of the assertion verification. The symbol ✓ indicates successful validation of the property, ✗ signifies a genuine counterexample, and Δ denotes that the tool returns a spurious counterexample, which would not occur under the given configuration. SimpleNat is a simple stateless NAT P4 implementation. For conciseness, the results of all six invalid header access checks are combined into one line since their verification tasks are nearly identical as well as their verification time. The results of valid header access checks are similar.

As depicted in Table 3, p4tv is able to verify stateless assertions as the priors do. It's worth noting that while its performance seems relatively degraded, this is attributable to the fact that p4tv evaluates assertions within the multiple-packet model, introducing additional complexity to the verification process. Regarding stateful assertions, p4tv effectively con-

Benchmark	Assertion	Stateful?	p4tv	bf4	p4v
SimpleNat-bug1_6	Whether the header is valid before accessing.	No	✗ (2m)	✗ (3s)	✗ (4s)
SimpleNat-safe1_11			✓ (10s)	✓ (3s)	✓ (4s)
P4NIS-safe1	The egress port should be equal to the register value if tunnel is applied.	Yes	✓ (2m36s)	✓ (3s)	✓ (2s)
P4NIS-safe2	Labeled packets cannot be implicitly dropped.	Yes	✓ (59s)	Δ(3s)	Δ(3s)
P4xos-acceptor-safe	Stored consensus round must be always less than the current valid round.	Yes	✓ (28m21s)	Δ(4s)	Δ(3s)
P4xos-learner-unsafe	The value of vote counter register is always zero.	Yes	✗ (6m54s)	Δ(2s)	Δ(3s)
P4xos-acceptor-unsafe	Stored consensus round must be always equal to the current valid round.	Yes	✗ (3m40s)	Δ(3s)	Δ(2s)

Table 3: Results of assertion verification.

firms safe assertions and reports genuine counterexamples for unsafe ones. Conversely, bf4 and p4v, assuming registers to be entirely non-deterministic, exhibit incompleteness in verifying assertions under specific register configurations. For instance, under the zeroing register configuration, the assertion $vround[hdr.inst] \leq round[hdr.inst]$ for P4xos-acceptor-safe remains valid throughout subsequent processing, but may easily fail under the non-deterministic configuration. Furthermore, for benchmarks like P4xos-learner-unsafe, although bf4 and p4v successfully invalidate the assertion, they reported counterexamples that would never occur under the given configuration. In conclusion, the overapproximation of the priors results in imprecision in verifying stateful properties.

7.3 Scalability Evaluation

In this section, we explore the scalability of p4tv in two dimensions: program complexity and specification complexity.

To assess the scalability of p4tv concerning program complexity, we introduce a parameter named PORT_NUM for P4NIS to parameterize the range of valid ports. We verify the following valid P4LTL property: if the register *count* is set within PORT_NUM, it will remain within this range for all subsequent processing. Since P4 is a loop-free language, increasing PORT_NUM by 1 entails adding an additional if block to the program that updates the *count* register and sets the metadata, without expanding the size of the P4LTL property. The verification results are presented in Figure 8.

To examine the scalability of p4tv concerning specification complexity, we conduct experiments on P4xos-acceptor. An essential property of the acceptor is that the round recording the consensus value must always be less than or equal to the current valid round, as expressed by the P4LTL property:

$$\wedge_{i=0}^{\mathrm{CNT}} vround[i] \leq round[i] \rightarrow \Box(vround[i] \leq round[i]).$$

This ensures that outdated messages are disregarded, which is crucial for achieving consensus. We verify P4xos-acceptor with the property as the parameter CNT increases. The verification results are illustrated in Figure 9.

As indicated in Figure 8 and Figure 9, with the other dimension fixed, the verification time of p4tv exhibits rapid growth as the complexity factors PORT_NUM and CNT become relatively large. Moreover, the validity of the property also impacts the verification time. For instance, manually substituting $\leq$ with $>$ in every clause of the P4xos-acceptor specification, the time required to return a counterexample remains nearly identical for CNT $= 8$ and CNT $= 16$ (in about 90s), since constructing a counterexample trace is straightforward. Ultimately, the root cause of verification time is the intrinsic complexity of the verification task.

8 Case Studies

P4tv can verify a wide range of temporal properties within stateful P4 programs. In this section, we present our verifying experiences with p4tv through some case studies.

P4NIS [38]. Processors-based network immune scheme (P4NIS) aims to protect the network against packet eavesdropping within the programmable data plane. The snippet of P4NIS is presented previously in Section 2. An important liveness property associated with the defense mechanism can be expressed as follows:

If specific packets keep arriving and being forwarded, they should be forwarded to ports 1, 2 and 3 infinitely often.

We can use P4LTL to concisely express the temporal property. Firstly, it is necessary to define the term "specific packets" within the context of P4NIS. These packets are identified as those with an IPv4 destination address other than the domain name server (123.123.123.123) and received from port 0. Moreover, the environment assumption in P4LTL can be leveraged to describe the "If" part of the property. Specifically, the environment assumption can be formulated as $\varphi_E = \Box\Diamond\phi_{sp}$, where

$$\phi_{sp} = standard_metadata.ingress_port == 0 \\ \wedge \mathrm{old}(hdr.ipv4.dstAddr) \neq 123.123.123.123,$$

denoting the "specific" condition in the context of P4NIS mechanism. Moreover, we utilize ϕ_{sp} to specify the subsequent portion of the property, which states that "they should be forwarded to ports 1, 2 and 3 infinitely often" as follows:

$$\varphi_P = \bigwedge_{i=1}^{3} \Box\Diamond(\phi_{sp} \rightarrow \mathrm{fwd}(i))$$

In summary, we can use the formula $\varphi = \varphi_E \rightarrow \varphi_P$ to formalize the temporal property of such stateful functionalities. However, the property φ does not hold. Upon inputting φ

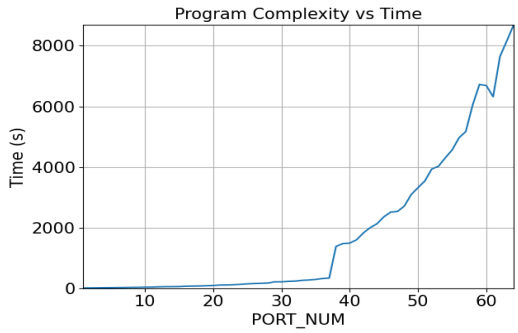


Figure 8: Verification time curve as PORT_NUM grows.

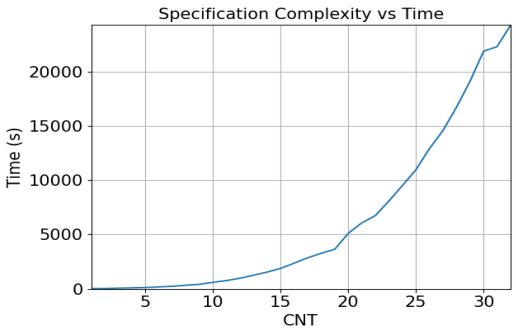


Figure 9: Verification time curve as CNT grows.

and the P4NIS code, p4tv will report a counterexample indicating that the value of the port record register `count[0]` is improperly configured. The oversight of register configuration leads to the potential of a packet skipping all the conditional branches and being implicitly dropped (i.e., the forwarding port *egress_spec* is not set). The intrinsic issue is that the P4 language specification [16] assumes that the default value of the register `count[0]` is undefined. Consequently, a switch deployed with the P4NIS mechanism may drop all the receiving packets when `count[0]` is configured with a value outside the range $[0,2]$, e.g., configured with 4.

To fix the potential vulnerability, we can adopt two strategies regarding different aspects. Suppose we are not certain about whether the underlying hardware performs the default zero configuration. In that case, we can amend the code with an additional branch condition, resetting `count[0]` when it is improperly configured.

```
1       count.read(temp,(bit<32>)0);
2       // set count back to normal value
3       if(temp > 2){
4           count.write((bit<32>)0, (bit<32>)1);
5           standard_metadata.egress_spec = 1;
6       }
```

If we have a certain confidence in the hardware configuration, the property part φ_P becomes

$$\square(count[0] \geq 0 \wedge count[0] \leq 2 \rightarrow \bigwedge_{i=1}^{3} \square\Diamond(\phi_{sp} \rightarrow \text{fwd}(i))),$$

which states that when `count` is appropriately configured, packets will be forwarded to ports 1,2 and 3 in a circular way. Either application of the strategies will enable p4tv to verify the correctness of the specification, proving that the temporal property is satisfied by P4NIS.

NDN [46]. Named-Data Networking (NDN) implements a content-centric networking paradigm that aims to provide an efficient and scalable way to manage distributed storage, as opposed to the typical destination-centric approach.

NDN employs two packet types, interest and data, to implement its content-centric paradigm. Interest packets represent requests, while data packets represent responses. If the request name in an interest packet is not matched in the Forwarding Information Base (FIB) table, the switch will drop the packet. Conversely, if the request name is present in the FIB, the switch will store the incoming port information in the Pending Interest Table (PIT) and forward the packet based on the port information stored in the FIB.

If a data packet experiences a miss in the Pending Interest Table (PIT), the switch will drop it. This occurs because the corresponding pending request is not logged. Conversely, if the data packet matches an entry in the PIT, the switch will forward the packet through the port where the request packet originates from and remove the entry. Both the Forwarding Information Base (FIB) and Pending Interest Table (PIT) are using registers as keys.

Based on the NDN implementation, its property φ_{ndn} can be formulated in P4LTL as

$$\square(\phi_{data} \wedge \neg\text{drop} \rightarrow \circ((\phi_{data} \rightarrow \text{drop})\,\mathcal{W}(\phi_{int} \wedge \neg\text{drop}))),$$

where $meta.nameHash == a \wedge meta.packetType == DATA$ is represented by ϕ_{data}, and ϕ_{int} denotes $meta.nameHash == a \wedge meta.packetType == INT$.

The specification φ_{ndn} states that, for any given name hash a, the switch will always drop the data packet until it forwards the corresponding request packet. Note that a serves as a free variable, indicating that the specification is expected to hold for any name hash. Additionally, we use $\phi_{data} \rightarrow$ drop to specify that we only expect the specification to hold after a data packet has been forwarded, since the corresponding PIT entry is cleared afterwards.

Upon inputting the specification and the NDN code, p4tv returns a counterexample. The reason for this outcome is that the table entries control a part of the key forwarding logic. The property does not hold if we assume that the table entries are arbitrary. To address the problem, we employ CPI to complement the forwarding logic, stating that if the requesting data is not recorded in PIT or FIB, the tables `routeData_table` and `updatePit_table` should apply the action drop, and otherwise the table `updatePit_table` should update the PIT table accordingly, as shown below:

$$\begin{aligned}&\square(\text{apply}(ut) \wedge \text{key}(ut, meta.FIB) == 0 \rightarrow \text{apply}(ut, drop)\\ &\wedge \text{apply}(ut) \wedge \text{key}(ut, meta.FIB) \neq 0 \rightarrow \text{apply}(ut, update)\\ &\wedge \text{apply}(rt) \wedge \text{key}(rt, meta.PIT) == 0 \rightarrow \text{apply}(rt, drop))\end{aligned}$$

After inputting the CPI along with the NDN implementation, p4tv verifies the correctness of the property.

9 Related Work

SDN verification. The verification for SDN can be classified into control plane and data plane verification.

Traditionally, studies analyzing the control plane have focused on topological properties such as reachability and isolation. Arc [26] facilitates rapid control plane verification without the data plane generation. Batfish [23] introduces a general approach to detect errors in network configurations by deriving the data plane effectively. CrystalNet [39] offers a proactive means of validating network operations using emulation techniques. Minesweeper [8] converts network configurations into logical formulas and checks for satisfiability to identify any network state that violates the specified query. These studies primarily analyze network configurations and strive to address queries related to network-wide properties.

Data plane verification tools address similar queries by requiring a snapshot of the network's data plane. Numerous techniques have been proposed to verify the data plane from various perspectives. Anteater [42] employs graph theory algorithms to verify network-wide properties, representing the network as a graph. HSA [33] develops an optimized formalism to statically check network configurations with specifications and identify network failures. Veriflow [34] aims to enable real-time validation of network-wide invariants, while Symnet [48] validates network-wide queries by injecting symbolic packets and tracking their behavior through the network. Our tool, p4tv, falls within the category of data plane verification tools and is specifically designed for verifying processing-level temporal properties of stateful P4 programs.

P4 verification tools. Aquila [50], bf4 [21] and p4v [40] encode P4 programs with SMT formulas and verify the properties using constraint solving. ASSERT-P4 [24] and Vera [47] use symbolic execution to verify P4 programs. However, these P4 verifiers overlook the stateful nature of packet processing. They assume register states are nondeterministic before each packet processing, which fundamentally hinders them from verifying the behaviors of stateful P4 programs. Moreover, Aquila, ASSERT-P4, bf4, and bf4 use Hoare logic and Vera proposes NetCTL that is a subset of temporal logic language CTL [13]. Both Hoare logic and NetCTL focus on properties of an individual packet processing and do not consider historical information. On the contrary, p4tv considers the preservation of register states in its execution model, and provides P4LTL to support processing-level temporal specification.

Stateful network verification tools. Several studies have previously addressed the verification of stateful networks, but they have focused on different aspects compared to p4tv. For instance, [54] concentrates on the behaviors of match-action tables in SDN, while [5] verifies congestion control protocols by transforming network models into SMT formulas. Previous works such as [1, 2, 52] utilize abstract interpretation to verify stateful networks, with a focus on network-level properties such as isolation. These works fall under the category of control plane verification. In contrast, p4tv targets data plane verification with its focus on P4. Aragog [53] aims to verify distributed network functions, but is limited to verifying network invariants and run-time checks. P4wn [31] is a symbolic execution tool for stateful P4 programs that reports the probability of each program path. In our work, we adopt a similar approach as P4wn, treating stateful P4 programs as infinite loops. However, we enhance this approach by incorporating additional register initializations to adapt it to our specific verification scenario. Notably, our research question differs from P4wn as our primary objective is *formal verification* rather than *statistical testing* for P4 programs.

Temporal verification. There have been a number of approaches to verifying the temporal properties of state-based systems [6, 12, 14, 15, 17, 18, 20, 29]. Several approaches [14, 15, 17, 18, 20] employ counterexample-guided abstraction refinement to verify temporal properties. They refine the abstract model iteratively utilizing spurious counterexamples encountered until no counterexample exists or a real counterexample is discovered. In our verification approach, we adopt the framework proposed by [20]. However, it is important to note that our approach differs in several aspects. Firstly, we aim at different semantics for temporal properties. Instead of verifying properties at the small time step, we focus on checking them at the big time step, which limits the direct application of the common approach to P4LTL. Moreover, for verification purposes, we construct Büchi transaction rather than Büchi program product. This imposes additional constraints on the transitions and final states, resulting in a reduced number of transitions and accepted traces and making the verification more practical.

10 Conclusions

Current P4 verifiers simplify P4 verification by overapproximating register states, which leads to challenges in specifying and verifying complex stateful P4 programs. We present p4tv, the first automated tool for temporal verification of stateful P4 programs at the packet processing level. We introduce a specification language called P4LTL to describe stateful functionalities and formulated Büchi transaction to make the verification more tractable. The evaluation of verification tasks demonstrates p4tv's effectiveness in verifying temporal properties of stateful P4 programs. P4tv is available at https://thufv.github.io/research/p4tv.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

References

[1] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Modular safety verification for stateful networks. *CoRR*, abs/1708.05904, 2017.

[2] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Abstract interpretation of stateful networks. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2018.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: semantic foundations for networks. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 113–126. ACM, 2014.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: stateful network-wide abstractions for packet processing. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 29–43. ACM, 2016.

[5] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 1–16. ACM, 2021.

[6] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. Divine 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer, 2013.

[7] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 155–168. ACM, 2017.

[9] Cristian Hernandez Benet, Andreas J. Kassler, Theophilus Benson, and Gergely Pongrácz. MP-HULA: multipath transport aware load balancing using programmable data planes. In Xin Jin and Changhoon Kim, editors, *Proceedings of the 2018 Morning Workshop on In-Network Computing, NetCompute@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*, pages 7–13. ACM, 2018.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.

[11] Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, and Jennifer Rexford. A NICE way to test openflow applications. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 127–140. USENIX Association, 2012.

[12] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July*

15-19, 2000, Proceedings, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[15] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[16] P4 Langhuage Consortium. $P4_{16}$ language specifcation, 2023.

[17] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 265–276. ACM, 2007.

[18] Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 399–410. ACM, 2011.

[19] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, 2020.

[20] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness modulo theory: A new approach to LTL software model checking. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015.

[21] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free P4 programs. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 571–585. ACM, 2020.

[22] Nick Feamster, Jennifer Rexford, and Ellen W. Zegura. The road to SDN: an intellectual history of programmable networks. *Comput. Commun. Rev.*, 44(2):87–98, 2014.

[23] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 469–483. USENIX Association, 2015.

[24] Lucas Freire, Miguel C. Neves, Lucas Leal, Kirill Levchenko, Alberto E. Schaeffer Filho, and Marinho P. Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018*, pages 4:1–4:7. ACM, 2018.

[25] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[26] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 300–313. ACM, 2016.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[28] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 161–176. USENIX Association, 2019.

[29] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[30] Enio Kaljic, Almir Maric, Pamela Njemcevic, and Mesud Hadzialic. A survey on data plane flexibility and programmability in software-defined networking. *IEEE Access*, 7:47804–47840, 2019.

[31] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. Probabilistic profiling of stateful data planes for adversarial testing. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 286–301. ACM, 2021.

[32] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: scalable load balancing using programmable data planes. In Brighten Godfrey and Martín Casado, editors, *Proceedings of the Symposium on SDN Research, SOSR 2016, Santa Clara, CA, USA, March 14 - 15, 2016*, page 10. ACM, 2016.

[33] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 113–126. USENIX Association, 2012.

[34] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 15–27. USENIX Association, 2013.

[35] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(268-272):30, 1994.

[36] Ralf Kundel, Jeremias Blendin, Tobias Viernickel, Boris Koldehofe, and Ralf Steinmetz. P4-codel: Active queue management in programmable data planes. In *IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2018, Verona, Italy, November 27-29, 2018*, pages 1–4. IEEE, 2018.

[37] Steffen Lindner, Marco Häberle, Florian Heimgaertner, Naresh Nayak, Sebastian Schildt, Dennis Grewe, Hans Löhr, and Michael Menth. P4 in-network source protection for sensor failover. In *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, pages 791–796. IEEE, 2020.

[38] Gang Liu, Wei Quan, Nan Cheng, Ning Lu, Hongke Zhang, and Xuemin Shen. P4NIS: improving network immunity against eavesdropping with programmable data planes. In *39th IEEE Conference on Computer Communications, INFOCOM Workshops 2020, Toronto, ON, Canada, July 6-9, 2020*, pages 91–96. IEEE, 2020.

[39] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 599–613. ACM, 2017.

[40] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. p4v: practical verification for programmable data planes. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 490–503. ACM, 2018.

[41] Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in p4 networks. *Technical Report, Tech. Rep*, 2016.

[42] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In Srinivasan Keshav, Jörg Liebeherr, John W. Byers, and Jeffrey C. Mogul, editors, *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 290–301. ACM, 2011.

[43] Rui Miguel, Salvatore Signorello, and Fernando M. V. Ramos. Named data networking with programmable switches. In *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*, pages 400–405. IEEE Computer Society, 2018.

[44] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[45] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. Supporting emerging applications with low-latency failover in P4. In Dipankar Raychaudhuri and Richard Li, editors, *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, NEAT@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*, pages 52–57. ACM, 2018.

[46] Salvatore Signorello, Radu State, Jérôme François, and Olivier Festor. Ndn.p4: Programming information-centric data-planes. In *IEEE NetSoft Conference and Workshops, NetSoft 2016, Seoul, South Korea, June 6-10, 2016*, pages 384–389. IEEE, 2016.

[47] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with vera. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM*

Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 518–532. ACM, 2018.

[48] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 314–327. ACM, 2016.

[49] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. D2R: dataplane-only policy-compliant routing under failures. *CoRR*, abs/1912.02402, 2019.

[50] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: a practically usable verification system for production-scale programmable data planes. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 17–32. ACM, 2021.

[51] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, page 1–22, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[52] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 811–830. Springer, 2016.

[53] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 701–718. USENIX Association, 2020.

[54] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani, and Aditya Akella. Liveness verification of stateful network functions. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 257–272. USENIX Association, 2020.

[55] Xiaoquan Zhang, Lin Cui, Kaimin Wei, Fung Po Tso, Yangyang Ji, and Weijia Jia. A survey on stateful data plane in software defined networks. *Comput. Networks*, 184:107597, 2021.

Appendix

A Theorem Proof

Theorem A.1. *The P4 program $\mathcal{P}$ satisfies the P4LTL specification ϕ if and only if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace.*

Proof. To begin with, we introduce the general idea of the theorem at the high level. The Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ corresponds to the automaton that represents the set of every *counterexample*. In other words, every fair and feasible trace of $\mathcal{B}$ corresponds to a counterexample indicating that $\mathcal{P} \not\models \varphi$. Next, we present the sketch of the theorem proof.

For every trace $\pi = (s_0, \text{assume } a_0)\ (s_1, \text{assume } a_1)\ \cdots$ that is fair and feasible to $\mathcal{B}$, we can extract the statement part $\pi_{\mathcal{P}} = s_0\ s_1 \cdots$ and the LTL part $\pi_{\mathcal{A}} = a_0\ a_1 \cdots$. For the LTL part, we can reduce $\pi_{\mathcal{A}}$ to $\pi'_{\mathcal{A}}$ by deleting the *true* characters imposed by Büchi transaction, and use the definition of Büchi transaction to show that π is fair and feasible to $\mathcal{B}$ if and only if $\pi_{\mathcal{P}}$ is fair and feasible to $\mathcal{P}$, $\pi'_{\mathcal{A}}$ belongs to $\mathcal{A}_{\neg\varphi}$ and $\pi_{\mathcal{P}} \models \pi'_{\mathcal{A}}$.

Moreover, we prove that P4 program $\mathcal{P}$ satisfies the P4LTL specification ϕ if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace. We prove it by contradiction. If it were false, then $\mathcal{P} \models \varphi$ and there exists an fair and feasible trace of $\mathcal{B}$. Therefore, we know that there also exists $\pi_{\mathcal{P}}$ and $\pi_{\mathcal{A}}$ such that $\pi_{\mathcal{P}}$ is fair and feasible to $\mathcal{P}$, $\pi'_{\mathcal{A}}$ belongs to $\mathcal{A}_{\neg\varphi}$ and $\pi_{\mathcal{P}} \models \pi'_{\mathcal{A}}$. According to the semantics of P4LTL, $\pi_{\mathcal{P}}$ can serve as a witness to the violation of φ since the execution conform to the formula $\neg\varphi$, and therefore leads to a contradiction.

Furthermore, we prove that P4 program $\mathcal{P}$ satisfies the P4LTL specification ϕ only if the Büchi transaction $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have any fair and feasible trace. Assume that $\mathcal{B}$ does not have any fair and feasible trace while $\mathcal{P}$ does not satisfy the P4LTL specification ϕ. According to the semantics of P4LTL, there must exists a fair and feasible trace $\pi_{\mathcal{P}} = s_0\ s_1 \cdots \in \mathcal{P}$ such that $\pi_{\mathcal{P}} \models \neg\varphi$. Moreover, there also exists a corresponding $\pi'_{\mathcal{A}} = a_0\ a_1 \cdots \in \mathcal{A}_{\neg\varphi}$ that witnesses the violation. We then reduce $\pi_{\mathcal{P}}$ to $\pi'_{\mathcal{P}} = s'_0\ s'_1 \cdots$ by deleting statements that do not satisfy $\text{TE}(s) \vee \text{TR}(s)$, and aligns

$\pi'_{\mathcal{P}}$ with $\pi'_{\mathcal{A}}$ to form $\pi' = (s'_0, \text{assume } a'_0)\ (s_1, \text{assume } a'_1)\ \cdots$. Next, we construct $\pi = (s_0, \text{assume } a_0)\ (s_1, \text{assume } a_1)\ \cdots$ by recovering $(s, \text{assume } \mathit{true})$ for every deleted statement s from $\pi_{\mathcal{P}}$. The trace π is fair and feasible to $\mathcal{B}$ based on the definition of Büchi transaction, and therefore contradicts the assumption. □

NDD: A Decision Diagram for Network Verification

Zechun Li*, Peng Zhang*, Yichi Zhang*, Hongkun Yang†
*Xi'an Jiaotong University, †Google

Abstract

State-of-the-art network verifiers extensively use Binary Decision Diagram (BDD) as the underlying data structure to represent the network state and equivalence classes. Despite its wide usage, we find BDD is not ideal for network verification: verifiers need to handle the low-level computation of equivalence classes, and still face scalability issues when the network state has a lot of bits.

To this end, this paper introduces *Network Decision Diagram (NDD)*, a new decision diagram customized for network verification. In a nutshell, NDD wraps BDD with another layers of decision diagram, such that each NDD node represents a *field* of the network, and each edge is labeled with a BDD encoding the values of that field. We designed and implemented a library for NDD, which features a native support for equivalence classes, and higher efficiency in memory and computation. Using the NDD library, we re-implemented five BDD-based verifiers with minor modifications to their original codes, and observed a 100× reduction in memory cost and 100× speedup. This indicates that NDD provides a drop-in replacement of BDD for network verifiers.

1 Introduction

Network verification is becoming an important toolchain for network operators to ensure the correctness of their networks [9, 10, 13, 14, 16, 19, 24–28, 30, 32–34, 37, 39, 41, 47, 48, 50–58], and some of them have been deployed by large service providers to detect misconfigurations [32, 37, 48, 54].

Many network verifiers use *Binary Decision Diagram (BDD)* [11, 20], to compactly represent network state like packet headers, routes, failures, etc. Despite its extensive usage, we find that BDD is not ideal for network verification tasks, in the following three aspects.

Memory inefficiency. We observe BDD-based network verifiers create a lot number of redundant nodes, which share the same sub-structure on some fields, but cannot be reduced since their differences on other fields. This redundancy is magnified when verifiers [13, 57] compute equivalence classes (i.e., atomic predicates, or simply atoms [1]), which leads to an "explosion of atoms" and overflows the memory.

Computation inefficiency. We observe that BDD-based network verifiers are slowed down by logical operations, e.g., $a \wedge b$. The reason is BDD operations proceed in a recursive way, with each recursion looking at a single bit of a and b. Even worse, if the BDDs encode different variables, more recursions are needed to "align" their variables. Considering networks often match on multiple fields (e.g., 5-tuple, 104 bits), the recursions can be deep and therefore very slow.

Lack of support for network verification. We observed that BDD libraries offer little support for common tasks of network verification, e.g., computation of atoms [51], incremental update of atoms [13, 57], and handling of packet transformers [53]. As a result, network verifiers need to design their own algorithms, which can be quite complex, for these common tasks, and still encounter scalability problems due to the aforementioned explosion of atoms.

We attribute the above problems to the fact that BDD is agnostic of *fields*, an important semantics in networks, where a device processes routes and packets by matching one or multiple fields. Moreover, the matching shows a characteristic of "field locality": *each rule matches few but different fields of the network.* This makes the fields mostly orthogonal, such that the number of nodes can be a cross-product of that for each field, and therefore quite large.

Based on the above observation, our basic idea is: *instead of viewing the entire network state as a whole, we partition it into a set of* ***fields***, so that logical operations are restricted to BDDs of the same field, and redundant nodes of the same fields can be reduced, fully independent of other fields.

After decoupling fields, a question arise: how to represent network state with the per-field BDDs? A naive way of using conjunctions of per-field BDDs is not compact, and can lead to an explosive number of conjunctions. Our approach to the

[1] For brevity, the following of this paper will use the term "atoms" to refer equivalence classes or atomic predicates [51].

representation problem is to wrap the per-field BDDs with another layer of decision diagram, termed *Network Decision Diagram (NDD)*. In an NDD, each NDD node u encodes a field f of multiple bits, and each edge of u is labeled with a BDD of field f. We show such a representation is *compact*, and prove it is actually *canonical*. In addition, since NDDs can look at many bits each time, logical operations of NDDs require much less recursions.

Finally, instead of computing single set of atoms for all fields, we compute a set of atoms for each field f, and transform the labels of each NDD edge from BDDs into a set of atoms. We term the resultant NDD as *atomized NDD*. Then, logical operations (e.g., $\wedge$, $\vee$) on NDDs can be replaced with simpler set operations (e.g., $\cap$, $\cup$), similar to [51].

The atomized NDD offers a lot of benefits. First, since the atoms are over a single field, we avoid the cross-product effect and can dramatically reduce the total number of atoms in the network. Moreover, all atom-related operations can be *fully transparent* to network verifiers, such that verifiers are agnostic of atoms, and do not need to realize their own atom-related algorithms.

Based on the above ideas, we realize an NDD library [2]. It features a native support for the computation and incremental update of atoms, and provides an efficient way for handling packet transformers. We use the NDD library to re-implement 5 BDD-based network verifiers, i.e., AP Verifier [52], APT [53], APKeep [57], SRE [58], and Batfish [19], with minor modifications to their original codes. After replacing the BDD library with the NDD library, these verifiers show a 2 orders-of-magnitude reduction in both memory and time.

Contributions. In sum, this paper makes three contributions:

- We introduce Network Decision Diagram (NDD), a new decision diagram customized for network verification.
- We implement an NDD library, and use it for 5 different network verifiers with small modified LOCs.
- We use real and synthetic datasets to show that by using NDD, the 5 verifiers achieve a 2 orders-of-magnitude reduction in both memory and time.

Limitations. NDD builds on the assumption of field locality, which generally holds on our datasets. However, if rules in a network match most of the fields, NDD may not perform better than BDD (§6.6).

2 Motivation

2.1 Preliminary to BDD

Binary Decision Diagram (BDD) is a rooted, directed acyclic graph (DAG) with two terminal nodes (`true` and `false`), and multiple non-terminal (variable) nodes (refer to Figure 1 as an example). Each non-terminal node u represents a test of a boolean variable $var(u)$, and has two outgoing edges, i.e., a high edge and low edge (shown as solid and dashed line, respectively). These two edges point to two *successor* nodes $high(u)$ and $low(u)$, i.e., a high successor and a low successor, respectively. The semantics is that: if x is tested to be *true* (*false*), then the high (low) edge will be followed and the next test will be $high(u)$ ($low(u)$). Each path from the root node to the terminal node `true` corresponds to one truth assignment of a boolean formula. For the BDD in Figure 1, there are only two paths from the root node to the terminal node `true`, encoding two truth assignments of $x_1 = false, x_2 = true$, and $x_1 = true, x_2 = true, x_3 = true$, respectively. Therefore, this BDD represents a boolean formula $(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2 \wedge x_3)$.

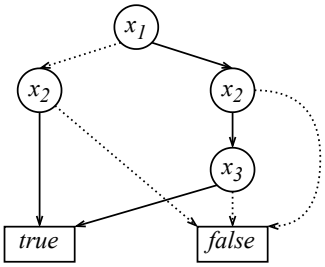


Figure 1: An example of BDD encoding the boolean formula $(\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2 \wedge x_3)$.

Reduced Ordered BDD. A BDD is said to be an *Ordered* (OBDD) if the boolean variables follow a fixed variable order (say $x_1 < x_2 < \ldots, < x_n$ where $x_i < x_j$ means variable x_i appears before variable x_j) on all paths through the graph. An OBDD is said to be a *Reduced* (ROBDD) if it satisfies the following two conditions:

(C1) Uniqueness: no two distinct nodes represent the same variable and have the same successors;

(C2) No redundant node: no non-terminal node has identical high and low successor.

An important property of ROBDD is *canonicity*, i.e., for any boolean formula, if the variable ordering is specified, there is a unique ROBDD representing it [11]. As a result, ROBDD can significantly reduce the node redundancy [20] for a compact representation of boolean formulas, and is very efficient for logical operations. Without specified otherwise, the BDDs in this paper are ROBDDs.

BDD library. To use BDD, applications like network verifiers leverage off-the-shelf BDD libraries [7, 8]. A BDD library needs to realize logical operations, e.g., `and`, `or`, `not`, and other useful functions like `exist`, `restrict`, `oneSat`, `satCount`, etc. Most of them are realized in a recursive way.

To achieve high efficiency of memory and time, modern BDD libraries rely on some efficient algorithms and data structures. The *node table (unique table)* is a hash table holding all BDD nodes, each of which is a distinct tuple $(var, low, high)$. When the node table is full, a BDD library will free unused nodes with *garbage collection (gc)*. A common way to realize

[2] open sourced at https://github.com/XJTU-NetVerify/NDD

BDD

```
1. foreach 1≤i≤32: vars(i) ← bdd.createVar();
2. foreach d ∈ devices: Preds.add(calPortPred(d, vars));
3. foreach pred ∈ Preds:
4.   foreach a ∈ Atoms:
5.     set.add(bdd.and(a, pred));
6.     set.add(bdd.and(a, bdd.not(pred)));
7.   Atoms ← set;
8. foreach d ∈ devices, p ∈ d.ports:
9.   foreach a ∈ Atoms:
10.    if (bdd.and(p.pred,a) = a):
11.      p.atoms.add(a);
12. traverseRec(Atoms, src);

13. Function traverseRec(pktSet, d)
14.   if (!pktSet.isEmpty()):
15.     foreach p ∈ d.ports:
16.       forwarded ← pktSet.intersect(p.atoms);
17.       traverseRec(forwarded, p.nextHop);
```

NDD

```
1. var ← ndd.createVar(32);
2. foreach d ∈ devices: Preds.add(calPortPred(d, var));
3. ndd.atomize(Preds);
4. traverseRec(ndd.true, src);

5. Function traverseRec(pktSet, d)
6.   if (pktSet != false):
7.     foreach p ∈ d.ports:
8.       forwarded ← ndd.and(pktSet, p.pred);
9.       traverseRec(forwarded, p.nextHop);
```

Figure 2: Pseudo code to verify networks based on BDD and NDD, respectively.

gc is to maintain a reference count for each node, and when the reference count of a node reaches zero, the node is marked as dead, and will be freed when gc is called. Most BDD libraries use *operation cache* to speed up logical operations: before each operation, the library firstly checks whether the result exists in the operation cache, and returns the result if there is a match; otherwise, the operation is performed and the result is put into the cache.

2.2 The use of BDD in network verification

In the recent decade, BDD has been extensively used for network verification. For example, many data plane verifiers [13, 51, 57] use BDDs to efficiently represent packet equivalent classes, and some control plane verifiers like SRE [57] use BDD to represent failure equivalence classes. Interested readers can refer to Appendix §G for an incomplete list of network verifiers based on BDDs.

The top of Figure 2 shows a code snippet to implement data plane verification based on BDD [51]. Note that we oversimplify the codes to reflect the core logic. First, the verifier creates a BDD variable for each bit of the field. Then, it computes port predicates for each device (Lines 1-2), computes atoms of all port predicates(Lines 3-7), and compute the set of atoms for each port (Lines 8-11). Finally, it computes reachability by recursively traversing the network with a packet set initialized to all atoms, and at each node filters some of atoms based on the forwarding rules (Lines 12-17).

Despite its extensive usage in network verification, we find BDD, however, is not ideal for achieving high efficiency for either memory or computation. To show this, we use a toy example with four nodes, as shown in Figure 3. For simplicity of illustration, we assume there are four fields, i.e., `dstIP`, `dstPort`, `srcIP`, and `srcPort`, where `srcIP` has only four bits and other fields each has only two bits.

Memory inefficiency. First, we observe that there are a lot of redundant BDD nodes that cannot be reduced. Figure 4(a) shows three BDDs encoding three sets of packets that are reachable from A to port 1 of D, following three different paths (i.e., $A \to B \to D$, $A \to C \to D$, and $A \to D$). These three BDDs have a common sub-structure consisting of 14 nodes which encode variables x_1 to x_6 of `srcIP` and `srcPort` (inside the red dashed box). That is, two copies of the sub-structure ($14 \times 2 = 28$ nodes) are redundant. However, we cannot eliminate these 28 nodes. The reason is that their nodes encoding x_6 point to different successor nodes encoding x_7 (the first bit of `dstIP`), and therefore satisfy the uniqueness condition of ROBDD and cannot be represented with a single node. We term such a redundancy as *partial redundancy* as it only exhibits on a subset of variables. The number of 28 redundant nodes may not seem a big issue in this toy example, while for real networks, this number can be as large as $O(10^8)$, leading to a huge memory cost (§6.5).

In addition, the effect of partial redundancy is magnified by the computation of atoms. When computing atoms, a verifier needs to perform many logical conjunctions on BDDs, thereby creating even more redundant nodes for each field. On our virtualized datacenter network datasets, which have multiple layers of packet headers, the number of redundant nodes can overflow the BDD node table, a phenomenon which we term as *explosion of atoms*. Interested readers can refer to Appendix D for an example.

Computation inefficiency. In addition to the high memory cost due to redundant nodes, we observe BDD-based network verifiers are also unnecessarily slow due to the frequent but inefficient logical operations. Specifically, to compute atoms or check reachability, network verifiers need to perform many logical operations ($\wedge$, $\vee$, $\neg$). Inside a BDD library, these operations are realized recursively: a conjunction $a \wedge b$, where a and b encode the same variable, boils down to two recursive calls of $low(a) \wedge low(b)$ and $high(a) \wedge high(b)$. Since each recursion only proceeds one bit, the recursions can go very deep when there are many bits (e.g., 104 bits for 5-tuples).

Worse still, if a and b encode different variables, with an ordering say $var(a) < var(b)$, many recursions may be needed to "align" variables, i.e., recursively find a descendant node of a, say a', such that $var(a') = var(b)$. For example, in Figure 4(a), to compute `and` of BDDs labeled by $A \to B \to D$ and b_3, we first need to recursively find the node of b_1, which encodes the same variable x_7 as b_3, rather than directly jumping to the field `dstIP`. This amounts to 17 recursions in total.

One may wonder whether we can mitigate these limitations with a better variable ordering. Putting aside finding an optimal variable ordering is a co-NP-complete problem [20], we observe that the redundancy is there no matter what the ordering of BDD variable is: when we re-order the variables, the redundancy is just shifting from one place to another (see Appendix A.9 for some numerical results.)

Lack of support for network verification. As a general-purpose data structure, BDD offers little support for some important tasks of network verification, including: (1) the computation of atoms, (2) the incremental update of atoms, and (3) the handling of packet transfomers.

First, many verifiers compute atoms over the network state for a faster verification. For example, BDD-based data plane verifiers like AP Verifier [52] partition the header space into a small set of atoms, such that packets in the same atomic predicate have the same forwarding behavior. When computing reachable packet sets from source to destination, the verifiers can use set operations ($\cup$, $\cap$) on sets of atoms, instead of the more expensive logical operations ($\vee$, $\wedge$) on BDDs. However, using BDD, these verifiers need to implement their own algorithms to compute atoms, which is nontrival and leads to explosion of atoms.

Secondly, to avoid from-scratch re-computation of atoms after each network update, network verifiers [13, 30, 57] need to design efficient algorithms to incrementally update the atoms. For example, APKeep [57], a state-of-the-art data plane verifier, designs efficient algorithms to achieve a realtime update of atoms; Katra [13] extends APKeep to multi-layer networks by computing *partial equivalence classes*, which can significantly improve the scalability of data plane verification for multi-layer networks. However, both APKeep and Katra need to implement their own algorithms for updating the set of atoms, which tend to be more complicated than the from-scratch computation of atoms, as in AP Verifier [51].

Finally, packet transformers such as NAT, IP-in-IP [40], VXLAN [38], etc., are common in networks. The existence of packet transformers makes the computation of atoms more complicated. Verifiers like APT [53] and APKeep [57] use customized algorithms to compute the atoms, such that the input and output of any transformer can be represented by a set of atoms. Such algorithms are also complex and may end up with many atoms.

3 Introducing NDD

3.1 Observations

We observe the inefficiency of BDDs in network verification is mainly due to the fact that BDD, as a general-purpose data structure, lacks the notion of "field", an important semantics in networks. For example, network devices forward or filter packets by matching one or multiple packet fields. Without such a notion, network verifiers based on BDDs model the network state as a whole, and the number of BDD nodes can grow exponentially with the number of fields, due to the following "field locality" phenomenon.

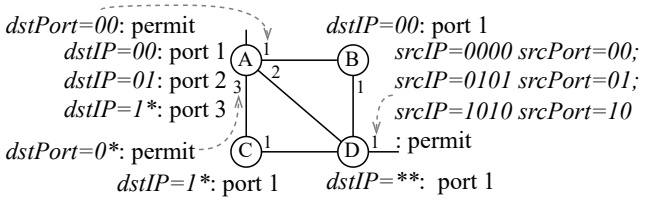


Figure 3: A toy example network used throughout this paper.

Field locality refers to the fact that even though there can be many fields in the network, each rule only matches on one or few of these fields, and different rules may match different fields. For example, forwarding rules in the underlay of a network virtualized with VXLAN only match on the `dstIP` of the outer header, while forwarding rules in the overlay only match the `dstIP` of the inner header. As another example, most ACL rules match a single or rather few fields, e.g., only 25 of 392 ACLs in the Purdue campus network match more than one field [46]. Finally, each route policy filters routes only based on one or two of all attributes, e.g., prefix, communities, AS path.

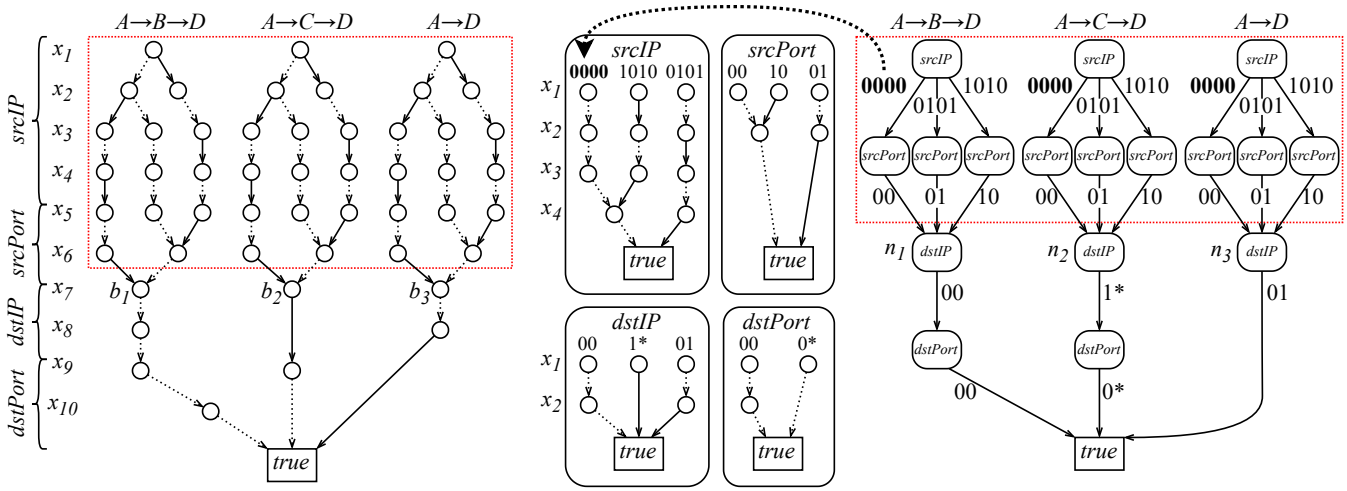
Due to field locality, the different fields in network state are *orthogonal*, meaning that the values of each field tend to be independent of those of other fields. As shown in Figure 4(a), the nodes for `src IP/Port` fields are orthogonal to those of the `dst IP/Port` fields. As a result, multiple copies of the same sub-structure for `src IP/Port` need to be created, due to the different sub-structures for `dst IP/Port`.

3.2 Idea

Decoupling the fields. Our basic idea is partitioning the whole network state (a bit vector) into a set of *fields*, which can be packet headers (e.g., 5-tuple), route attributes (e.g., community, AS Path), or up/down state for different sets of links, etc. Then, for each field, we create per-field BDD nodes and place them into their own *namespace*, such that logical operations and reduction of redundant nodes are restricted to BDDs in the same namespace. As shown in Figure 4(b), after reducing the redundant nodes for each of the four fields, there are only 11, 5, 5 and 3 BDD nodes, respectively. This amounts to a total number of 24 BDD nodes, instead of 50 ones as previously shown in Figure 4(a).

After the decoupling of fields, we can effectively reduce the number of redundant nodes. The question is how can we represent the network state with these BDDs, each of which only encodes a field? A straightforward way is to use a set of conjunctions or arrays of BDDs, where each BDD in an array encodes a specific field (see Appendix E for details). Such an approach is simple but can lead to an explosion of arrays after logical operations (see Appendix A.10 for details).

Assembling fields. We approach this representation problem



(a) BDDs (b) Per-field BDDs (c) NDDs

Figure 4: Illustrations of BDDs and NDDs. The dashed/solid lines represent `false`/`true`, and all edges to the BDD/NDD terminal node `false` are omitted for simplicity.

by layering another layer of decision diagram on top of the per-field BDDs. We term this layer of decision diagram as *Network Decision Diagram (NDD)*. In an NDD, each NDD node represents a field of multiple bits and has multiple edges; each edge of it is labeled with a BDD encoding those bits.

Such a representation is more compact than using BDD arrays or conjunctions. Returning to the example, Figure 4(c) shows the NDDs representing three sets of packets reachable from A to D. Compared to (a) BDDs, NDD only creates one copy of per-field BDD nodes in (b), as shown inside the red box. This leads to 24 BDD nodes in total, instead of 50 ones as in (a). Note that NDD also needs three copies for the same sub-structure; otherwise, it cannot distinguish the three different packet sets. However, since each NDD variable represents a field, each copy of the sub-structure has only 4 NDD nodes, which is much less than 14 BDD nodes as in (a) (in red rectangles).

Moreover, logical operations on NDDs is much faster than those on BDDs, since it can quickly skip a field in one recursion. For example, in Figure 4(c), to compute `and` of the NDD nodes labeled by $A \rightarrow B \rightarrow D$ and n_3, from the root node of $A \rightarrow B \rightarrow D$, NDD only needs 6 recursion for all paths in total to reach the node n_1, which has the same variable with n_3. This contrasts with BDDs, which need 17 recursions.

Finally, recall that network verifiers compute atoms over the whole network state, leading to an explosion of atoms. In addition, network verifiers need to design and implement their own algorithm, in order to efficiently compute and update the atoms, which are non-trival tasks.

Atomizing fields. Towards this problem, we design efficient algorithm for the computation and update of atoms as an internal process of NDD. Specifically, we compute a separate set of atoms for each field, and then transform the label of each NDD edge from a BDD to a set of atoms for that field. After that, the logical operations (e.g., conjunctions) on BDDs become set operations (e.g., intersections), which tend to be faster. By computing atoms over individual fields, NDDs avoid the problem of explosion of atoms. We term the NDDs where labels of edges are represented with atoms as *atomized NDDs*. Since NDD natively supports atomization, network verifiers become fully agnostic of atoms, and can still use the same logical operators after the NDDs are atomized.

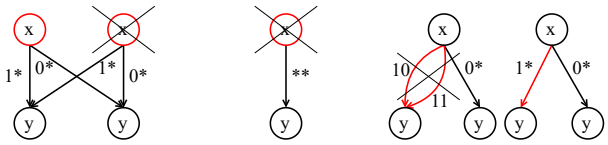
3.3 Network Decision Diagram

Similar to BDD, an Network Decision Diagram (NDD) is also a rooted, directed acyclic graph with two terminal NDD nodes `true` and `false`. The key difference is that there are multiple non-terminal NDD nodes, each of which represents a *field*, rather than a *bit*. Formally, we have the following definition, which is adapted from the definition of BDD in [11].

Definition 1. *A **Network Decision Diagram (NDD)** is a rooted, directed acyclic graph with*

- *two terminal NDD nodes `true` and `false`, with an out-degree of zero.*
- *a set of non-terminal NDD nodes. Each node u is associated with a variable $var(u)$ representing a **field** of one or multiple bits, and has a set of outgoing edges, denoted as $edges(u)$. Each $e \in edges(u)$ points to a successor of u, denoted as $next(e)$, and has a predicate over the variable $var(u)$, denoted as $label(e)$.*
- *$\forall u$, $\forall x, y \in edges(u)$ with $x \neq y$: $label(x) \wedge label(y) = false$, and $\bigvee_{e \in edges(u)} label(e) = true$.*

Note above the third condition generalizes that of BDD, saying that the labels of edges from the same NDD node are exclusive, and the union of these labels covers all values of



(a) Non-unique nodes (b) Redundant nodes (c) Redundant edges

Figure 5: Three types of redundancy in NDD.

the field. For BDD, a label is either 0 or 1, clearly satisfying this condition.

Reduced Ordered NDD. The definition of Reduced Ordered NDD (RONDD) is similar to that of ROBDD (§2.1). The difference is that besides the two conditions *uniqueness* and *no redundant node*, RONDD has a third condition:

(C3) No redundant edges: no two edges from the same NDD node point to the same successor, i.e.,

$$\forall u, \forall x, y \in edge(u) : next(x) = next(y) \Rightarrow x = y \qquad (1)$$

Figure 5 shows examples of the three conditions for RONDD. Note that the number of edges for a non-terminal NDD node u can be up to 2^n in the worst case, where n is the number of bits for $var(u)$. But due to the third condition (C3), the actual number is much smaller than that. For example, 99% of NDD nodes in SRE have no more than 10 edges (see Appendix A.6 for details). Similar to ROBDD, RONDD also has the canonicity property, such that RONDD can compactly represent equivalent network state.

Lemma 1. ***Canonicity of NDD.*** *For any boolean function $F(x_1, x_2, \cdots, x_N)$ over N boolean variables with ordering $x_1 < x_2 < \cdots < x_N$, there is exactly only one RONDD u with k NDD variables and ordering $f_1 < f_2 < \cdots < f_k$, where f_i has n_i bits $x_i^1 < x_i^2 < \cdots < x_i^{n_i}$, and $N = \sum_{i=1}^{k} n_i$, such that u can represent F.*

Proof. The proof can be found in Appendix C. □

3.4 Atomized Network Decision Diagram

Atomized Network Decision Diagram is an NDD where the label of each edge is a set of atoms, instead of a BDD.

Definition 2. *Given a set of NDDs $\mathcal{N}$ defined over a set of variables F, we say $\mathcal{A}(f) = \{a_1^f, \ldots, a_k^f\}$ is the **set of atoms** for variable $f \in F$, with respect to $\mathcal{N}$, if it satisfies the following conditions:*

(1) $a_i^f \neq false, \forall i \in \{1, \ldots, k\}$;

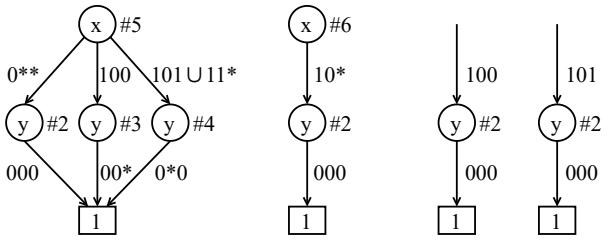
(2) $\vee_{i=1}^{k} a_i^f = true$;

(3) $a_i^f \wedge a_j^f = false$, if $i \neq j$;

(4) $\forall e \in edges(u)$, u is a node of $\mathcal{N}$ with $var(u) = f$: there exists a set $atoms(e) \subset \mathcal{A}(f)$, s.t., $label(e) = \bigvee_{a \in atoms(e)} a$;

(5) k is the minimum number satisfy the above properties.

Definition 3. *Given a set of NDDs $\mathcal{N}$, we say $\mathcal{N}^a$ is the **atomized NDDs** of $\mathcal{N}$, if $\mathcal{N}^a = \mathcal{N}$, except that for each edge e of $\mathcal{N}$, $label(e) \leftarrow atoms(e)$.*



(a) Operand A (b) Operand B (c) Sub-results

Figure 6: Examples for `and` operation of NDD.

4 Design of the NDD Library

Realizing an efficient NDD library to replace the current BDD libraries for network verifiers is a non-trivial task, given that modern BDD libraries apply various optimizations to reduce memory cost and speed up computation. In this paper, we design an NDD library based on existing BDD libraries like JDD, by (1) extending the standard APIs with new functions, e.g., `atomize`, and `update`; (2) adapting the design of logical operations, garbage collection, and node table of BDDs to work for NDDs, (3) re-using some optimizations of BDD, e.g., operation cache, and (4) applying some new NDD-specific optimizations like removing the `false` NDD node.

4.1 The APIs

Table 1 shows the APIs of our NDD library, which is quite similar to those of BDD libraries: The NDD library provides operations similar to BDD, including `createVar`, `apply`, `not` and `exist`, etc., such that existing verifiers can use the same operators when switching from BDD to NDD. One exception is `createVar`, which takes a parameter of *len* to represent the number of bits for the NDD variable. To enable a native support for the computation and update of atoms, the NDD library also provides two additional functions, `atomize` and `update`, using which a verifier can instruct the NDD library to compute and update the atoms, respectively.

`createVar`. This API lets verifiers to declare a variable in NDD, to represent a field in network. When a verifier calls `createVar` with parameter *len*, the NDD library firstly calls the `createVar` function of BDD libraries to create *len* BDD variables, and then creates an NDD variable based on these *len* BDD variables. Then, the NDD library maintains the mapping between the NDD variable and the set of BDD variables, which helps find the related BDD namespace of an NDD node during NDD operations.

`apply`. This API provides a unified logical operation $op \in \{and, or, diff\}$ on two NDDs. In BDD libraries, `apply` is realized in a recursive way. Taking $c = \texttt{apply}(\texttt{op}, a, b)$ for example, where a and b encode the same BDD variable *var*, the function will recursively compute $l = \texttt{apply}(\texttt{op}, low(a), low(b))$, and $h = \texttt{apply}(\texttt{and}, high(a), high(b))$, and returns a BDD node c with $var(c) = var$, $low(c) = l$ and $high(c) = h$.

Table 1: The APIs of NDD library. The first four are similar to BDD libraries, and the last two are introduced by NDD.

Function	Description
`createVar` (Integer *len*)	create an NDD variable with *len* bits
`apply` (*op*, NDD *a*, NDD *b*)	apply a logical operation *op* on two NDDs *a* and *b*, where $op \in \{and,\ or,\ diff\}$
`not` (NDD *a*)	apply a logical negation on an NDD *a*
`exist` (NDD *a*, Field *field*)	apply existential quantification for field *field* on NDD *a*
`atomize` (Set<NDD> *N*)	compute per-field atoms for a set of NDDs *N*, replace BDDs of edges in *N* with a set of atoms
`update` (NDD δ, NDD *a*)	update the set of atoms when a new NDD δ is added; *a* is an atomized NDD satisfying that $\delta \Rightarrow a$

For NDD, things become more complicated since each NDD node may have multiple successors, each with a predicate. We use the example in Figure 6 to illustrate this. Suppose there are 2 fields x and y each with 3 bits, and we want to compute `apply`$(op, \#5, \#6)$. Without loss of generality, we assume `op=and`. Unlike in BDD, where both BDD nodes have the same number of 2 successors, the first NDD #5 has 3 successors while the second NDD #6 has a singleton successor. Therefore, we need to enumerate all pairs of successors (one from #5 and the other from #6). If the two successors have overlapping test conditions, we can recursively call `apply` on these two successors; otherwise, we continue to another pair. For this example, the test condition of the left successor of #5 does not overlap with that of #6, since `bdd_and`$(0**, 10*) = false$. Therefore, we continue to the middle successor of #5, whose test condition overlaps with that of #6. Then, we compute `apply`$(and, \#3, \#2)$, and the result is the NDD node #2, and the test condition for this node is 100, as shown on the left of Figure 6(c). Similarly for the right successor of #5, we have the result shown on the right of Figure 6(c). As we can see, there are two edges both pointing to the NDD node #2, with different conditions 100 and 101. Then, NDD can merge these two edges into a single one, and compute a disjunction of the two conditions as $10*$, resulting in the NDD node #6. Thus, we have `apply`$(and, \#5, \#6) = \#6$.

Theoretically, if two NDD nodes have m and n successors, respectively, we need to enumerate $m \times n$ pairs of successors, and for each pair, we need to compute a conjunction of two BDDs. However, in practice, this number is small. For example, 99% of NDD nodes have less than 10 successors, when we use NDD-based SRE [58] on an 80-nodes fattree networks (Appendix A.6).

`atomize`. This API takes a set of NDDs as input, and computes atoms for each variable, and transforms each of the NDDs into atomized NDDs, where each edge has a set of atoms $atoms(e)$, satisfying $\vee_{a \in atoms(e)} a = label(e)$. We realize this by firstly dividing all the NDD nodes and their successors into groups, one for each NDD variable. After that, for each group, we compute the atoms over all NDD nodes in the group, in a similar way as [51]. Finally, for each edge $e \in edges(u)$ of some NDD node u, we compute $atoms(e)$ as follows: for each atom a of $var(u)$, if $a \Rightarrow label(e)$, we insert a into the atom set $atoms(e)$. For logical operations on two atomized NDDs, we can use set operations over $atoms(e)$ instead of logical operations over $label(e)$.

`update`. This API takes an NDD δ and an atomized NDD a satisfying $\delta \Rightarrow a$ as input, updates per-field atoms and computes the atom set $atoms(e)$ for each edge e of δ and all (direct or indirect) successors of δ. For the NDD node δ, we can compute the conjunctions of BDDs of edges of δ with each atom of field $var(\delta)$. But this is not necessary as $\delta \Rightarrow a$, where a is an atomized NDD: we just need to compute conjunctions with the atoms on paths from a to terminal `true`, since atoms not in a never intersect with BDDs in δ. After that, we recursively apply the `update` for the successors of δ and a. But instead of enumerating all the edges of a for each $e_\delta \in edges(\delta)$, we just need to consider an edge e if any atom in $atoms(e)$ has a non-false conjunction with $label(e_\delta)$. Finally, we need to replace the atoms being split with new atoms on all related edges of atomized NDDs.

Details on the implementations of the above APIs can be found in Appendix B.

4.2 The Internals

Node table, also termed as unique table, is a core data structure used by a BDD library to hold BDD nodes. Since the BDD node table often dominates the memory cost of a BDD library, and most API calls need to manipulate the node table, the efficiency of node table will be the key to the performance of a BDD library. A node table is essentially a hash table with each BDD node is uniquely identified by a tuple $(var, low, high)$. In order to achieve a higher efficiency, instead of using the built-in data structures like `HashMap` in Java, modern BDD libraries like Buddy (C++), JDD (a Java version of Buddy), etc., choose to implement their own node table with a dynamic array, with customized hash algorithms and collision resolution methods.

Different from BDD, where each node has exactly two successors, an NDD node can have a variant number of successors, and thus need a variant size of memory. Due to the non-uniform NDD node size, it is nontrivial to adapt the array-based implementation of BDD node table to NDD. Fortunately, we find that the number of NDD nodes is much smaller compared to that of BDD nodes, and therefore choose a simpler implementation based on hash map, as shown in Figure 7. Specifically, we represent an edge set of a node with a hash map , where the key is the next node of the edge, and the value is the label of the edge. Then, for each NDD variable, we also maintain a hash map, where the key is an edge set,

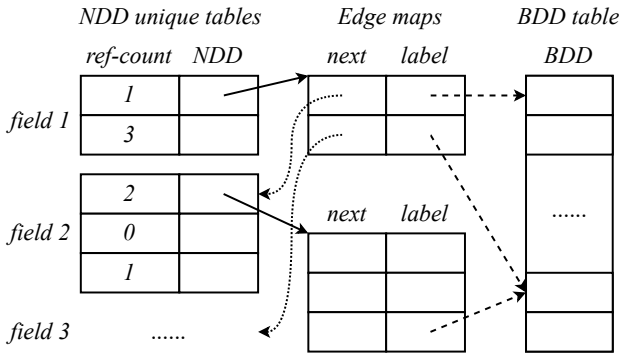


Figure 7: The node table of NDD.

and the value is an NDD node which has such an edge set. This hash map can be seen as a table holding nodes of the same NDD variable. Then, the NDD node table is just a set of these tables, one for each NDD variable. Note that we do not need *var* in the hash key, since each variable has its own sub-table. Also note that the hash function used for the edge set should be associative and commutative; otherwise, two nodes with the same successors and labels, but in a different order may not have the same hash code, and will be treated as different nodes. Our current implementation uses the Java native `HashMap` (JDK 1.8), which satisfies such requirements. Further optimizations may work better but we find the above implementation already achieves a considerable improvement for network verifiers based on BDDs.

As shown in Figure 7, for each NDD node, the library creates a hash map consisting of all its edges. Then, the library stores a tuple of a reference count, and edges of the node, into the NDD node table, which is also a hash map. Note that each field has its own NDD node table, and the BDD nodes for different fields are *logically isolated*: they still share the same BDD node table, but do not interfere with each other.

Redundancy elimination is a function that is necessary for (RO)BDDs to achieve a canonical representation of boolean formulas. Recall that (RO)BDDs use two redundancy eliminations rules *uniqueness* and *no redundant nodes (tests)*. Besides the above two rules, NDDs apply a third rule *no redundant edges*. To apply this rule, the NDD library checks whether two edges of an NDD node point to the same successor; if so, it merges them into a single edge whose condition is the disjunctions of the conditions of the two merged edges. For *uniqueness*, BDD views two nodes as the same if their two edges point to the same successor. NDD, however, requires that not only their edges point to the same set of successors, but also require that for each successor, the test conditions (BDD) are the same. For *no redundant node*, BDD views a node is redundant if both the high and low edges point to the same successor. NDD, however, applies this rule in a slightly different way: since NDD always merges edges pointing to the same successor, an NDD node is viewed as redundant if it has a single edge with condition `true`.

Garbage collection is important for BDD libraries to reduce memory usage. A BDD library maintains a reference count for each node, and marks those nodes with reference count of zero as *dead*. When the size of a node table is reaching a specified threshold, it frees all those dead BDD nodes to make more space for new nodes. Similar to a BDD library, an NDD library also uses reference counts for `gc`. One thing worth mentioning is the dependency of the `gc` processes of the NDD library and the BDD library. Firstly, since an NDD edge uses BDDs to represent the test conditions, when the NDD `gc` frees some NDD nodes and their edges, it should decrease the reference count of BDD nodes used by these edges. Secondly, we should ensure *an NDD `gc` is always be called right before a BDD `gc`*. The reason is as follows. Suppose the BDD `gc` is going to free all dead BDD nodes, some of which are still in use on NDD edges (this can happen when an NDD node is marked as dead but not freed by a `gc`). Those BDD nodes can't be freed without NDD `gc` freeing those edges, and that will affect the performance of BDD library.

Removing the `false` NDD node. In BDD, there are two terminal nodes `true` and `false`, where the `false` node and all edges pointing to it can be removed without affecting the correctness. However, most BDD libraries choose not to do so, since the benefit is marginal. In NDD, however, removing the `false` NDD node can achieve a considerable reduction in memory cost. The reason is that each edge in NDD is associated with a BDD, which represents the test condition of that edge, and by removing edges to `false`, we can save the BDDs on those edges. Moreover, since these BDDs often represent some "default" cases, they tend to be complex and freeing them often brings even larger reduction than other BDDs not on edges to `false`. For atomized NDD, we observe that the edges to the `false` NDD node often hold most of the atoms, and removing the edges allows NDD to maintain much less atoms. On our datasets, 85% edges pointing to `false` take more than 95% atoms of related field, and 30% of the edges each takes more than 2000 atoms (see Appendix A.7 for details). In addition, since the `and` operations do not care about the `false` NDD node (the conjunction of an NDD with it is still `false`), removing edges to the `false` node can reduce the running time by not enumerating these edges.

Operation cache. Similar to BDD libraries, our NDD library also maintains an operation cache, a hash table where the key is the operation, e.g., (`op`, *a*, *b*) and the value is the result of the operation. Our experiment shows an operation cache can significantly speed up NDD operations, thereby making the NDD-based verifiers faster. For example, NDD-based SRE [58] with an operation cache is shown to be over 10× faster than without an operation cache.

4.3 Using NDD

We show how to use the APIs of NDD library to realize the same data plane verification task as using BDD libraries. The bottom of Figure 2 shows the code snippet for data plane

Table 2: Number of modified LOC when replacing BDD with NDD, for different verifiers. Note that the numbers exclude those for trivial string replacement (e.g., "BDD" → "NDD").

Verifiers	AP Verifier	APT	APKeep	SRE	Batfish
Library	JDD	JDD	JDD	JDD	JavaBDD
Add	+25	+84	+66	+205	+8
Delete	-159	-167	-311	-53	-7

verification based on NDD. The logic is mostly the same with that of BDD, with key differences including: (1) NDD creates a single variable with 32 bits, instead of 32 boolean variables (Line 1); (2) NDD uses a single line of `atomize` to automatically compute atoms, after which the predicate of each port becomes an atomized NDD (Line 3); (3) When computing reachability based on atomized NDDs, verifiers use the same `and` operation as the standard (non-atomized) NDD, and therefore fully agnostic of atoms (Line 8). Details about this case, and two more use cases for incremental data plane verification and data plane verification with transformers can be found in Appendix F.

5 Implementation

The NDD Library. We implemented a library of NDD with ∼2K lines of Java code, based on JDD [8], a BDD library used by many verifiers like AP Verifier [51], APT [53], APKeep [57], SRE [58], etc. In addition, we integrated the NDD library as a new factory in JavaBDD, with another ∼100 lines of code, so that verifiers using JavaBDD, e.g., Batfish [4], Campion [47], can switch to use NDD instead.

Network verifiers based on NDD. We use the NDD library to replace the BDD library in 3 data plane verifiers, and 2 control plane verifiers. Table 2 shows the number of modified LOC for each verifier. As we can see, the number of added LOC is small since the NDD exposes similar APIs as BDD, and existing verifiers can directly use them without modifications. On the other hand, the number of deleted LOC is slightly larger. The reason is that NDD natively supports the computation of atoms, and therefore we remove the corresponding codes in those verifiers. An exception is SRE, which uses customized algorithms to traverse BDDs, and we need to adapt those algorithms to NDDs, with more modified LOC. Therefore, we conclude that the NDD library offers a drop-in replacement for BDD-based verifiers, with a small modifications to their original code base. More details can be found in Appendix A.1.

6 Evaluation

We evaluate the performance of NDD in terms of running time and memory cost, and compare to that of using BDD.

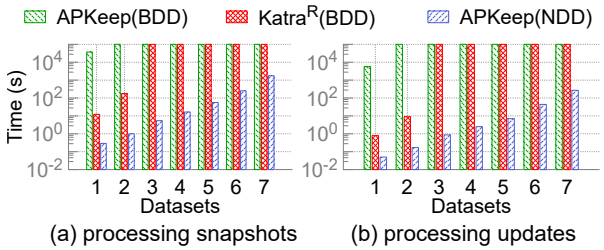


Figure 8: Running time in datacenter networks.

6.1 Setup

Datasets. We use three types of datasets for experiments.

(1) Virtualized datacenter networks (multi-layer networks). We use the virtualized datacenter network datasets (based on VXLAN) to evaluate the performance of NDD on packet transformers. The datasets are synthesized according to production networks [36,55], consisting of 7 different sizes (from 6 to 500 leaf routers), and for each size there are 5 types of updates, e.g., adding a subnet or virtual network. Details can be found in both [36] and Appendix A.2.

(2) WAN and campus networks (single-layer networks). We use three real networks that are extensively used by data plane verifiers [30,33,51,53,57], i.e., the Stanford network [5] of 16 nodes, the Internet2 network [6] of 9 nodes, and the Purdue network [46] of 1646 nodes. The Internet2 has only forwarding rules, while the other two have both forwarding and ACL rules. Details can be found in Appendix A.2.

(3) Fat trees. To evaluate the performance of our NDD libraries for control plane verification, we use different sizes (from 20 to 500 nodes) of fat trees running BGP. The datasets are synthesized and used by SRE [58].

BDD variable ordering. When comparing with data plane verifiers, we use the ordering of `srcIP < dstIP < srcPort < dstPort < protocol` for single-layer networks, and `innerDstIP < srcIP < vni < srcEPG < dstEPG < outerDstIP` for multi-layer networks. When comparing with SRE, we follow the same ordering as SRE, i.e., placing header variables before link variables, and the header variables follow the above ordering as the single-layer networks. For Batfish, we use its default ordering of `dstIP < srcIP < dstPort < srcPort < protocol < flags`.

All experiments run on a server with 2× 12-core Intel Xeon CPUs @ 2.3GHz and 256G RAM (a single core is used).

6.2 Data plane verification: running time

We run the two data plane verifiers, i.e., APKeep and Katra^R, to check all-pair reachability on the multi-layer (virtualized datacenter) networks and the single-layer networks, and compare the running time when using BDD and NDD.

Multi-layer networks (snapshot). Figure 8(a) shows the total time of computing atoms and checking reachability on all snapshots of the virtualized datacenter networks. APKeep(NDD) is the only verifier that runs to complete on

Table 3: Performance in virtualized datacenter networks. Columns 2-4 show memory overhead, columns 5-7 show BDD nodes used for representing atoms, columns 8-10 show the number of atoms and columns 11-13 show the number of new atoms introduced by updates. TO means timeout ($> 24h$).

# Leaf nodes	Memory overhead(GB)			BDD nodes			Atoms			New atoms		
	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**	APKeep (BDD)	Katra^R (BDD)	**APKeep (NDD)**
6	4.36	0.17	**0.01**	1354365	114536	**2195**	28077	5467	**112**	8043	1309	**7**
10	>20	2.88	**0.01**	TO	513069	**3716**	TO	25542	**195**	TO	4062	**10**
20	>32	>115	**0.05**	TO	TO	**8659**	TO	TO	**483**	TO	TO	**14**
50	>80	>115	**0.16**	TO	TO	**19775**	TO	TO	**1173**	TO	TO	**14**
100	>180	>115	**0.45**	TO	TO	**37127**	TO	TO	**2321**	TO	TO	**15**
200	>180	>115	**1.42**	TO	TO	**69934**	TO	TO	**4623**	TO	TO	**15**
500	>180	>115	**7.19**	TO	TO	**161603**	TO	TO	**11544**	TO	TO	**15**

all snapshots, while the other two BDD-based verifiers either time out (>24h), or run out of memory (>256GB). For snapshots that all verifiers finish, APKeep(NDD) is much faster than the other two: for dataset 1 with 6 leaf nodes, APKeep(NDD) runs $>10\times$ and $10^5\times$ faster than Katra^R(BDD) and APKeep(BDD), respectively.

Multi-layer networks (update). Figure 8(b) shows the average time to incrementally check properties after the 5 types of update. Since incrementally checking the updates relies on finishing the check of snapshots, we set the running time of the two BDD-based verifiers to maximum for snapshots that they can not finish. Still, APKeep(NDD) is orders-of-magnitude faster than two BDD-based verifiers.

Single-layer networks (snapshot). Figure 9 shows the running time for the three single-layer networks. We do not include the results for Katra^R, since it has no performance improvement over APKeep on these single-layer networks. We can see that for Stanford and Internet2, the running times of BDD and NDD are comparable, while for Purdue, using NDD is $3\times$ faster than using BDD. The reason for the limited speedup is that these networks mostly match a single field, i.e., dstIP (even Stanford and Purdue have ACL rules that match 5-tuples, the number of ACL rules is quite small). This makes the benefits of decoupling fields less significant than in multi-layer networks.

Single-layer networks (update). To simulate updates on single-layer networks, we choose the Purdue dataset, which has relatively more ACL rules such that the result are more statistically significant. For each ACL rule r, we initially insert all ACL rules other than r, and then insert r and measure the time to update the network model. As shown in Figure 10, the running time for APKeep(BDD) is highly skewed: most updates cost less than 50ms, while 1% of them cost 300ms to 2000ms. In contrast, when using NDD, the update time is always below 150ms. A possible reason is that NDD creates much less atoms after each update: for 7 out of 2707 updates, APKeep(BDD) creates >3000 new atoms (some of them will be merged later), while APKeep(NDD) just creates at most 24 new atoms after each update (see Appendix A.4 for details).

6.3 Data plane verification: memory cost

Table 3 shows the memory usage of different verifiers, computed as the difference of JVM memory usage before and after verification. As shown in Table 3 columns 2-4, the memory usage of APKeep(NDD) is orders-of-magnitude smaller than its BDD counterpart, and also Katra^R(BDD). Specifically, for the network of 6 leaf nodes, the memory cost of APKeep(NDD) is less than $1/100$ and $1/10$ that of APKeep(BDD) and Katra^R(BDD), respectively. The reason that APKeep(BDD) requires 4.36GB memory here is that the network matches 6 fields, leading to an explosion of atoms (28077 atoms creates over 10^6 BDD nodes). As shown in Columns 5-7, the number of BDD nodes of the two BDD-based verifiers is quite large, quickly blowing up the BDD node table (a maximum of $O(2^{32}/3)$ nodes as in JDD), while the number of BDD nodes for APKeep(NDD) is relatively small. This shows the power of NDD in reducing the number of BDD nodes. As for single layer-networks, using NDD reduces the memory cost of APKeep by half on the Purdue dataset, and has comparable memory cost on the Stanford and Internet2 datasets, compared to using BDD.

A major factor for the memory cost is the number of atoms. We continue to study the the number of atoms using NDD and BDD, respectively. Columns 8-10 reports the number of atoms, where the number for APKeep(NDD) is the sum of atoms for each field. As we can see, the number of atoms for APKeep(NDD) increases almost linearly with network size, and is orders-of-magnitude smaller than those of the other two BDD-based verifiers. Columns 11-13 further shows the number of newly created atoms after each update. We can see that BDD-based verifiers may create thousands of atoms after updates, while APKeep(NDD) always creates a rather small number (≤ 15) of atoms after each update. The above results show NDD allow verifier scale better to large networks without explosion of atoms.

6.4 Handling packet transformer

To understand how NDD supports packet transformers, we synthesize a set of NAT rules for the Purdue dataset. We

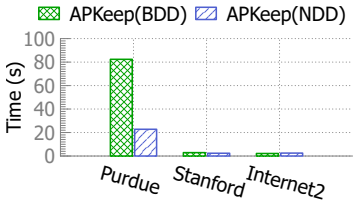


Figure 9: Running time of APKeep on single-layer networks.

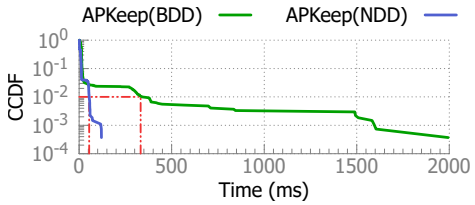


Figure 10: The complement CDF of time for APKeep to update one ACL rule on Purdue dataset.

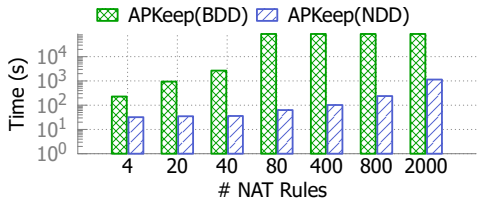


Figure 11: Running time of APKeep on Purdue dataset with NAT rules.

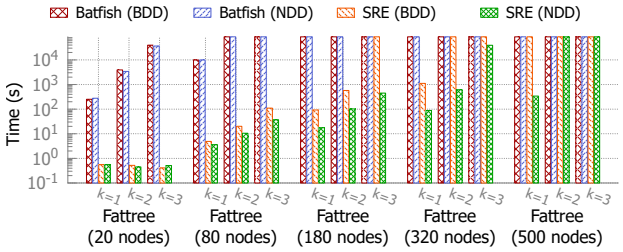


Figure 12: Running time of Batfish and SRE on fattrees.

randomly select k ports not connected to any other, and for each of them add two NAT rules matching srcIP or dstIP , one for each direction. Then, we randomly select another k ports and for each of them add two twice NAT rules (matching srcIP and dstIP simultaneously), one for each direction.

Figure 11 shows the running time of APKeep when using BDD and NDD, respectively. We can see that APKeep(NDD) is 10× faster than APKeep(BDD) even when there are only 4 NAT rules; and 100× faster when 40 NAT rules are inserted. APKeep(BDD) runs out of memory after inserting 80 NAT rules, while APKeep(NDD) runs to complete up to 2000 NAT rules. The reason that NDD can handle packet transformers more efficiently than BDD is NDD computes atoms separately for srcIP and dstIP, leading to much fewer atoms.

6.5 Control plane verification

We run two control plane verifiers, i.e., Batfish and SRE, using both NDD and BDD, to check all-pair reachability on fat tree topologies under $k = 1, 2, 3$ link failures. For Batfish(NDD), we create 4 NDD variables for the 32-bit IP address, each of which has 8 bits. For SRE(NDD), we create 8 NDD variables, each with the same number of bits representing the up/down state of links. An exception is the 500-node fattree, for which we create 16 NDD variables.

As shown in Figure 12, SRE has a better scalability by replacing BDD with NDD: for $k = 1$ failure on the 500-node fattree, SRE(BDD) aborts due to BDD table overflow, while SRE(NDD) can run to complete. We confirm the improved scalability is owing to the reduction of BDD nodes (Appendix A.5). For Batfish, however, the improvement of scalability is limited by using NDD. This is because the datasets only use a single field, i.e., dstIP, and Batfish doesn't have link variables as SRE. While we still find the number of BDD nodes in Batfish is reduced by using NDD (Appendix A.5). The reason is the synthesized IPs in this dataset share the same 16-bit prefix, i.e., 10.0.∗.∗. Using BDD, all packet sets need a copy of BDD nodes for those 16 bits (these nodes can not be shared by the packet sets since they differ in the last 16 bits).

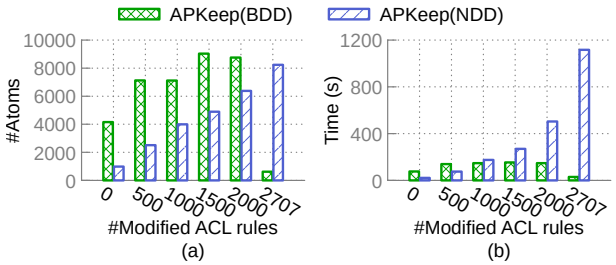


Figure 13: NDD vs. BDD on modified Purdue dataset, with different number of ACL rules matching all the 5 fields.

6.6 The importance of field locality

To study the importance of field locality for the effectiveness of NDD, we modified ACL rules in the Purdue dataset, so that more rules will match all the five fields (against randomly generated values). As shown in Figure 13, both the number of atoms and the running time of APKeep(NDD) grow linearly with the number of modified rules. In contrast, the number and time of APKeep(BDD) significantly decrease when all ACL rules match all fields. The reason is that when all rules match all fields, the phenomenon of field locality disappears, so the number of atoms when using BDD is not so large. This implies that the performance gain of NDD relies on the assumption of field locality, which generally hold in networks, but can theoretically fail (as in this synthesized scenario).

7 Related Work

Besides the classic BDD, researchers have proposed many variants of Decision Diagrams.

Decision diagrams with multiple terminals. Multi Terminal BDD (MTBDD) [12, 22], also termed Algebraic Decision Diagram (ADD), is a variant of BDD but with more than two terminals. MTBDD can model multiple outcomes (rather than true/false) of decisions, and have been used by some network

verifiers [16, 28, 35]. However, since MTBDDs still branch at a single bit, they acknowledge the same limitations of BDD when applied to network verification.

Decision diagrams branching on constraints. Forwarding Decision Diagram (FDD^b) [42, 43] is proposed to compile NetKat programs (a set of `IF-THEN-ELSE` clauses). Each FDD^b node has two edges representing whether a packet satisfies a *specific* constraint, e.g., $dstIP = 10.0.0.1$, or not. When using it for network verification, we may need many levels of nodes for each field, e.g., another level for $dstIP \in 192.168.0.0/16$. Constrained Decision Diagram (CDD) [21] is proposed to represent all solutions to a constraint satisfaction problem (CSP). Different from FDD^b, a CDD node can have multiple edges, each representing a SMT constraint, e.g., $x_1 < x_2 + 4$. In this sense, CDD represents a set of SMT constraints in the form of a decision diagram, but without solving them. To obtain a truth assignment, CDD still needs to try all possible values for variables in the constraints of each node (`cddFindall` algorithm [21]). This can be inefficient when the domain of the variable is large, e.g., if we encode `dstIP` with single variable, CDD needs to try 2^{32} values for each edge labeled by a constraint on the variable of `dstIP`.

Decision diagrams branching on a field. Multi-valued Decision Diagram (MDD) is a diagram that branches on multiple bits each time [44]. However, MDD has an edge for each single value of these bit. This which makes MDD not scalable for network verification, since a field often has many bits (e.g., a 32-bit IPv4 address requires 2^{32} edges). Similarly, Interval Decision Diagram (IDD) [45] and Firewall Decision Diagram (FDD^a) [29] branch based on one or multiple ranges of a field. However, network verifiers often need to reason about arbitrary sets of packets, which lead to a huge number of disjoint ranges, and using IDD or FDD^a is not scalable. NDD differs from the above DDs by using BDDs for a relatively small number of edges for each field.

Other implementations of BDD. Apart from JDD and JavaBDD, there are other excellent implementations of BDD. For example, Sylvan [49] is a BDD implementation which supports parallelism in both operation and algorithm level. Decision Diagram [1] is a native .NET implementation of BDD with better performance than many other BDD libraries. It is used by ZEN [17], a general and compositional framework for network modeling. However, the above implementations still suffer the memory/computation inefficiency when applied to network verification, as discussed in this paper.

Other data structures. Early verifiers like Veriflow [34] and HSA [33] use Trie and ternary bit vectors (TBV), respectively, to represent packet sets. However, such data structures cannot compactly encode arbitrary disjoint ranges. BDD-based verifiers like AP Verifier [52] and APKeep [57] have been shown to be orders of magnitude faster than HSA (based on TBV) and Veriflow (based on Trie). More recently, researchers proposed ddNF [18] and #PEC [31], which uses disjunction normal form to represent packet sets. All these data structures may perform well in some specific settings, but are still unable to supersede BDDs in general scenarios.

In sum, existing data structures are either not compact or inefficient for multiple fields: BDD is compact but the bit-level encoding makes it inefficient for multiple fields; other structures like Trie, CDD, MDD, etc. are more efficient for multiple fields by encoding at field level, but they encode each field with integers, ranges, constraints, etc., are thus not compact. In addition, when applied to network verification, all these data structures require verifiers to implement their own computation of ECs over the whole network state, resulting in a large number of ECs.

Theoretically, NDD can be seen as a special form of symbolic automata [23], where the transitions among nodes are boolean formulas. This paper is not fundamentally contributing to such theory, but provides a new design of design diagram library customized for network verification tasks.

8 Discussion

Extending NDD with multiple terminals. Besides the standard BDD, the Multi-Terminal BDD (MTBDD) [12, 22] has also been used by some network verifiers, e.g., ShapeShifter [16], YU [35], ProbNV [28], etc. For example, YU uses MTBDD to model the different (multiple) traffic volumes on a link, under different failures, so as to verify whether the link will be overloaded under some failures. We tentatively think NDD can be extended to support multiple terminals, such that it can apply to a broader range of network verifiers.

How to partition fields? For network verification, partitioning the packet header into to multiple fields, and encoding each field with an NDD variable achieves good performance. For more general scenarios where fields may not be quite clear (links as in SRE), we leverage some heuristics: (1) variables encoding a complete semantics together should be partitioned into the same field, (2) variables appearing in the same operand of BDD operations should be partitioned into the same field.

9 Conclusion

This paper introduced Network Decision Diagram, a new decision diagram customized for network verification. We designed and implemented a library for NDD, and show it can significantly reduce the memory and time for five BDD-based network verifiers. Our future work includes extending NDD with multiple terminals, making NDD library distributed and parallel, and testing the NDD library with more verifiers.

Acknowledgement. We thank our shepherd Ryan Beckett, and all the anonymous NSDI reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No. 62272382). Peng Zhang is the corresponding author of this paper.
This work does not raise any ethical issues.

References

[1] A native .NET implementation of BDDs. https://github.com/microsoft/DecisionDiagrams.

[2] AP Verifier. https://www.cs.utexas.edu/users/lam/NRL/Atomic_Predicates_Verifiers.html.

[3] APKeep. https://github.com/XJTU-NetVerify/apkeep.

[4] Batfish. https://github.com/batfish/batfish/.

[5] Hassel, the header space library. https://bitbucket.org/peymank/hassel-public.

[6] Internet2. http://www.internet2.edu.

[7] JavaBDD, a Java BDD library. https://javabdd.sourceforge.net/.

[8] JDD, a pure Java BDD and Z-BDD library. https://bitbucket.org/vahidi/jdd/.

[9] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast multilayer network verification. In *USENIX NSDI*, 2020.

[10] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *IEEE ICNP*, 2009.

[11] H. R. Andersen. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen*, page 5, 1997.

[12] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10:171–206, 1997.

[13] R. Beckett and A. Gupta. Katra: Realtime verification for multilayer networks. In *USENIX NSDI*, 2022.

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[15] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Control plane compression. In *ACM SIGCOMM*, 2018.

[16] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Abstract interpretation of distributed network control planes. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

[17] R. Beckett and R. Mahajan. A general framework for compositional network modeling. In *ACM Workshop on Hot Topics in Networks*, 2020.

[18] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese. ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference*, 2016.

[19] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *ACM SIGCOMM*, 2023.

[20] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[21] K. C. Cheng and R. H. Yap. Constrained decision diagrams. In *AAAI*, volume 5, pages 366–371, 2005.

[22] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *International Design Automation Conference*, 1993.

[23] S. Drews and L. D'Antoni. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017.

[24] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*, 2016.

[25] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX NSDI*, 2015.

[26] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, 2016.

[27] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker. NV: an intermediate language for verification of network control planes. In *ACM PLDI*, 2020.

[28] N. Giannarakis, A. Silva, and D. Walker. ProbNV: probabilistic verification of network control planes. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, 2021.

[29] M. G. Gouda and X.-Y. Liu. Firewall design: Consistency, completeness, and compactness. In *IEEE ICDCS*, 2004.

[30] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[31] A. Horn, A. Kheradmand, and M. R. Prasad. A precise and expressive lattice-theoretical framework for efficient network verification. In *IEEE ICNP*, 2019.

[32] S. K. R. Kakarla, A. Tang, R. Beckett, K. Jayaraman, T. Millstein, Y. Tamir, and G. Varghese. Finding network misconfigurations by automatic template inference. In *USENIX NSDI*, 2020.

[33] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[34] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[35] R. Li, Y. Yuan, F. Ye, M. Liu, R. Yang, Y. Yu, T. Guo, Q. Ma, X. Zeng, C. Xu, et al. A general and efficient approach to verifying traffic load properties under arbitrary k failures. In *ACM SIGCOMM*, 2024.

[36] X. Liu, P. Zhang, H. Li, and W. Sun. Modular data plane verification for compositional networks. *Proceedings of the ACM on Networking*, 1(CoNEXT3):1–22, 2023.

[37] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *USENIX NSDI*, 2015.

[38] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical report, 2014.

[39] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

[40] C. Perkins. IP encapsulation within IP. Technical report, 1996.

[41] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*, 2020.

[42] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for netkat. In *ACM SIGPLAN International Conference on Functional Programming*, 2015.

[43] S. Smolka, P. Kumar, D. M. Kahn, N. Foster, J. Hsu, D. Kozen, and A. Silva. Scalable verification of probabilistic networks. In *ACM PLDI*, 2019.

[44] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *IEEE international conference on computer-aided design*, 1990.

[45] K. Strehl and L. Thiele. Interval diagrams for efficient symbolic verification of process networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):939–956, 2000.

[46] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards systematic design of enterprise networks. In *ACM CoNEXT*, pages 1–12, 2008.

[47] A. Tang, S. K. R. Kakarla, R. Beckett, E. Zhai, M. Brown, T. Millstein, Y. Tamir, and G. Varghese. Campion: debugging router configuration differences. In *ACM SIGCOMM*, 2021.

[48] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *ACM SIGCOMM*. 2019.

[49] T. Van Dijk and J. Van de Pol. Sylvan: multi-core framework for decision diagrams. *International Journal on Software Tools for Technology Transfer*, 19:675–696, 2017.

[50] D. Wang, P. Zhang, and A. Gember-Jacobson. Expresso: Comprehensively reasoning about external routes using symbolic simulation. In *ACM SIGCOMM*, 2024.

[51] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[52] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

[53] H. Yang and S. S. Lam. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking*, 25(5):2900–2915, 2017.

[54] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *ACM SIGCOMM*, 2020.

[55] L. You, J. Zhang, Y. Jin, H. Tang, and X. Li. Fast configuration change impact analysis for network overlay data center networks. *IEEE/ACM Transactions on Networking*, 30(1):423–436, 2021.

[56] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li. Differential network analysis. In *USENIX NSDI*, 2022.

[57] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. APKeep: Realtime verification for real networks. In *USENIX NSDI*, 2020.

[58] P. Zhang, D. Wang, and A. Gember-Jacobson. Symbolic router execution. In *ACM SIGCOMM*, 2022.

*The appendices are subject to changes, and the latest version can be found at https://xjtu-netverify.github.io/papers/2025-ndd-a-decision-diagram-for-network-verification.

A More Evaluation Results

A.1 Implementation Details

In the following, we give more details on the implementations of the 5 verifiers with the NDD library.

AP Verifier [51] and APT [53]. AP Verifier is a pioneering work that pre-compute atomic predicates based on BDD, and APT extends AP Verifier to support packet transformers. We used their open-source codes [2], and modified them to use the NDD library. The modifications are to remove their implementations of computing atomic predicates, and re-implement it with the `atomize` API of NDD.

APKeep [57]. APKeep is a data plane verifier that extends AP Verifier to incrementally update atomic predicates, also based on BDD. We used its source codes [3], and modified it to use the NDD library. The modifications are to remove its previous implementation of incremental update of atomic predicates, and re-implement it with the `update` API of NDD.

Katra [13]. Katra is a data plane verifier proposed for multi-layer networks. Katra introduces a novel *partial equivalence classes* to significantly reduce the total number of ECs, and is proven to be more efficient than APKeep when verifying multi-layer networks. We are interested in whether we can achieve the same or even better efficiency as Katra, by simply changing the underlying data structure, without computing partial equivalence classes. Therefore, we re-implemented Katra based on BDD, referred to as Katra^R, and further extended it to support VXLAN overlay networks in Table 5. We confirmed the correctness of Katra^R based on the datasets synthesized according to [13]. However, we did not implement the NDD version of Katra since the core difference of Katra from APKeep is its novel *partial equivalence classes* based on BDD, and replacing BDD with NDD will make Katra no different from APKeep (NDD).

SRE [58]. SRE is a control plane verifier that jointly encodes failure scenarios and packet headers, so as to scale to large networks. We used the authors' implementation and replaced its JDD library with our NDD library. For packet headers, we encode each header field with an NDD variable; for failure scenarios, we divide the set of all links evenly into 8 groups, and encode each link group with an NDD variable.

Batfish [19, 25]. Batfish is an open-source control plane verifier, which uses JavaBDD [7]. We replaced the BDD factory in JavaBDD with our NDD factory, with small modifications of Batfish, including declaring fields and initializing the unique table and operation cache.

A.2 Statistics of datasets

Table 4 and Table 5 show the experimental datasets.

Table 4: Datasets for WAN and campus networks.

Networks	Nodes	Links	Forwarding rules	ACL rules
Purdue	2,159	3,607	3.52×10^6	2707
Stanford	124	182	3.84×10^3	686
Internet2	9	56	1.26×10^5	0

Table 5: Datasets for virtualized datacenter networks.

Networks	Leafs (VPCs)	Subnets (VRFs)	Forwarding rules	MCS rules	Static routes
1	6	36	856	12	12
2	10	100	2,508	20	20
3	20	400	13,428	40	40
4	50	1,000	59,808	100	100
5	100	2,000	144,508	200	200
6	200	4,000	388,908	400	400
7	500	10,000	1,528,538	1,000	1000

A.3 Micro-benchmark results

Table 6: Running time of each stage and the number of atoms for forwarding and ACL rules on Purdue dataset.

Verifiers	IPv4		ACL		Property
	Time(s)	Atoms	Time(s)	Atoms	Check(s)
APKeep(BDD)	13.88	265	22.37	3893	46.11
APKeep(NDD)	14.87	265	7.04	411	0.88

The per-stage running time for Purdue is shown in Table 6. NDD uses more time than BDD when processing forwarding rules (Column 2), due to the overhead of maintaining per-field atoms; while NDD uses roughly $1/3$ the time of BDD when processing ACL rules (Column 4), since the ACL rules match 5-tuples. For checking properties, NDD takes much less time than BDD (Column 6). This is because using BDD will result in much more atoms than using NDD, as shown in column 5: by using NDD, APKeep can reduce the number of atoms by around 89.4%.

A.4 Number of new atoms after each update

Figure 14 shows the complement cumulative distribution of the number of new atoms for each update in Purdue dataset.

A.5 Number of BDD nodes for Control Plane Simulation

Figure 15 shows the number of used BDD nodes for SRE and Batfish on the fattree datasets. The number of BDD nodes of Batfish(BDD) is many times more than that of Batfish(NDD), and the number of BDD nodes of SRE(BDD) is more than that of SRE(NDD) by orders of magnitude, which shows memory inefficiency of BDD.

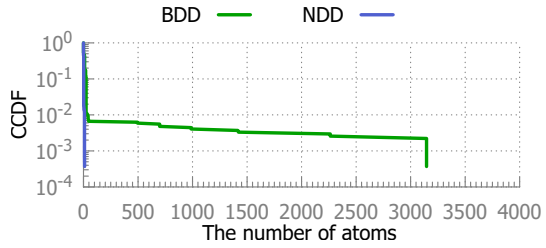


Figure 14: The complement cumulative distribution of the number of new atoms for each update in Purdue dataset.

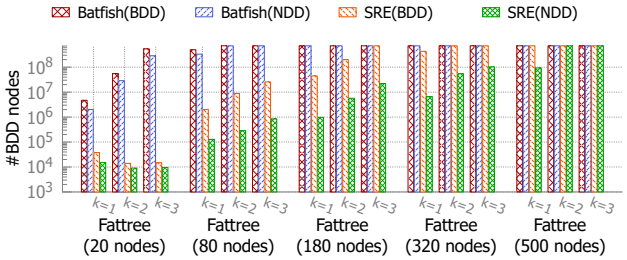


Figure 15: The number of used BDD nodes for control plane simulation on fattree networks.

A.6 Number of edges in NDD

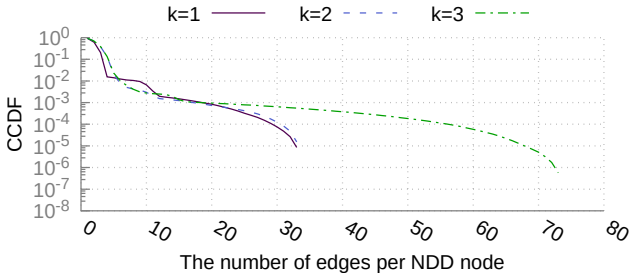


Figure 16: The complement cumulative distribution of the number of edges for each NDD node.

Figure 16 shows the number of edges for each NDD node created by SRE(NDD) on the 80-nodes fattree network. More than 99% of the nodes have no more than 10 edges, and all the nodes have no more than 75 edges, which shows that NDD can efficiently merge redundant edges with a high compression ratio.

A.7 Number of atoms on edges to `false`

Figure 17 (a) shows the number of atoms that are labeled on each edge pointing to the terminal node `false`. We can see that almost 30% of these edges have more than 2000 atoms. Figure 17 (b) shows the percentage of atoms on edges pointing to the `false` node to the total number of atoms of the corresponding field. We can see that for 85% of these edges have more than 95% of atoms of the corresponding field.

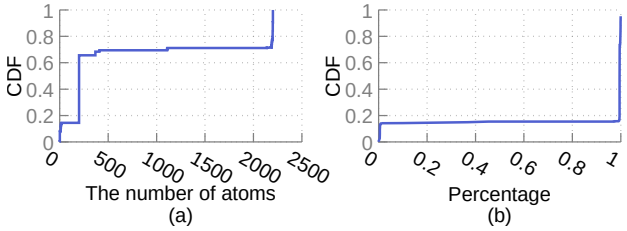


Figure 17: (a) the number and (b) the percentage of atoms on edges pointing to `false`.

Table 7: Time to solve the NQueens problem.

N	6	7	8	9	10	11	12
BDD (s)	0.017	0.023	0.04	0.223	0.615	2.567	19.109
NDD (s)	0.012	0.019	0.038	0.176	0.344	2.257	12.417

A.8 Performance of NDD for general problems

We compare the performance of BDD and NDD in solving the NQueens problem, a widely recognized benchmark for BDD libraries. For BDD, we use the implementation of the JDD library, which is basically an implementation of algorithms in [11]. For NDD, we declare each row as a field (therefore there are N fields), and implement the same algorithm. Table 7 shows that the time of BDD is around 1.5× that of NDD for $N = 12$. This is because the BDD library has an considerable overhead for aligning variables when computing BDD conjunctions, especially for large N. This indicates the phenomenon of "field locality" may not be limited to network verification, and NDD may be used to speedup other applications.

A.9 BDD variable orderings

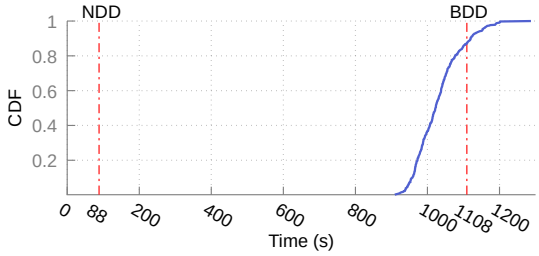


Figure 18: Running time of SRE on fattree (320 nodes, $k = 1$) with random variable orderings. The two dotted lines represent the running time of SRE(BDD) and SRE(NDD) reported in Figure 12, respectively.

As briefly mentioned in §2.2, one may wonder whether we can mitigate limitations of BDD with a better variable ordering. We discuss this for two different types of verifiers.

First, for verifiers that compute atoms, e.g., AP Verifier, APT and APKeep, the performance is highly related to the number of atoms, while little affected by the variable ordering. Therefore, variable ordering makes little differences to their

performance when suffering explosion of atoms.

Second, for verifiers that do not compute atoms, e.g., SRE, variable ordering indeed impact the performance. Here, we use SRE(BDD) as an example to study such impact. We found an ordering with header variables (`dstIP`) before link variables, which was used by SRE, always leads to better performance. Therefore, when we run SRE(BDD), we first declare the variables of `dstIP`, and then declare the link variables with random order. Figure 18 reports the running time, where we can see the time varies from 800s to 1200s with different orderings, but is still much larger than that of SRE(NDD). This confirms that re-ordering the variables does not fundamentally make BDD more efficient for network verification.

A.10 Comparing NDD to BDD arrays

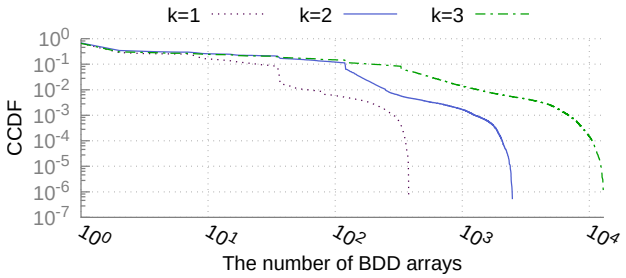


Figure 19: The complement cumulative distribution of the number of BDD arrays to represent a predicate in SRE.

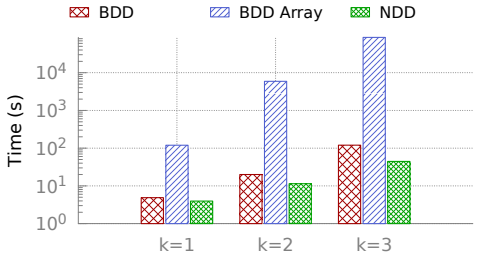


Figure 20: Running time on 80-node fattree of verifiers based on three methods.

We implement SRE based on BDD arrays, which simply assembles per-field BDDs by conjunctions, and compare its performance to that based on NDD and BDD, on a 80-node fattree. Figure 19 shows the number of BDD arrays representing each predicate, which is as large as $10^2, 10^3, 10^4$ for $k = 1, 2, 3$, respectively. Figure 20 further shows the running time. We can see that using BDD arrays is orders-of-magnitude slower than using NDDs, and even slower than directly using BDDs. Especially when $k = 3$, using BDD arrays times out due to too many BDD arrays.

B Algorithms of NDD operations

Algorithm 1 shows the implementation of `apply`(op, a, b), where `op` can be `and`, `or`, `diff`; a and b are two NDDs. For trivial cases such as either a or b is a terminal node, or $a = b$, the algorithm directly returns the result (Lines 1-2). If the variable of a and b is the same (Line 3), the algorithm recursively performs `apply` for each successor of a and each successor of b (Lines 4-5). Note that, the algorithm merges redundant edges in Line 9. If the variable of a has a higher order, the algorithm recursively performs `apply` for b and each successor of a (Lines 10-15). Finally, it looks for an NDD node (or create a new node) from node table representing the result and returns it (Line 16). The algorithm for atomized NDDs is the same with the above, except that BDD operations of *and* and *or* in Line 6, 9 and 15 change to set operations of *intersect* and *union*. Note that, when we enable the optimization of removing the `false` node (§4.2), the implementations of `or` and `diff` need some minor modifications.

Algorithm 1: `apply`$(op,\ a,\ b)$

Input: a logical operator $op \in \{and, or, diff\}$, and two NDDs a and b.
Output: the result of logical operation $a\ op\ b$.

```
1  if a or b is a terminal node, or a = b then
2      return handleTrivialCases(op, a, b);
3  if var(a) = var(b) then
4      foreach e_a ∈ edges(a) do
5          foreach e_b ∈ edges(b) do
6              label ← bdd.and(label(e_a), label(e_b));
7              if label ≠ bdd.false then
8                  next ← apply(op, next(e_a), next(e_b));
9                  edgeMap[next] ←
                     bdd.or(edgeMap[next], label);
10 else
11     if var(a) > var(b) then
12         swap(a, b);
13     foreach e_a ∈ edges(a) do
14         next ← apply(op, next(e_a), b);
15         edgeMap[next] ← bdd.or(edgeMap[next],
             label(e_a));
16 return mk(var(a), edgeMap);
```

Algorithm 2 shows the implementation of `exist`. If the field variable of a has lower priority than $field$, the algorithm directly returns a as the result (Lines 1-2). If the field variable of a is equal to $field$, the algorithm applies existential quantification by merging all successors of a through NDD's function of `or` (Lines 3-7). If the field variable of a has higher priority than $field$, the algorithm invokes a new recursion for each successor of a and merges results (Lines 8-12).

Algorithm 3 shows the implementation of `atomize`. It takes

Algorithm 2: `exist`(a, $field$)

Input: an NDD a and a variable $field$.
Output: an NDD with existential quantification on a for variable $field$.

```
1  if var(a) > field then
2      return a;
3  else if var(a) = field then
4      sum ← ndd.false;
5      foreach e ∈ edges(a) do
6          sum ← apply(or, sum, next(e));
7      return sum;
8  else
9      foreach e ∈ edges(a) do
10         next ← exist(next(e), field);
11         edgeMap[next] ← bdd.or(edgeMap[next],
               label(e));
12     return mk(var(a), edgeMap);
```

Algorithm 3: `atomize`(N)

Input: a set of NDDs N.
Output: a set of atomized NDDs N' that are logically equivalent to N.

```
1  labels ← allLabels(N);
2  foreach field ∈ allFields(N) do
3      atoms[field] ← computeAtoms(labels[field]);
4  N' ← {};
5  foreach n ∈ N do
6      N' ← N' ∪ atomizeNDD(n, atoms);
7  return N';
```

a set of NDDs N to be atomized as input and returns a set of atomized NDDs N', which are logically equivalent to N. The algorithm first collects all labels (BDDs) used by some of the input NDDs, and group these labels by field (Line 1). Then, the algorithm computes atoms for each field (Lines 2-3), in a similar way as AP Verifier does. After that, it transforms the each input NDD into a logically-equivalent atomized NDD (Lines 4-6).

Algorithm 4 shows the implementation of `update`. It takes as input an NDD δ and an atomized NDD a, δ is a predicate to be atomized and a is used to accelerate the function. *split* in Line 2 is a map that maps each old atom to be split to 2 new atoms. The algorithm first recursively traverses δ and a, and detects the atoms to be split (Line 3), then performs the split by replacing each old atom in labels of existing atomized NDDs with 2 new atoms in *split* (Line 4). Since an atom may be split many times by different label BDDs in δ, the function repeatedly performs steps above until *split* is empty after Line 3. If the δ and a have the same variable, for each edge e_δ of δ and each edge e_a of a, the function intersects the BDD on e_δ with only atoms held by e_a to detect atoms to be split (Lines

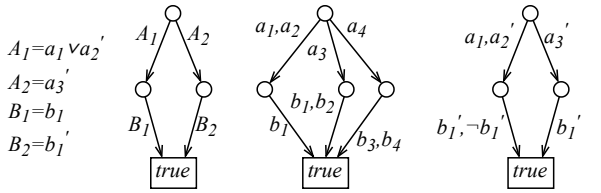
Algorithm 4: `update`(δ, a)

Input: an NDD δ to be atomized; an atomized NDD a satisfying $\delta \Rightarrow a$.

```
1  do
2      split ← {} ;
3      updateRec(δ, a, split);
4      splitAtoms(split);
5  while ¬split.isEmpty();
6  Function updateRec(δ, a, split):
7      if δ = ndd.true then
8          return;
9      else if var(δ) = var(a) then
10         foreach e_δ ∈ edges(δ) do
11             foreach e_a ∈ edges(a) do
12                 hit ← false;
13                 foreach ap ∈ label(e_a) do
14                     t ← bdd.and(ap, label(e_δ));
15                     if t ≠ bdd.false then
16                         hit ← true;
17                         if t ≠ ap then
18                             split[ap] ←
                                   {t, bdd.diff(ap, t)};
19                 if hit then
20                     updateRec(next(e_δ), next(e_a), split);
21     else if var(δ) < var(a) then
22         foreach e ∈ edges(δ) do
23             foreach ap ∈ atoms[var(δ)] do
24                 t ← bdd.and(ap, label(e));
25                 if t ≠ false and t ≠ ap then
26                     split[ap] ← {t, bdd.diff(ap, t)};
27             updateRec(next(e), a, split);
```

9-20). Note that, it only traverses successors of edges whose atoms intersect with the label (Lines 19-20). If the variable of δ has higher priority than that of a, the algorithm intersects the labels with each atom of $var(\delta)$ (Lines 21-27).

Figure 21 shows an example for `update`, where (b) is an NDD δ to be atomized, and (c) is an atomized NDD a. Labels on δ are per-field BDDs and labels on a are per-field atoms. Relations of per-field BDDs and atoms are shown in (a), where $A_1 = a_1 \vee a_2'$ means A_1 contains (is implied by) a_1 and partially intersects (has a non-false conjunction) with a_2. The operation first chooses the left edge of δ and a, computes conjunctions of A_1 with a_1 and a_2, respectively, and splits a_2 into new atoms a_2' and $\neg a_2'$, such that A_1 can be atomized. Then, the operation tries to match the edges labeled by B_1 and b_1 in the second field. After that, the operation tries to match the left edge of δ and the middle and right edges of a, respectively, finds that A_1 doesn't intersect with a_3 or a_4, and skips calculation in the second field. As shown above, A_1



(a) Assumptions (b) NDD δ (c) atomized NDD a (d) atomized NDD δ

Figure 21: An example for `update` operation. A_i and B_i are BDDs; a_i and b_i are atoms. $a_i^{'}$ means it implies a_i, i.e., $a_i^{'} \Rightarrow a_i$.

only needs to intersect with atoms a_{1-4} which appear on the paths from a to terminal `true`, rather than all atoms of the field (e.g., atoms on the edge of a pointing to terminal `false`, which are not shown in the figure), thereby accelerating the operation. The steps are similar for the right edge of δ. Finally, atoms a_2, a_3 and b_1 are split, such that δ can be atomized as shown in (d).

C Proof of canonicity of NDD

Definition 4. ***Boolean formulas.*** *A boolean formula F has $n_1+n_2+\cdots+n_k$ boolean variables, which can be grouped into k fields $f_1, f_2, \ldots, f_k$, where field f_i has n_i consecutive boolean variables $x_{s_i+1}, x_{s_i+2}, \ldots, x_{s_i+n_i}$, where $s_i = n_1+n_2+\cdots+n_{i-1}$.*

Definition 5. ***Truth assignments and restrictions.*** *A truth assignment of a field f_i is denoted as $B_i = [b_{s_i+1}, b_{s_i+2}, \ldots, b_{s_i+n_i}]$, where $b_j \in \{true, false\}$. A restriction of F with respect to a variable x and a boolean constant b, denoted as $F[b/x]$, is defined as the same formula F, except that the variable x is replaced with b. Similarly, we can extend the definition to a field f_i and a truth assignment B_i, as*

$$F[B_i/f_i] = F[b_{s_i+1}/x_{s_i+1}, b_{s_i+2}/x_{s_i+2}, \ldots, b_{s_i+n_i}/x_{s_i+n_i}]$$

Suppose bdd is a BDD defined over f_i, and $\forall B_i \in bdd$, $F[B_i/f_i]$ is the same, then define the restriction of F with respect to field f_i and BDD bdd as $F_{bdd} = F[B_i/f_i], \forall B_i \in bdd$.

Shannon expansion is the theoretical basis of BDD: a BDD node encoding F branches on x, with a *high* edge assigning `true` to x and pointing to the successor encoding $F[1/x]$, and a *low* edge assigning `false` to x and pointing to the successor encoding $F[0/x]$.

Definition 6. ***Single-bit Shannon expansion.*** *First define the if-then-else operator as $x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$. Then, the Shannon expansion of formula F with respect to a boolean variable x is defined as:*

$$F = x \rightarrow F[1/x], F[0/x].$$

Since each NDD variable (i.e., field) has multiple bits, we extend the above single-bit Shannon expansion to the following multi-bit Shannon expansion.

Definition 7. ***Multi-bit Shannon expansion.*** *Let $\mathcal{BDD}$ be a set of BDDs satisfying $\bigvee_{bdd \in \mathcal{BDD}} bdd = true$, and $\forall bdd_1, bdd_2 \in \mathcal{BDD}$, $bdd_1 \neq bdd_2 \Rightarrow bdd_1 \wedge bdd_2 = false$. Then, the Shannon expansion on formula F with respect to field f_i is:*

$$F = \bigvee_{bdd \in \mathcal{BDD}} (bdd \wedge F_{bdd}) = f \rightarrow F_{bdd_1}, F_{bdd_2}, ..., F_{bdd_n}.$$

Definition 8. ***The formula encoded by an NDD.*** *Suppose u is an NDD node having a set of edges $e_1, e_2, \ldots, e_n$, Then, u encodes a boolean formula F^u defined as:*

$$F^u = var(u) \rightarrow F_{label(e_1)}, F_{label(e_2)}, ..., F_{label(e_n)}. \quad (2)$$

Lemma 2. ***Canonicity of NDD.*** *For any boolean function $F(x_1, x_2, \cdots, x_N)$ over N boolean variables with ordering $x_1 < x_2 < \cdots < x_N$, there is exactly one RONDD u with k NDD variables and ordering $f_1 < f_2 < \cdots < f_k$, where f_i has n_i bits $x_i^1 < x_i^2 < \cdots < x_i^{n_i}$, and $N = \sum_{i=1}^{k} n_i$, such that u can represent F.*

Proof. Clearly the lemma holds for $k = 0$, and for $k \geq 1$, we prove it by induction on k. The proof is adapted from that for BDD canonicity [11].

Assume now that we have proven the lemma for k fields $(f_2, f_3, \cdots, f_{k+1})$. We prove the lemma for $k+1$ fields $(f_1, f_2, \cdots, f_{k+1})$. For a boolean function F with $k+1$ fields, the following two steps proves (1) there exists an NDD node u that can represent F, and (2) if there is another NDD node v, such that $F^v = F$, then we must have $v = u$.

(1) The existence of an NDD for boolean formula F. For f_1 with n_1 bits, there are 2^{n_1} distinct truth assignments, represented as a set $\mathcal{B}$. Each $B \in \mathcal{B}$, the restriction $F[B/f_1]$ has no more than k bits, and thus there exists an NDD node that can represent it (according to our assumption for k). We can partition $\mathcal{B}$ into m classes $C_1, C_2, \ldots, C_m$, satisfying that $\forall B_1, B_2 \in \mathcal{B}, F[B_1/f_1] = F[B_2/f_1] \Leftrightarrow \exists C_i, B_1 \in C_i, B_2 \in C_i$. For each C_i, there exists an NDD node v_{C_i} such that $\forall B \in C_i$, v_{C_i} represents each $F[B/f_1]$, and a BDD bdd_{C_i} that can represent $\bigvee_{B \in C_i} B$. We create an NDD node u with $var(u) = f_1$ and m edges, satisfying that $\forall e_i \in edges(u) \Rightarrow label(e_i) = bdd_{C_i}, next(e_i) = v_{C_i}$. According to Definitions 7 and 8, we have

$$F = f_1 \rightarrow F_{bdd_{C_1}}, F_{bdd_{C_2}}, \ldots, F_{bdd_{C_m}} = F^u$$

, meaning node u represents the formula F.

(2) The uniqueness of NDD for boolean formula F. There are two cases to consider.

(i) u has a single edge e with $label(e) = true$ and $next(e) = v_{C_1}$. This violates RONDD's condition of **no redundant node**. By reducing redundant nodes, u degenerates to v_{C_1}, which is canonic (canonicity of NDD with $\leq k$ fields).

(ii) u has more than one edges. Assume v is an NDD node with $F^v = F$, and we need to show $u = v$. First, we must have $var(v) = f_1$; otherwise if $var(v) \neq f_1$, then the

assignment of f_1 doesn't affect F, and u must be an NDD node with a single edge, which violates the assumption that u is a RONDD. For $\forall e_u \in edges(u)$ and a truth assignment $\forall B_1 \in label(e_u)$, we can find an edge $e_v \in edges(v)$ such that $B_1 \in label(e_v)$, because v must encode any truth assignment in f_1. In addition, we have $next(e_u) = next(e_v)$ because these two nodes encode the same formula $F[B_1/f_1]$ (canonicity of NDD with $\leq k$ fields). For any other truth assignment $\forall B_2 \in label(e_u), B_2 \neq B_1$, we can also find an edge $e_{v'} \in edges(v)$ such that $B_2 \in label(e_{v'})$. We can get that $next(e_v) = next(e_u) = next(e_{v'})$, and if $e_v \neq e_{v'}$, it violates RONDD's condition of **no redundant edges**. Therefore, we get that $e_v = e_{v'}$. That means, for $\forall e_u \in edges(u)$, there must exist $e_v \in edges(v)$, such that $label(e_u) \Rightarrow label(e_v)$. Vice versa, we get $label(e_v) \Rightarrow label(e_u)$ and therefore $label(e_u) = label(e_v)$ and $next(e_u) = next(e_v)$. Therefore, v and u have the same variable and set of edges. According to the **uniqueness** of RONDD, it follows that $v = u$.

□

D An example for explosion of atoms

We use an example in Figure 22 to show why the number of atoms can explode when using BDDs. For simplicity of illustration, we consider only `dst IP/Port` fields, and suppose ACL rules at port 1 of D have not been inserted.

Using BDD, a data plane verifier, i.e., APKeep, works in following steps. First, it splits each node (device) in (a) into *elements* in (b), each of which represents a single action in networks, e.g., forwarding, filter. Then, the verifier encodes match conditions of rules with BDDs, and computes port predicates for each element. The predicates are shown in (c) with labels of $B_1 - B_6$. Packets encoded in a port predicate will be forwarded to the related port or pass the related filter. After that, the verifier computes atoms in (d), each of which represents a set of packets with the same global behaviour. Suppose for now there are only forwarding rules matching `dstIP`. There are only 3 different behaviors, which correspond to $dstIP \in$ 00, 01 and 1*, respectively. Suppose there are also ACL rules matching `dstPort`, and the number of behaviours (atoms) will increase to 3 × 3 = 9, a cross-product of the 3 different behaviors for each of the two fields (B_{1-3} for `dstIP`, B_4, $B_5 - B_4$ and $\neg B_5$ for `dstPort`). As shown in (d), the three sub-structures of BDD nodes for `dstIP` are combined with the three sub-structures of BDD nodes for `dstPort`, resulting in 20 nodes in total. As shown in (e), the verifier replaces each port predicate in (b) with an equivalent set of atoms in (d). Then, as shown in (f), the verifier injects all atoms into A and traverses the network with a set of reachable atoms. At each element, it intersects the set of reachable atoms with atoms of each port of the element by using set operation $\cap$ and $\cup$, and forwards the result to the next element. Finally, atoms of $A_{1-3,5,6,8}$ can pass port 1 of D from A.

Using NDD, in contrast, the data plane verifier also computes port predicates with logical operations of $\wedge$, $\vee$ and $\neg$ as shown in (g), whose edges are labeled by BDDs in (c). Then, it collects all per-field BDDs on the edges, and calculates per-field atoms. As shown in (h), comparing with $3 \times 3 = 9$ atoms for the BDD-based verifier, there are only $3 + 3 = 6$ atoms for the NDD-based verifier, labeled by D_{1-3} and E_{1-3}. After that, as shown in (j), the verifier atomizes each port predicate of NDDs in (g) by atomized NDDs in (i). For example, the BDD label B_5 on the edge of C_5 encoding $dstPort = 0*$, is replaced by atoms E_1 and E_2 encoding $dstPort =$ 00 and 01, respectively. Finally, the verifier traverses the network and computes reachability by the operation $\wedge$ and $\vee$ of atomized NDD and the reachable packets are encoded by F_8.

E An example for BDD arrays

To show the limitation of simply assembling per-field BDDs as arrays, let us take a look at an example in Figure 23 (a), where we use BDD arrays to represent packet sets that are reachable from A to D in the network in Figure 3. Specifically, each array has four BDDs representing four fields, i.e., `src IP/Port` and `dst IP/Port`. Initially, an array of {****, **, **, **} which represents all packets is injected at device A. Then, each device filters the arrays based on its forwarding or ACL rules, and sends the resultant arrays to the next hops. At device D, we end up with 9 arrays which collectively represent all reachable packets from A to D. These 9 arrays correspond to a product of 3×3 paths from the root nodes of the three BDDs in Figure 4 (a) to the BDD terminal node `true` (3 sub-paths for $srcIP/Port$ and 3 sub-paths for $dstIP/Port$). 9 arrays are not a big issue in the example, but what we want to highlight here is the number of arrays can grow quickly due to a cross-product effect. In real scenarios, there can be over $O(10^4)$ arrays even when the network is very small (see Appendix A.10 for details). Figure 23 (b) shows the NDD representing reachable packets from A to D on the right. NDD compactly represents compositions of sub-paths from different fields by a form of decision diagram. Specifically, the 3 sub-paths for $dst\ IP/Port$ are represented once by an NDD node of the $dstIP$ field, and the 3 sub-paths for $src\ IP/Port$ share such NDD node without representing the 3 sub-paths for $dst\ IP/Port$ by 3 times, and therefore occupies much less memory than BDD arrays in practice.

F Use cases of NDD in network verification

F.1 Automatic atomization

We use AP Verifier [51], a data plane verifier, as an example to show how to use the NDD APIs to automatically compute atoms.

Step 1. Computing port predicates. Using NDD, we first use the `createVar` function to declare the header fields (i.e.,

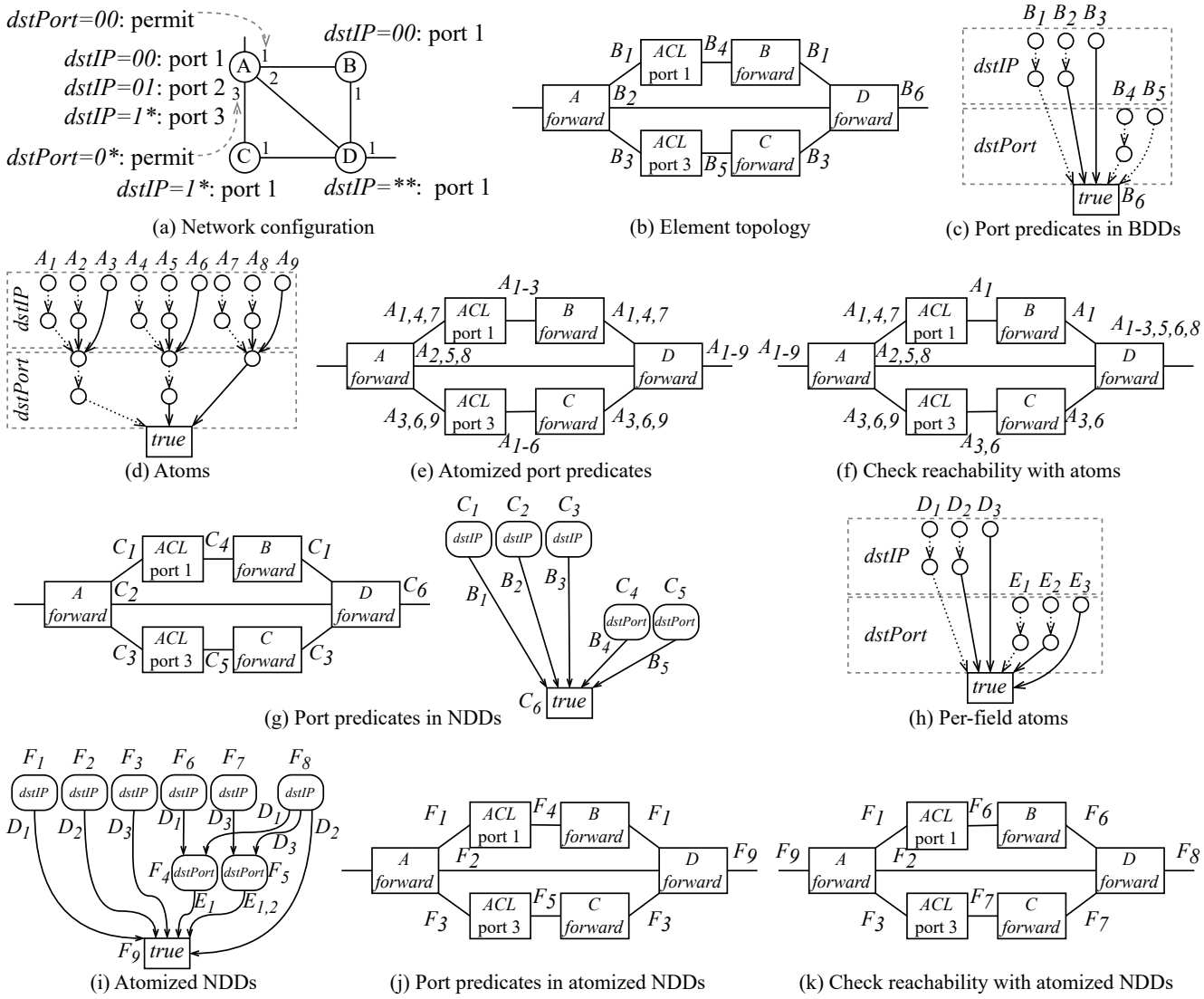


Figure 22: An example for explosion of atoms when using BDDs.

5-tuple). Then, we use the same algorithms as in AP Verifier to compute port predicates, with the only difference that BDD operations are replaced with NDDs operations (`apply`).

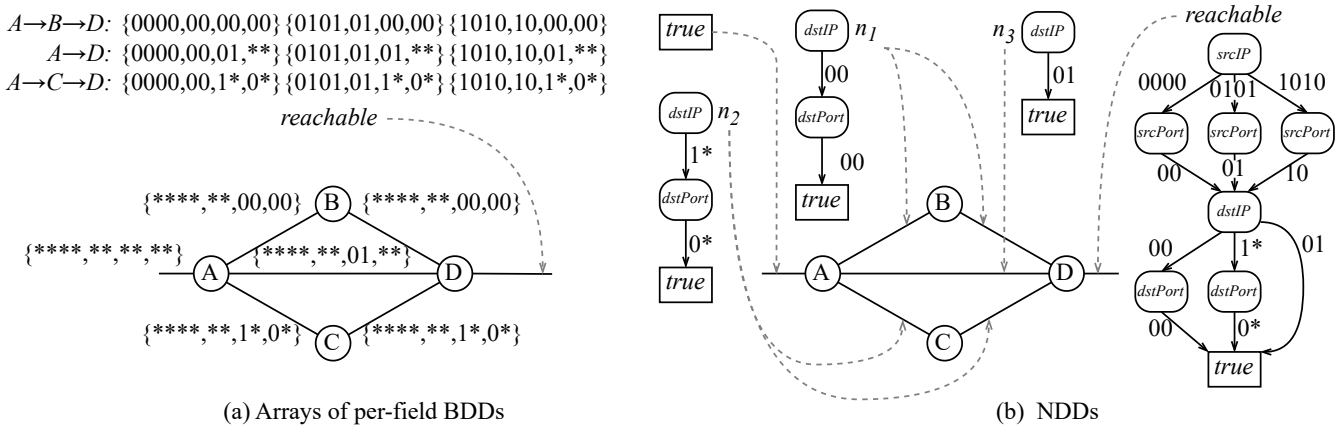
Step 2. Computing atomic predicates. Instead of computing atoms using the port predicates with a custom algorithm (a two-fold loop implemented in AP Verifier), we use the `atomize` function with the port predicates as input. The function will compute atoms for each field, and return the *atomized NDDs*, a new NDD with each BDD on an edge replaced by the equivalent sets of atoms.

Step 3. Checking reachability. We follow the same approach in AP Verifier to check network reachability, which maintains a set of atoms during the traversal, and performs intersections on the set with those of the ports being traversed. The difference is that the traversal uses logical operations on atomized NDD (e.g., `and`, `or`), instead of set operations, as in AP Verifier. In this sense, the process is similar to using BDDs without computing atoms, but can still enjoy a speedup due to computing per-field atoms. Note here the logical operations on atomized NDDs are different from those of standard NDDs: since labels of edges are sets of atoms in atomized NDD, the operations on labels become set operations instead of BDD operations.

F.2 Incremental update of atoms

We use APKeep [57], an incremental data plane verifier, as an example to show how to use the NDD APIs to incrementally update the atoms.

Step 1. Identifying changes. This step takes rule insertions or deletions as input, and calculates changes of forwarding behaviors. A change is represented as a tuple $(from, to, \delta)$, meaning that packets with headers in δ are forwarded to port $from$ before the update and to port to after the update. This



(a) Arrays of per-field BDDs (b) NDDs

Figure 23: Computing reachable packets from A to D in the example in Figure 3, (a) and (b) use arrays of per-field BDDs and NDDs, respectively. Arrays/NDD on each link represent the reachable packets at that position, each element of an array represents a per-field BDD with an ordering of $\{srcIP, srcPort.dstIP, dstPort\}$.

step is identical to that of APKeep.

Step 2. Updating atoms. This step takes changes from step (1) and updates the set of atoms. For each change in the form of $(from, to, \delta)$, it first gets the atomized port predicate a of port $from$ and invokes `update`(δ, a) to update atoms.

Step 3. Transferring atoms. After step (2), δ can be atomized. This step atomizes δ and transfers it from port $from$ to port to using the `diff` and `or` operations for atomized NDD.

F.3 Handling packet transformers

We show how to use the NDD APIs to efficiently handle packet transformers, which refer to rules that *rewrite*, *encapsulate*, or *decapsulate* packets. A packet transformer can be represented as a tuple $(match, field, action)$, where *match* is the matching condition, *field* is the field to be transformed, and *action* is the value to rewrite/encapsulate.

Step 1. Computing port predicates. For each action of transformers, e.g., encapsulating a new IP header with destination IP 10.0.0.1/32 for packets from source IP 192.0.1.0/24, the verifier creates a logical port on the device of the transformer. Each logical port P_i has two predicates: $P_i.match$, an NDD encoding the match conditions, e.g., `innerSrcIP`= 192.0.1.0/24, and $P_i.action$, an NDD encoding the actions, e.g., `outerDstIP`= 10.0.0.1/32. All these predicates are computed as in AP Verifier [51].

Step 2. Atomizing predicates. In this step, the verifier atomizes the predicates computed by step 1, using the NDD API `atomize`, in the same way as §F.1. The differences is that both the $P_i.match$ and $P_i.action$ are used for computing the atoms.

Step 3. Checking properties. In this step, when the packet set *pkt* (an atomized NDD) reaches a transformer, the verifier enumerates each logical port of the transformer: for each port P_i, it computes the conjunction of *pkt* and $P_i.match$, an atomized NDD encoding the match conditions. This is the same as §F.1. If the conjunction is not `false`, and the type of the transformer is `decapsulate` or `rewrite`, the verifier erases the original value of *pkt* on the field $P_i.field$, e.g., `innerSrcIP`, using the `exist` API of NDD. If the type is `encapsulate` or `rewrite`, the verifier sets a new value of *pkt* on the field $P_i.field$, e.g., `outerDstIP`, by computing the conjunction of *pkt* and $P_i.action$. The output is denoted as pkt_i. Finally, the verifier merges the output pkt_i of each logical port P_i into a packet set pkt', and forwards it to the next hop.

G The usage of BDD in network verification

Table 8 shows an incomplete list of network verifiers based on BDDs.

Table 8: Network verifiers based on BDD. ✓means the verifier has already been re-implemented with our NDD library.

	Verifier	BDD library	Symbolic state	Impl.
Data plane	AP Verifier [51]	JDD	packets	✓
	APT [53]	JDD	packets	✓
	APKeep [57]	JDD	packets	✓
	Katra [13]	DD [1]	packets	
	PPV [55]	JavaBDD	packets	
	MNV [36]	JDD	packets	
	ConfigChecker [10]	Buddy	packets	
Control plane	Batfish [19, 25]	JavaBDD	packets	✓
	SRE [58]	JDD	failures, packets	✓
	Expresso [50]	JDD	route adv., packets	
	NV [27]	CUDD	routes	
	ProbNV [28]	CUDD	routes	
	ShapeShifter [16]	unknown	routes	
	DNA [56]	JDD	packets	
	ERA [24]	JDD	routes	
	Bonsai [15]	JavaBDD	packet/route filters	
	Campion [47]	JavaBDD	packet/route filters	

Smart Casual Verification of the Confidential Consortium Framework

Heidi Howard[*], *Markus A. Kuppe*[*], *Edward Ashton*[*], *Amaury Chamayou*[*], *Natacha Crooks*[*†]
Azure Research, Microsoft[*] *UC Berkeley*[†]

Abstract

The Confidential Consortium Framework (CCF) is an open-source platform for developing trustworthy and reliable cloud applications. CCF powers Microsoft's Azure Confidential Ledger service and as such it is vital to build confidence in the correctness of CCF's design and implementation. This paper reports our experiences applying *smart casual verification* to validate the correctness of CCF's novel distributed protocols, focusing on its unique distributed consensus protocol and its custom client consistency model. We use the term smart casual verification to describe our hybrid approach, which combines the rigor of formal specification and model checking with the pragmatism of automated testing, in our case binding the formal specification in TLA$^+$ to the C++ implementation. While traditional formal methods approaches require substantial buy-in and are often one-off efforts by domain experts, we have integrated our smart casual verification approach into CCF's CI pipeline, allowing contributors to continuously validate CCF as it evolves. We describe the challenges we faced in applying smart casual verification to a complex existing codebase and how we overcame them to find six subtle bugs in the design and implementation before they could impact production.

1 Introduction

The Confidential Consortium Framework (CCF) [28, 61] is a general-purpose platform for developing trustworthy and highly available cloud applications. CCF combines centralized compute with decentralized trust, supporting deployment on untrusted cloud infrastructure and transparent governance by mutually untrusted parties. CCF achieves this by leveraging hardware-based trusted execution environments (TEEs) for remotely verifiable confidentiality and code integrity [77, 78], coupled with state machine replication backed by an auditable immutable ledger for data integrity and high availability. CCF is trusted in production by services such as Azure Confidential Ledger [57], a tamper-proof append-only ledger, which is utilized for storing critical data like digests for integrity-protection with SQL Server [3, 63] and Azure Immutable Blob Storage [35]. Open source applications of CCF range from code transparency [17, 58] and mediation of multi-party data sharing [43], to decentralized identity [59], privacy-preserving ad auctions [62], and confidential storage [32, 56].

This paper summarizes our experience with verifying the correctness guarantees of the distributed protocols in CCF, a large-scale, production distributed system, using TLA$^+$ [45, 47]. We have five primary requirements that guided our approach to verification:

(1) Verify high-level distributed safety properties. We aim to check the correctness of our custom distributed consensus protocol, including its dynamic reconfiguration logic. CCF's consensus logic, though based on Raft [74] has been sufficiently modified such that it is now based on an unproven algorithm. This is a common problem: the Chubby authors [8] similarly note that the additional requirements imposed by a real-world system over a vanilla Paxos [46] implementation require significant changes. Moreover, distributed consensus protocols are notoriously hard to get right, and even well-known protocols, including PBFT [9], Egalitarian Paxos [66], and Zyzzyva [39], as well as the previously mentioned Raft and Chubby, have been found to contain subtle bugs [1, 2, 4, 55, 72, 82, 90]. The chosen verification strategy must therefore be able to check properties across distributed nodes, handling asynchrony, concurrency, and non-determinism in the order that events, such as timeouts and message delivery, occur as well as expected failures, like nodes crashing and message loss.

(2) Document and communicate the behavior of the system. A formal specification can act as compact and unambiguous documentation of system behavior for potential users. This gives developers and users clarity about the guarantees they can expect and rely on. The existence of such a spec also offers a succinct mechanism to communicate changes to these guarantees. Due to resource constraints on early TEEs [14], the consistency guarantees offered by CCF to clients can be subtle, confusing users and paper reviewers alike. In fact, even when consistency guarantees are seemingly simple, they

can still be famously difficult to reason about [36]. We thus wish to formally define these guarantees, communicate them clearly, and check that these hold, even as failures occur.

(3) Increased confidence in the implementation & design. While many verification efforts focus solely on checking the design via reference specs, we also wish to validate that our concrete implementation corresponds to our specs. If, in the future, there were to be variants or even multiple implementations of CCF, we would like to verify that they are functionally equivalent in key areas.

(4) Integrates with the existing codebase. We do not wish, however, to rewrite CCF for the purpose of verification. At the time of writing, the implementation is already 63 kLoC in C++. Our support for Intel SGX [14] (via OpenEnclave SDK [19]) constrains us to C++. Moreover, since the project started seven years ago, we have invested significantly in adding functionality, improving performance, and supporting new hardware (AMD SEV-SNP [31]). We want our verification efforts to *improve* upon that existing investment, rather than impose a fresh implementation.

(5) Pragmatic and evolves with the implementation over time. CCF is an ongoing project which is continuously growing and evolving, with an average 16 pull requests merged every week. Since 1.0, the first release to be deployed to production, there have been four further major versions, with minor versions and patches released every 11 days on average. Any verification effort must thus be continuous, automatic, and sufficiently lightweight to integrate into the existing CI pipelines and software engineering workloads. The approach should also be approachable and pragmatic such that anyone contributing to the project can update the specs and debug discrepancies between the implementation and specs.

1.1 Approach

Full formal verification of distributed systems has been successfully applied in various research projects [21, 23, 24, 91], for instance, by synthesizing an executable implementation from a formally proven spec. Unfortunately, while formal verification is a powerful tool, it requires a significant upfront investment in time and expertise. We thus chose a different approach. We already used traditional testing techniques (*casual verification*), but wanted to augment them with an approach that was more rigorous and complete. We thus chose to adopt *smart casual verification*[1], a pragmatic yet systematic approach to verification that combines the rigor of formal methods with the easy-of-use and flexibility of more casual methods. We combine a rigorous TLA$^+$ specification which we *tie* to our existing production implementation using *trace validation*.

We chose TLA$^+$ for several reasons. TLA$^+$ has been successfully utilized to verify the design of a number of production distributed systems [7, 68, 81, 85, 92]. Most notably for us, TLA$^+$ has been used to describe both consistency guarantees [22, 26, 86], and distributed consensus protocols, namely Paxos [46, 87] and Raft [73]. The existence of the last of these was a significant factor in our decision, as it allowed us to start from a complete spec of Raft and adapt it to our protocol's specificities, rather than from scratch. Finally, the maturity of TLA$^+$ and its tools [15, 37, 42, 93], the extensive examples available [51], and its active community [12] assured us of ongoing support and resources. Our increased investment in TLA$^+$, following initial successes, was supported by the availability of a recurring, two-day TLA$^+$ workshop [40] to train our team.

However, achieving our goals is not simply a matter of writing some high-level specs as this would not provide any guarantees about the production code itself. We bridged this gap by applying trace validation to validate implementation traces against the formal specs. In this paper, we describe our experience applying smart casual verification to CCF using TLA$^+$, focusing on the challenges we faced and how we overcame them. First, we present our TLA$^+$ specs of CCF from the perspective of its nodes (§4) and its clients (§5). Next, we present how we validate traces generated from the CCF implementation against our TLA$^+$ specs (§6). Finally, we reflect on our experiences applying smart casual verification to CCF (§7), documenting the six bugs we prevented, and the lessons learned along the way (§8). Both our specifications and implementation are open source and actively maintained [64].

2 CCF

This section provides an overview of the distributed architecture of CCF, focusing on the components that are relevant to our verification efforts. Interested readers can find a more comprehensive description of CCF in [28].

At its core, CCF uses state-machine-replication (SMR) [79] and trusted execution to offer the abstraction of an always available application that remains robust to attacks, including from the nodes themselves. CCF assumes that neither node operators nor other applications running on the hardware can be trusted. The host, the OS, the hypervisor, the network and persistent storage are all assumed to be corruptible. Intuitively, SMR provides clients with the illusion that there is a single server that will, sequentially, execute individual application requests. SMR achieves this by replicating application logic on a set of nodes, a fraction of which may fail. The system maintains consistency across nodes by deciding on a totally ordered transaction log. Formally, SMR guarantees the following (def. from [74, Fig. 3]):

Property 1 (State Machine Safety) *If a node has applied a log entry at a given index to its state machine, no other node will ever apply a different log entry for the same index.*

Most SMR systems further ensure that the agreed-upon set of transactions will be applied in a way that guarantees *linearizability* [25] (or *strict serializability*): the resulting execution

[1] Smart casual refers to a style of dress that is neat and stylish, without the expense and discomfort associated with formal attire such as a business suit. Smart casual dress can easily be made more/less formal, for instance, with the addition/removal of a blazer or tie. [18]

will be equivalent (equal read and write sets) to an execution in which each transaction was executed in sequence, and in an order that matches the order in which they were issued. Though appealing, strict serializability can be costly to enforce. SMR systems thus often relax this guarantee to read-only transactions specifically. They offer only *serializability* and allow read-only transactions to read stale state. CCF is no different: it offers strict serializability for committed read-write transactions and serializability for committed read-only transactions.

Serializability is a gold standard in system design, but is fairly pessimistic: a client must wait until a transaction has committed to learn any information about that transaction and its effects. This design was unfortunately at odds with CCF's initial SGX-related design constraint, which precluded keeping potentially large amounts of application-defined responses in the limited amount of in-enclave memory (128 MB). CCF thus provides configuration settings that clients can use to achieve good performance. They are useful in practice, but make it more challenging to formalize the consistency guarantees that users can expect.

In CCF, the leader node executes transactions as soon as they are received, and prior to them being replicated to other nodes. The leader then directly replies to the client with the result of the transaction without waiting for confirmation that the request has been replicated. As a consequence, a leader failure can cause the transaction to fail, even after a response has been returned to the client. Clients can then, on a per-response basis, decide whether they wish to wait for the transaction to be committed before proceeding or not. Either way, this cuts down the number of open client connections and pending responses, reducing memory footprint significantly.

Transactions in CCF can be in one of the following client-observable states: COMMITTED, PENDING, or INVALID. A transaction is COMMITTED if it has been replicated by the leader to a majority of nodes in its current term. A transaction is PENDING if it has been executed but not yet replicated. If the leader fails before replication is complete, CCF will mark the transaction as INVALID.

PENDING transactions, can eventually become INVALID, but cannot return arbitrary results. They provide a guarantee akin to *fork-linearizability* [52] (or *fork sequential consistency* [6] when considering read-only transactions). A pending transaction observes a prefix of committed transactions and a sequence of pending transactions. Leader failures may cause the system to fork and generate multiple (locally linearizable) sequences of pending transactions. Only one forked sequence will eventually commit, thus ensuring that the set of committed transactions remains linearizable. All other sequences will be marked as invalid. In other words, if a pending transaction commits, the result it returned to clients is guaranteed to have been linearizable.

CCF makes extensive use of timestamps to help clients understand when transactions transition from PENDING to COMMITTED or INVALID. Each transaction is associated with a unique transaction identifier, consisting of a lexicographically ordered pair $\langle t.i \rangle$ of term t and log index i. The client can use this ID to quickly understand the system state. CCF, for instance, enforces *timestamp ordering*: if $txid < txid'$ and the two transactions are committed, then the transaction with $txid$ executed before $txid'$. Clients can further use this ID to learn the state of not only this transaction, but its ancestors. For instance, CCF guarantees that:

Property 2 (Ancestor Commit) *If $\langle t.i \rangle$ is committed then any transaction $\langle t.j \rangle$ where $j \leq i$ is also committed.*

Ensuring that CCF does indeed provide these specific guarantees requires care and adapting existing linearizability specs, which only reason about committed operations. They support neither reasoning about forks nor timestamp properties.

To tolerate node crashes and network asynchrony, SMR requires a crash fault-tolerant distributed consensus protocol or equivalent [10], such as Multi-Paxos [46] or Raft [74], to agree on a total-ordered transaction log. While the terminology varies, such protocols are typically leader-based and operate by electing one of the nodes to be the *leader* while the other nodes are *followers*. The leader is responsible for proposing new transactions for the log and the followers are responsible for replicating them. The leader will only consider a transaction to be committed once a strict majority of nodes have replicated the transaction at the same position in the log. When the leader fails, a new leader is elected from the remaining nodes and the protocol continues. We use *terms* to distinguish between the different periods of leadership and there should be at most one leader per term.

It is the responsibility of the leader-based consensus protocol to ensure that the new leader has knowledge of all previously committed transactions. In Raft, this is achieved by requiring followers to become *candidates* (transition ① in Fig. 1) before they can become leaders. A candidate will only become a leader (transition ② in Fig. 1) if it can obtain a strict majority of votes from the other nodes. A node will only vote for a candidate if it has not already voted in the candidate's term and the candidate's log is at least as up-to-date as its own log. The former ensures that at most one leader can be elected per term and the latter ensures that the new leader has knowledge of all committed transactions from previous terms. A more comprehensive description of Raft can be found elsewhere [74].

2.1 What makes CCF's distributed consensus protocol interesting?

CCF uses a custom consensus protocol, which evolved from Raft. Now we will outline some of the ways that the consensus protocol in CCF today differs from the description of the Raft protocol given the original paper. While some of these modifications may seem small, the combined effect is significant and the interactions between them have proven complex and subtle, leading to the bugs we later describe (§7).

Signature transactions Offline log integrity and transaction provenance are key requirements for CCF, neither of which is provided by Raft, nor by the AEAD mechanism used inside the CCF network. The offline guarantees crucially enable external audit, and disaster recovery. To implement them efficiently, CCF utilizes *signature transactions*, which include the root of a Merkle tree [54] over the whole log thus far, signed by the current leader. A transaction in the log is not considered committed unless a subsequent signature has been committed.

Messaging not RPCs CCF does not use RPCs to communicate between nodes, and instead it uses a uni-directional messaging layer. When a node receives a response to a message it has sent, it does not know which message the response corresponds to as we do not assume reliable or in-order delivery. Raft uses two main message types: AppendEntries (AE) and RequestVote (RV). AE messages are used to replicate transactions and RV messages are used to elect a leader. In the case of RVs, the term in the response is sufficient to handle the reply. In the case of AEs, the response in CCF contains an additional field, LastIndex. For positive responses to AE messages (AE-ACK), this is the index of the last transaction in the follower's log.

Optimistic acknowledgement In Raft, the leader replicates transactions to its followers using AE requests. The leader maintains a NextIndex for each node to record which log entry to send next to the follower. This index is updated when a follower responds positively to an AE request (AE-ACK). If a leader receives a new transaction from a client, before it has received a response from a follower, it must either (1) wait for the reply for the previous transaction, potentially impacting liveness and performance as AEs cannot be pipelined, or (2) send both transactions in the next AE request, even though the follower is likely to already have the first transaction. CCF avoids this problem by allowing the leader to update the index, known in CCF as the SentIndex, as soon the AE message is sent. This therefore means that if the leader receives a negative response to its AE request (AE-NACK), it might need to rollback the SentIndex.

Express node catch up AE messages include a previous log index and term, which allows a follower to determine if it diverged from the leader. If a follower does not have the previous log index and term, either because it does not have, or has a different transaction at that index, it then responds with an AE-NACK. CCF uses an express catch up mechanism, where a leader makes a conservative estimate of how far behind a follower is, and sends a batch of transactions to the follower.

Partition leader step down A known limitation of the base Raft protocol is that partial/asymmetric network partitions can cause a loss of liveness [27, 33]. For instance, if a leader can no longer make progress because it cannot receive messages from the other nodes, it continues to send AE heartbeats to followers, preventing them from timing out and from electing a new leader who can make progress.

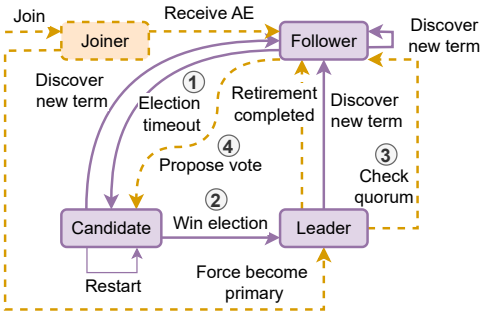


Figure 1: State transitions for CCF's consensus protocol. Solid boxes and lines show Raft's original states and transitions, CCF's additional states and transitions are shown with dashed lines.

CCF implements a known extension [71, pg. 69] to the Raft protocol, referred to as *CheckQuorum*, where a leader steps down if it does not hear back from a quorum of nodes within a specified time period (transition ③ in Fig. 1).

Bootstrapping to retirement Reconfiguration is the process by which the set of nodes participating in consensus can be changed. Raft support two reconfiguration protocols: *joint consensus*, described in the original paper [74], and single-server reconfiguration, subsequently described in [71]. CCF's reconfiguration protocol is more similar to the former. Reconfigurations are recorded in CCF's log with *configuration transactions*, and are therefore ordered in the same total order. Logs always begin with an initial singleton configuration transaction followed by a signature transaction. To change the configuration, the leader proposes a new configuration transaction specifying the new set of nodes, which may be different in cardinality to the current set of nodes and may or may not be disjoint. To commit this transaction, the leader must obtain a quorum of AE-ACKs from both the previous and new configurations. Once the transaction is committed, the leader no longer needs quorum agreement from the previous configuration. If a node has been removed from the configuration, we refer to it as *retiring*. In order to complete its retirement and permanently switch off the node, a *retirement transaction* must be committed to ensure that any future leader will know that the reconfiguration which removed the node has been committed, and thus the node will never be needed. CCF also adds a message to the protocol, *ProposeVote* which is utilized by a retiring leader to nominate a successor, fast-tracking the usual leader election process (transition ④ in Fig. 1).

3 Primer on TLA$^+$

TLA$^+$ [45, 47] is a formal modeling language widely used to verify concurrent and distributed systems. It is easy to learn and use, as well as agnostic about system frameworks or implementation languages.

TLA$^+$ is a variant of linear temporal time logic with only two operators, *Always* (□) and *Eventually* (◇). A system is defined by a set of *behaviors*, each a sequence of pairs of states called *actions*, beginning at one of the system's initial states. A *state* is an assignment of values to variables.

Syntactically, TLA$^+$ describes a system's state machine using a canonical (temporal) formula $Init \wedge \Box[Next]_{vars} \wedge L$. Here, $Init$ is a predicate defining the system's set of initial states. The system's next-state relation $Next$ is a first-order logic formula that is usually decomposed into a disjunct of actions which relates the values of the variables in the current state to the ones in the successor state. CHECKQUORUM (Listing 3), for instance, states that a node i can abdicate as a leader and become a follower; we change the value of the variable *role* to $Follower$ in the successor state, while the values of the other variables remain unchanged.[2] The tuple, $vars$, represents the spec's variables, and $[Next]_{vars}$ stipulates that either $Next$ is true, or the variable in $vars$ do not change. This asserts that TLA$^+$ specs are stuttering-insensitive, allowing a spec to always be refined by a more detailed, low-level one. The optional formula L is used to assert fairness, i.e., constraints on the system's behavior that ensure that certain actions eventually occur. Composition of actions allows us to change the *grain of atomicity* by defining more coarse-grained behaviors [47, §7.3]. Concretely, the composition $A \cdot B$ states that the two actions happen atomically; the intermediate state between A and B is not observable.

We also state desired safety (*something bad never happens*) and liveness (*something good eventually happens*) properties in TLA$^+$. For example, the safety properties LOGINV, APPENDONLYPROP, and MONOLOGINV (Listing 3, described in §4) assert properties of the *log*. Properties are checked and verified using, random state space exploration (simulation), model checking, or theorem proving. TLC [93], an explicit-state model checker, verifies that a *finite* model of a spec satisfies its properties by enumerating all reachable states. A symbolic model-checker [37] is especially well-suited for the verification of inductive invariants of finite systems. The TLA$^+$ Proof System [15, 88] mechanically verifies a deductive proof of the properties of an infinite system. These tools complement each other, enabling a combination of model-checking and deductive proofs to verify specs. This approach allows for varying levels of verification depth, from push-button model checking to fully mechanized safety and liveness proofs [38]. Our companion paper [11, §2.1] provides a more detailed summary of TLA$^+$.

4 Distributed Consensus Specification

Fig. 2 summarizes our consensus verification architecture. The consensus specification, shown as ① in Fig. 2, consists of 17 actions to describe the transitions over 13 variables. The first 12 variables are local and track consensus state (CURRENTTERM, LOG, COMMITINDEX, ...). The last variable instead tracks the set of in-transit messages, allowing support for different network abstractions (ordered/unordered delivery, etc.) and is shown as ② in Fig. 2. Each action models a node taking a single step within the protocol, updating its

[2]The value of the variable *role* is a mapping from all node identifiers to their roles such as *leader*, *follower*, and *candidate*.

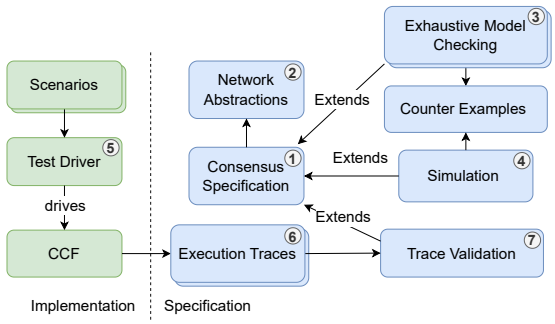


Figure 2: Components of our verification architecture for consensus

$$
\begin{aligned}
&CheckQuorum(i) \triangleq \\
&\quad \wedge role[i] = Leader \\
&\quad \wedge role' = [role \text{ EXCEPT } ![i] = Follower] \\
&\quad \wedge \text{UNCHANGED } \langle currentTerm, log, votedFor, ... \rangle
\end{aligned}
$$

$$
\begin{aligned}
&LogInv \triangleq \forall i, j \in Nodes: \\
&\quad \vee IsPrefix(Committed(i), Committed(j)) \\
&\quad \vee IsPrefix(Committed(j), Committed(i))
\end{aligned}
$$

$$
\begin{aligned}
&AppendOnlyProp \triangleq \\
&\quad \Box[\forall i \in Nodes: \\
&\quad\ IsPrefix(Committed(i), Committed(i)')]_{vars}
\end{aligned}
$$

$$
\begin{aligned}
&MonoLogInv \triangleq \forall i \in Nodes: log[i] \neq \langle\rangle \Rightarrow \\
&\quad \forall k \in 1 .. Len(log[i]) - 1: \\
&\quad\ \vee log[i][k].term = log[i][k+1].term \\
&\quad\ \vee \ \wedge log[i][k].term < log[i][k+1].term \\
&\quad\quad\ \wedge log[i][k].contentType = Signature
\end{aligned}
$$

Listing 3: Excerpt of our consensus spec.

local state and optionally the collection of in-transit messages, (i.e. CHECKQUORUM in Listing 3). Actions can either be initiated from a message receipt or by a node itself. Actions in the latter class include adding a signature transaction, stepping up as a candidate or stepping down as a leader. Such actions are always enabled, to model the fact that we make no assumptions about clock synchrony; each node's opinion of the progression of time is independent. Later we discuss how action-weighted simulation enabled us to find bugs in the prototype despite the state space explosion inherent with such an approach. The spec is parameterized by the set of nodes available in the service. The initial states of the spec include every non-empty subset of nodes in the initial configuration with any node in that initial configuration as an initial leader.

Our key correctness property is State Machine Safety (Property 1) which we check with invariant LOGINV and action property APPENDONLYPROP (Listing 3). LOGINV states that all pairs of committed logs must be consistent and APPENDONLYPROP states that each node can only extend its committed log. LOGINV checks for safety violations across nodes (in space) and APPENDONLYPROP checks for safety violations within a node (in time). We also checked a further 27 invariants/properties to ensure the correctness of our consensus protocol and of our understanding (see [64]). One such example is MONOLOGINV (Listing 3), which is a stronger

variant of the property that log terms are monotonically increasing which we depend upon extensively. More concretely, MONOLOGINV states that terms in the log can only increase after a signature and remains the same otherwise.

Our consensus spec is unbounded, there is always the possibility of a new transaction being proposed or a timeout triggering a new election. To exhaustively model check our spec, we extend our consensus spec, shown as ③ in Fig. 2, to restrict the state space by adding additional constraints to the actions limiting the max term, number of client requests, and the sequence of reconfigurations. We found exhaustive model checking too time-consuming to run with all but the strongest state constraints in our CI pipeline, particularly after the spec was updated to describe reconfiguration accurately (discussed further in §8). We therefore developed an extension to our spec for simulation as a lightweight alternative to exhaustive state exploration. Our simulation spec, shown as ④ in Fig. 2, takes a time quota and explores as many behaviors as possible, up to a given depth, within that time. To expand the coverage of simulation, specifically to explore behaviors where the system exhibits more forward progress, we manually weighted failure actions to reduce the likelihood of them being chosen. We also implemented Q-Learning proposed by [67] in TLC to automatically weight actions to increase coverage of simulation. However, we were unable to find the right set of variables as input to Q-Learning's state hash function $\mathcal{H}$ that achieved better coverage than manual weighting.

5 Client Consistency Specification

Following our success with the consensus spec and a discussion regarding the linearizability of read-only transactions [60], we decided to formalize the possible externally visible behaviors of a CCF service in TLA$^+$ to better understand the guarantees provided. Our aim with this spec was to keep it as high-level as possible, focusing solely on the possible safe interactions between clients and the service. By design this spec does not model the internal details of the service itself such as the state of individual nodes or the messages exchanged between nodes as such low level details are already modeled by our consensus spec.

Our consistency spec uses just two variables. The first is HISTORY, an append-only sequence which records the messages exchanges between clients and the services. Five messages are supported, read-only/read-write transaction requests and responses as well as a transaction status messages. Note that since we are focused on safety, we omit messages that cannot impact correctness but can increase the state space, for instance, we do not track when a client requested a transaction status, nor do we track status responses of PENDING. In order to stress the consistency guarantees, we modelled an application where all transactions operate on a simple value, reading the current value, appending a new identifier to the value and writing back the new value. All transactions conflict and each transaction observes every transaction that has been executed before it. To succinctly record the current state of the service we use LOGBRANCHES, an append-only two-dimensional sequence, where the sequence at index i corresponds with the local log of the leader of term i if such a node still exists. This representation does not therefore need to be parameterized over the number of nodes in the service and usefully models the fact that there can be multiple leaders at one time (although they will have different terms).

$$
\begin{aligned}
&PrevCommittedInv \triangleq \\
&\quad \forall i,j \in \{x \in \text{DOMAIN } history : history[x].type = Status\} : \\
&\quad \wedge history[i].status = Committed \\
&\quad \wedge history[i].term = history[j].term \\
&\quad \wedge history[j].index \leq history[i].index \\
&\quad \Rightarrow history[j].status = Committed
\end{aligned}
$$

$$
\begin{aligned}
&ObservedRoInv \triangleq \\
&\quad \forall i \in RwResCommitIndexes : \\
&\quad\quad \forall j \in RoReqCommitIndexes : \\
&\quad\quad\quad \forall k \in RoResCommitIndexes : \\
&\quad\quad\quad\quad history[k].tx = history[j].tx \wedge i < j \\
&\quad\quad\quad\quad \Rightarrow Contains(history[k].observed, history[i].tx)
\end{aligned}
$$

Listing 4: Two properties over histories checked by our consistency spec.

To populate these histories, we defined a set of actions to describe how the history can be extended depending on the log branches. Initially, the history and log branches are empty. The possible actions are: append a transaction request or response to the history, execute a transaction by appending it to any log branches, append a transaction status to the history, or starting a new log branch to simulate leader election. Note that when a transaction is executed, it can be appended to any log branch, this simulates the fact that a transaction can be handled by any node that believes itself to be the leader, even if it is not the latest leader. At any time, a new log branch can be started. This new log branch can be any prefix of any existing log branches, provided it includes the last committed transaction. Having established an approach to generate histories, we formalized in TLA$^+$ the properties that we expect to hold over these histories and check them with TLC. For instance, PREVCOMMITTEDINV (Listings 4) formalizes Property 2 by stating that for any pair of transaction status responses from the same term, if the one with the greater (or equal) index is COMMITTED, then the other status response must also be COMMITTED (as PENDING states are not modelled). OBSERVEDROINV (Listings 4) states that if a committed read-write transaction received an initial response (event i), before a committed read-only transaction was started (event j) then the read-only transaction response (event k) must observe the read-write transaction. Note that RWRESCOMMITINDEXES is the set of indexes for all read-write transaction response events, filtered to include only transactions that were subsequently committed. ROREQCOMMITINDEXES and RORESCOMMITINDEXES are similarly defined for read-only transaction requests and responses.

6 Trace Validation

Conceptually, trace validation checks that every observed implementation trace matches a behavior of the system's high-level specification. We must therefore first collect implementation traces before validating them against our high-level consensus TLA+ specification. A more formal discussion of trace validation, including insights from applying trace validation to systems other than CCF, is provided in [11].

6.1 Trace Collection

When we began the trace validation work, CCF already had extensive unit, functional, and end-to-end testing. Unit testing of CCF's consensus layer consisted of approximately 1.6 kLoC. End-to-end testing covered complex scenarios such as reconfigurations, node failover, and network partitions, spread over more than 2 kLoC of Python tests and infrastructure code. Consensus functional testing was done through a scenario driver, shown as ⑤ in Fig. 2, that serialized execution deterministically across nodes, and isolated the consensus layer by mocking unrelated CCF components, such as governance, networking etc. This driver allowed the injection of network faults such as partitions, delays, reorderings, and message loss, and provides observability. Core correctness invariants and properties were checked at designated execution steps in 13 manually written scenario tests exercising replication, election, and reconfiguration under controlled fault conditions. Additionally, an initial prototype to fuzz-test the consensus layer through generated inputs and faults was developed but ultimately abandoned since it failed to generate interesting behaviors that would achieve satisfactory coverage.

The driver allows us to replace the node's wall clocks with a single global clock. If this global clock had not been possible, a distributed clock, such as a Lamport or vector clock, would have also provided the necessary event ordering to establish the happen-before relationships. However, a distributed clock would have required changes to the network message format. We enhanced system observability by incorporating an additional 15 log statements to capture consistent system states at well-defined, side-effect-free linearization points including (i) the sending and receipt of network messages, and (ii) transitions in a node's high-level state, such as moving from candidate to leader[3]. It is important to note that the driver logs only those values that remain constant in space. For instance, the driver records the length of the logs but not the log entries themselves, which would become impractically large. Still, the logging is disabled at compile time for production builds, and thus does not impact CCF's performance.

Before validation, implementation traces, shown as ⑥ in Fig. 2, are preprocessed to exclude and de-duplicate events from the initial bootstrapping phase of a CCF network, as this phase is not modeled in our consensus spec.

[3] State changes are logged immediately after acquiring a global lock.

```
VARIABLE ln

IsSendAppendEntries ≜
    Enablement conditions on current state.
  ∧ IsEvent(ln,"sndAE")
  ∧ commitIndex[ln.snd] = ln.commit_idx
  ∧ ...
    High-level spec actions.
  ∧ AppendEntries(ln.snd,ln.rcv)
    Assertions on successor states.
  ∧ ∃m ∈ Network ! Messages' :
    ∧ Network ! OneMoreMessage(m)
    ∧ IsAppendEntriesRequest(m,ln)
  ∧ ...

IsRcvAppendEntries ≜
    Enablement conditions on current state.
  ∧ IsEvent(ln,"recvAE") ∧ ...
    High-level spec actions.
  ∧ ∃m ∈ Network ! MessagesToFrom(ln.rcv,ln.snd) :
    IsAppendEntriesRequest(m,ln) ∧
    ∨ HandleAppendEntriesReq(ln.rcv,ln.snd,m)
      Impl optimization: Piggyback term on AppendEntries.
    ∨ UpdateTerm(ln.rcv,ln.snd,m) ·
        HandleAppendEntriesReq(ln.rcv,ln.snd,m)
    ∨ ...
    Assertions on successor states.
  ∧ ...

IsSendAppendEntriesResponse ≜
    Enablement conditions on current and successor state.
  ∧ IsEvent(ln,"sndAER") ∧ ...
    High-level spec actions.
  ∧ UNCHANGED vars

IsFault ≜ ∃s ∈ MultiPowerset(network) :
    network' = s ∧ UNCHANGED AllVarsExceptNetwork

Spec ≜ Init ∧ □[IsFault · Next]⟨vars,ln⟩
```

Listing 5: Excerpt of the *Trace* spec for trace validation.

6.2 Consensus Trace Validation

Trace validation, shown as ⑦ in Fig. 2, ensures that an implementation trace is consistent with the specification. More formally, it verifies whether the set of behaviors $\mathcal{T}$, which encapsulates the values and events from a trace, intersects with the set of behaviors $\mathcal{S}$ derived from the high-level spec, thereby checking that $\mathcal{T} \cap \mathcal{S} \neq \emptyset$. While TLC can construct $\mathcal{S}$, it cannot directly generate $\mathcal{T}$ from the trace. Thus, we write a new TLA+ spec, *Trace*, reusing many actions of the definitions from the high-level spec. However, the actions are only enabled iff the current event in the trace matches an action. Likewise, the actions are parameterized by the values taken from the trace, effectively constraining successor states.

Consider for example *IsSendAppendEntries* (Listing 5). This action is enabled iff the current line of the trace, denoted by *ln*,

is a *sndAE* event, and the *commitIndex* of the sending node (denoted by *ln.snd*) matches the trace's *commit_idx*. The action then reuses the definition of the *AppendEntries* action from the consensus spec, parameterized by *ln.snd* and *ln.rcv* from the trace. Given that the number of entries in the high-level action *AppendEntries* is chosen nondeterministically within defined limits, *IsSendAppendEntries* determines the successor state by asserting the existence of an additional *AppendEntriesRequest* in the network with a matching number of entries. Note that while the consensus spec modeled the network as a set of messages, thereby leaving the variable upon a resend of an *AppendEntriesRequest* unchanged, *Trace* redefines the value of the variable to be a multi-set. This allowed the spec to account for the addition of messages in the network, even during resend events. Subsequently, this approach to address the impedance mismatch was expanded to verify, with TLC, the impact of various message delivery guarantees, such as ordering, duplication, and other message loss patterns.

6.2.1 Aligning Grains Of Atomicity

The granularity of some actions in the consensus spec did not align with the granularity of events in the traces. The action *IsRcvAppendEntries* outlines the alignment of such different *grains of atomicity*. Like many Raft implementations, CCF minimize network round trips by piggybacking *term* updates on *AppendEntries* messages. This optimization was not reflected in the consensus spec. Instead, the spec modeled term updates with an action that increases the term upon a pending *AppendEntriesRequest*, while leaving the variable unchanged. In effect, a term update might nondeterministically occur before the receipt of the *AppendEntriesRequest*. To reconcile these different grains of atomicity, we composed the actions *UpdateTerm* and *HandleAppendEntriesReq*, allowing them to occur atomically, i.e., in a single action.

Another important application of action composition addresses events that are omitted from the trace, such as losing messages. Although our consensus specs explicitly modeled losing messages, message loss was not recorded in the trace. Therefore, to account for faults at any step of a behavior, we composed an *IsFault* action with *Trace*'s next-state relation. Conversely, aligning a single high-level action with multiple implementation events is addressed by introducing finite stuttering that does not change the high-level variables. For instance, the action *IsSendAppendEntriesResponse* is enabled iff *ln* is a *sndAER* event. However, it leaves the high-level variables unchanged.

6.2.2 Deriving Consensus Trace Validation

We began to derive *Trace* by mapping the trace of a straightforward happy-path test line by line to the consensus spec. To proactively catch discrepancies, we added as many assertions as possible to *Trace*. Whenever we discovered discrepancies, we investigated them by examining relevant sections of the implementation's source code. Adding cross-references between the implementation and the spec proved useful, particularly when the terminology——such as variable and function names——differed between the two. We further debugged the *Trace* using the TLA$^+$ debugger [41] in tandem with implementation debugging. Upon detecting discrepancies, we corrected either the implementation or the spec, subsequently rerunning verification on the revised consensus spec, and executing tests on the updated implementation.

6.3 Debugging Discrepancies

Bogus logging, incorrect mappings from implementation to spec state, or true discrepancies between the spec and the implementation, resulted in $\mathcal{T} \cap \mathcal{S} = \emptyset$, i.e., the verdict that a trace is invalid. In either case, contrary to ordinary model checking, a failure to validate a trace had no counterexample. However, the behaviors within $\mathcal{T}$ helped explain why a trace failed to validate. We typically compared the final state of the longest behaviors and the corresponding line in the trace to identify the source of the mismatch. The TLA$^+$ debugger was instrumental in this process, as it allowed us to step through the evaluation of formulas and compare variables at the current and successor states with the trace values. To determine if *Trace* is overly restrictive, we implemented a new *unsatisfied breakpoint*. It triggers for each state in $\mathcal{T}$ that is found to be unreachable. Furthermore, $\mathcal{T}$ can be visualized as a graph that not only includes all unreachable states but also references the subformula responsible for each state being unreachable.

6.4 Scalability of Trace Validation

The cardinality of $\mathcal{T}$, the set of potential system behaviors, can become prohibitively large due to nondeterminism resulting from incomplete traces. Recognizing that it suffices to find a single behavior in the intersection of $\mathcal{T}$ and $\mathcal{S}$ to check the validity of a trace, we implemented depth-first search (DFS) in TLC. This method mitigated the issue of state-space explosion, making trace validation orders of magnitude faster compared to enumerating all behaviors with breadth-first search (BFS). For instance, validating a trace against our consistency spec started to take less than a second using DFS, compared to about an hour with BFS.

6.5 Trace Validation Effort

The enhancements to the test driver and the addition of detailed logging were completed in approximately one day. The effort to derive a version of the *Trace* spec that validated the majority of the traces required approximately two engineer-months, spread over four months. The primary tasks included enhancing the TLC model checker to support trace validation, which involved implementing support for action composition, DFS, improved debugging support, and visualizing the state graph. The second major tasks was diagnosing if the root cause of discrepancies arose from bugs in the reverse-engineered spec, the implementation, or both. This frequently required consulting the original Raft paper and discussions with the CCF experts to elucidate differences between Raft

and CCF. In this context, the shared vocabulary developed during the TLA+ workshops, along with TLA+ counterexamples from trace validation, simulation, and model checking proved invaluable. The third major task involved finding modeling patterns to bridge impedance mismatches between the high-level TLA+ design and low-level implementation. Writing the 400 LoC *Trace* spec itself was a minor task. The introduction of trace validation resulted in 88 fine-grained commits to the *Trace* spec, while the consensus spec underwent 107 changes.

The discovery of a serious safety bug through trace validation (*Commit advance on AE-NACK* §7) led to increased investment in trace validation. Substantial changes were made to the consensus spec to accurately reflect the implementation. For example, the bootstrapping of a CCF network was modeled with greater fidelity. Additionally, the spec was expanded to include all node states, especially those related to node retirement. These comprehensive changes necessitated substantial revisions to the test driver and the development of new tests. This uncovered a serious liveness bug (*Premature node retirement* §7). Once the consensus spec was validated to accurately mirror the implementation, we transitioned to a spec-driven development, wherein the spec served as the source of truth. Notably, this phase included the integration of the *ProposeVote* messages (described in §4; used for transition ④ in Fig. 1).

Later, trace validation was also applied to the consistency spec, requiring significantly less effort. The consistency spec was considerably less complex, written with the implementation in mind, and we had already gained experience validating our consensus spec. Moreover, TLC had already been enhanced to support trace validation. No instrumentation of the CCF source code was required for consistency trace validation. Instead, the implementation state was observed by making calls to the system's REST API. As with consensus specification, we had to address impedance mismatches. For instance, the consistency spec assumed knowledge of the transactions of all clients, whereas a trace is limited to the transactions of a single client. This required defining a TLA+ action in the specification to reconstruct all transactions based on observed transaction IDs. Yet, applying trace validation to consistency was almost entirely carried out by the CCF experts, with minimal involvement from the formal methods expert. The effort required to apply trace validation to the consistency spec was approximately one engineer-week, spread over a two-week period.[4]

7 Results

This section summarizes the core results of our efforts, focusing on state coverage as well as the bugs found. Table 1 compares the sizes of the TLA+ specs against the implementation and test infra, to give a sense of scale and illustrate the level of detail necessary to execute trace validation. We find that verification of our TLA+ specs is an extremely efficient way to achieve state coverage. Comparing state exploration between implementation and specs is straightforward, because traces can also be collected in end-to-end tests, and one log line is largely equivalent to a spec action. It is immediately apparent that verification of our consensus spec explores orders of magnitude more states at a higher rate than implementation testing. While code size is not a direct measure of cost, the overall size of the spec and its models are not out of proportion compared to the tests. Importantly, the consistency spec required very little infrastructure to verify, and to validate traces against the implementation. The cost of writing formal documentation of the log's consistency guarantee was thus low, and validating it and keeping it in sync with the implementation is equally affordable.

Table 1: Scale of specifications and state coverage.

	Item	LoC	Vars	Approx states /min	Approx states Total
Consensus	Specification	1134	13		
	Model Checking	158		10^6	10^8
	Simulation	69		10^6	10^8
	Trace Validation	369			
	Implementation	2174	25		
	Functional Tests	2579		10^5	10^3
	End-to-end Tests	2815		10^3	10^4
Consistency	Specification	375	2		
	Model Checking	70		10^6	10^5
	Simulation	0		10^5	10^3
	Trace Validation	111			
	Functional Tests	123			

All numbers measured on an Azure DC8s v3 VM.

Table 2 lists the most serious bugs found in our consensus protocol as part of the verification work. These bugs affected both the safety and the liveness of CCF, and were uncovered at several stages of the process by each tool in our verification wardrobe. In the rest of this section we describe the bugs which were found and corrected during this process, explaining how they were uncovered by our combination of smart casual and classical testing methods. All these bugs were fixed before they affected any end-users or resulted in customer bug reports. Thus, although this formalization work took place after the system was initially developed and deployed, it still provided a core benefit of spec-driven development; the identification and removal of critical bugs before they impact production, proving that it is never too late to benefit from smart casual verification.

Incorrect election quorum tally 48 hours of exhaustive model checking of the consensus spec on a 128 core machine revealed that CCF was tallying election quorums against the union of active configurations (the current configuration plus any pending reconfigurations), rather than against each indi-

[4]The work on trace validation has been tracked in milestones 18 and 20 at https://github.com/microsoft/CCF/milestones/.

Table 2: Bugs found in CCF's consensus protocol before they could impact production.

Name	Violation	High-Level Description (Issues)
Incorrect election quorum tally	Safety	Quorum was tallied against union of active configurations, rather than against each individual active configuration. (#3837, #3948, #4018)
Commit advance for previous term	Safety	Leaders could advance commit for historical terms without extending log in the current term. (#3828, #3950, #3971, #5674)
Commit advance on AE-NACK	Safety	Variable reuse could cause the leader's commit index to advance when receiving an AE-NACK (#5324, #5325)
Truncation from early AE	Safety	Followers could roll back committed entries, after a sequence triggered by stale AE-NACK messages. (#5927, #5991, #6016)
Inaccurate AE-ACK	Safety	AE-ACK could report an index beyond the end index of AE received, despite the suffix potentially being incompatible. (#6001, #6016)
Premature node retirement	Liveness	Nodes could stop participating in consensus too early during retirement, leading to diminished fault tolerance. (#5919, #5973)

vidual active configuration (compare §2.1). The initial implementation had correctly used the majority in each term, as described in [74, §5.2], but had not been updated appropriately when reconfiguration was implemented. This meant that a node could be elected leader in a term without having a quorum in one of the active configurations, potentially allowing two leaders being elected in the same term, violating a core safety property. The issue was reproduced with functional and end-to-end testing, a fix was applied, and resolved the problem in both tests and model checking. This was the first bug identified by TLC and motivated our further investment in TLA$^+$.

Commit advance for previous term While assessing the work required to align the implementation and the spec, we discovered that the implementation omitted a check described in Raft. Our implementation allowed a leader to advance its commit index based solely on receiving a quorum of AE-ACKs confirming a given log entry, and missed the additional requirement that this entry must have been appended by the current leader. This restriction is fully explained in [74, §5.4.2], along with the corresponding risk to safety, but had been accidentally omitted in the implementation. We added a scenario test based on [74, Fig. 9] to confirm that the implementation was faulty. An initial fix emptied the node's set of indices eligible for commit (because they are signature transactions) when becoming a leader. This fix passed all existing and amended tests and was thus integrated into the codebase. Our work on trace validation, several months later, required us to revise the spec to represent committable indices accurately. Subsequent simulation revealed a safety violation caused by the initial fix; the *fix* broke an implicit property that committable indices contains *all* signatures. A second fix was implemented and tested as well as verified with TLC. This bug illustrated that even deterministic testing (compare §6.1) is insufficient to guarantee the correctness of changes. Moreover, it confirmed that trace validation is effective at guiding the alignment of the spec and the implementation to enable verification.

Commit advance on AE-NACK Trace validation discovered that the spec defined a leader's *matchIndex* to remain unchanged after receiving a follower's AE-NACK, whereas the implementation allowed it to decrease. This difference was due to an aggressive implementation of an optimization proposed in [74, §5.3, last paragraph]. After a single LoC change to align the spec with the implementation, subsequent simulation found a 34-state counterexample violating one of the spec's main correctness properties. This counterexample was manually translated into a 150 LoC functional test, confirming that the implementation could incorrectly advance its commit index. We also noted that [74, fig. 2, p. 4] implicitly states that *matchIndex* should never decrease, except after a leader election. Adding this property to the spec allowed model checking to find a shorter counterexample. The combination of functional testing and model checking allowed us to fix this bug quickly and confidently.

Truncation from early AE Once a subset of our initial scenarios passed trace validation, investigating why the remaining scenarios failed trace validation uncovered a safety violation. Log entries necessary to the persistence of committed transactions could be rolled back by a follower. This bug was introduced by an optimization, and existed in the implementation for some time. When the suffix of a follower's log is incompatible with that of the leader, there is a need to find the last agreement point. The Raft paper describes an iterative reverse search of the sequence numbers, but we instead implemented a suggested optimization to skip entire terms of divergence. CCF thus finds an agreement point after a sequence of round trips bounded by the number of divergent terms, rather than sequence numbers. We implemented this optimization by changing the semantics of the AE-NACK message, in which followers now include a safe best-estimate of an agreement point communicated using existing fields in the AE-NACK message.

Because the leader cannot distinguish these estimate messages from stale AE-NACKs emitted in previous terms,

it may respond with an AE starting before the end of the follower's log. This, coupled with an insufficiently defensive code path in the follower's code, would cause the AE to be treated as a conflicting suffix, and trigger a roll back preceding the application of the AE, potentially violating Leader Completeness (defined in [74, Figure 3]). The fix proved simple: rather than rolling back optimistically on an AE in a new term, the follower should only do so on true conflicts.

Notably, this bug was triggered by existing functional tests, producing output that did not pass trace validation, but the tests' assertions were not strong enough. The reproduction scenario we wrote produced a trace 305 events long at the point the bug manifests itself. This bug showed the important benefit of trace validation to check the invariants in every state, compared to the traditional approach of manually inserting assertions in scenarios at specific points.

Inaccurate AE-ACK Fixing this bug directly led to the discovery of the AE-ACK index issue. Because CCF uses unidirectional messages rather than RPCs (§2.1), the code responsible for responding to messages sends values from local state where possible, rather than values specific to the message they responded to. The AE-ACK handler did so for the LastIndex field, which is the index of the last entry in the appended entries, without correctly checking that the log was also compatible. This was discovered while conducting trace validation on the previous issue, and a specific scenario was added to test it in isolation. Here too, the fix was simple, and involved constraining the LastIndex in AE-ACKs to the last index contained within the received AE.

Premature node retirement Network configuration in CCF is stored in a map (compare §2.1). Adding or removing nodes happens through updates to this map, which produce write sets that are also replicated and handled like any other transaction, and are therefore part of the totally ordered log. The consensus logic is notified through hooks, which can be called when a transaction is ordered and/or committed. As a result, the reconfiguration logic is not isolated from the consensus code, and was initially mocked with low fidelity in the scenario driver, and correspondingly simplified in the TLA$^+$ spec. This was known to be a significant discrepancy between the implementation and the spec, which we decided to address by making the driver more realistic, and by improving the scenario coverage used for trace validation. As the spec was aligned, simulation produced counterexamples where a reconfiguration would leave the CCF network permanently unable to make progress; a retiring node stopped responding before all future leaders were aware of its retirement. We translated the counterexamples into new functional tests to reproduce the issue. A fix, leveraging an existing mechanism to shut down retired nodes safely, was proposed, verified, and implemented together.

Non-linearizability of read-only transactions Thus far we have described bugs founds in our consensus protocol. In this last example we describe how we identify an ambiguity in CCF's docs, aided by the consistency spec. Recall that linearizability requires that transaction execution is consistent with the real-time ordering of client requests and responses. Recall that ObservedRoInv (Listings 4) specifies that any committed read-only transaction must observe any previously committed read-write transactions. Model checking found a 12-step counterexample to ObservedRoInv in four seconds. A read-only transaction is handled by an old, yet active leader that has not added a read-write transaction to its log since the new leader was elected. This is rare in practice, a leader has to be falsely replaced when it is still active. The leader's logs must be identical, then, in the short window of time before the old leader retires, it needs to handle a read-only transaction.[5] Currently, we have no plans to change this behavior, as serializability for read-only transactions is sufficient for most applications, however, we hope that our consistency spec will help us to more clearly communicate this guarantee to developers.

8 Lessons Learned

We were surprised that, despite our extensive testing, *many bugs were first spotted during the development of the spec and subsequent alignment with the implementation.* The development and refinement of the consensus and consistency specs forced us to think deeply about our protocol and its invariants. During the process of spec development, the implementation was very closely scrutinized (with the target invariants in mind) and many bugs were first identified. In some cases, someone would become suspicious of a particular part of the code but would be unable to confirm that the current behavior led to a violation. This was particularly true for situations including multiple reconfigurations and failure handling (see Incorrect election quorum tally §7 and Commit advance for previous term §7), where counterexamples require many steps and were no longer feasible to work through on a whiteboard. This is where verification was invaluable, as it allowed us to quickly check the behavior against the expected invariants.

While finding implementation bugs by gradually and manually aligning a formal spec with its implementation is possible, we found that *trace validation is a more systematic and efficient approach*. Prior to our trace validation efforts, which began in Spring 2023, we were not confident that our consensus spec matched the implementation. Different team members worked on the spec and the implementation, and as such, they reflected different understandings of how the consensus worked. Moreover, we corrected safety violations present only in the spec, while the implementation contained bugs that spec verification could not find. Trace validation, and its inclusion in our CI pipeline, proved to be a turning point in our verification efforts, as it finally allowed us to systematically identify and fix these discrepancies.

Our results show that *that software verification is beneficial even for systems that have already been "proven in*

[5] The counterexample can be found and interactively explored online [65].

production". During the time that the incorrect election quorum tally (§7) was present in CCF, the operators added and removed nodes one at time. This meant that the bug did not lead to a safety violation and thus remained undiscovered, but could have surfaced if the operators had changed their reconfiguration strategy. Similarly, that the incorrect fix for the commit advance for previous term bug (§7) did not lead to a production incident can be attributed to chance.

Despite modern hardware and advances in tooling, we found that *exhaustive model checking, at our level of abstraction, took significant time to complete*. This led us to limit the state space more than we would have liked. We could have used a proof system, such as the TLA$^+$ Proof System, instead of a model checker. By opting for model checking, we chose to prioritize developer time and accessibility over compute time. As interactive theorem provers continue to advance [13, 75], along with AI-assisted verification [44], this trade-off may no longer be necessary. However, *simulation proved effective at quickly finding bugs*, especially when combined with action weighting (described in §4).

9 Related Work

Validating the correctness of distributed systems is a widely studied problem with approaches ranging from rigorously verified implementations, *formal verification*, to the many flavors of software testing, *casual verification*, including *smart casual verification* which sits between the two.

Formal verification of distributed systems provides the strongest guarantees of correctness, but is often impractical for real-world systems due to the high cost of development and expertise required. For example, IronFleet [24] and Verdi [91] both proved implementations of Raft correct, but, to the best of our knowledge, have not been used outside of an academic setting. Moreover, they are not easily amenable to systems already implemented in a general-purpose programming language. PGo [21] follows a related approach in which one could prove the correctness of a TLA$^+$ spec, and then extract a Go implementation using their PGo compiler. Again, an approach that works best for greenfield projects.

On the other hand, there is a broad spectrum of approaches to testing distributed systems (casual verification) [7, 34, 53], which tend to follow the same pattern: (i) orchestrate the creation of one or more configurations of the system, (ii) schedule workloads, and (iii) inject faults, such as network partitions, node failures, clock skew etc. As the extent of the system and its dependencies being orchestrated increases, it becomes more difficult to maintain determinism and repeatability. Test times for equivalent scenarios also tend to grow longer, and the likelihood of spurious failures goes up.

CCF's consensus spec is the latest addition in a recent tradition of modeling consensus in TLA$^+$. This began with Lamport's description of Paxos [46] in TLA$^+$, as a refinement of higher-level specs [87]. This was followed by the formalization in TLA$^+$ of other consensus algorithms [81, 85, 92] including Paxos variants [20, 29, 30, 48–50] and Raft [73, 80].

Tasiran et al. [84] were the first to extract and validate traces obtained from a hardware simulator against a TLA$^+$ spec, demonstrating the practical applicability of trace validation. The adoption of TLA$^+$ among distributed system practitioners, spurred by Newcombe et al. [69], and the formalization of trace validation as a refinement check by Pressler [76], caused trace validation to be applied to real-world distributed systems. For instance, Davis et al. [16] applied the technique to MongoDB, discovering a non-trivial implementation bug. However, they faced challenges in consistently logging implementation state, and aligning different grains of atomicity, which we attribute to them not leveraging TLA$^+$'s non-determinism to infer implementation state, and action composition to align atomicity. Niu et al. [70] also validated traces of Zookeeper, ensuring that its implementation corresponds to its spec. Similarly, Wang et al. [89] revealed several implementation bugs by replaying TLA$^+$ behaviors against instrumented implementations. Likewise, SandTable is capable of replaying behaviors but, additionally, provides a generic testing framework for distributed systems [83]. SandTable intercepts network communication at the POSIX layer, making it incompatible with systems like CCF that encrypt communications at the application layer. Furthermore, Wang et al. and SandTable serve as examples of the challenges of aligning the grains of atomicity, illustrated by the authors identifying two bugs in Ongaro's well-established Raft [73] spec. We contend that these are, in fact, common TLA$^+$ modeling patterns and can be handled with action composition. Nevertheless, all efforts found non-trivial bugs in real-world systems by comparing implementation traces to high-level TLA$^+$ behaviors; a testament to the effectiveness of this approach.

More pragmatic verification efforts are not limited to TLA+; Amazon's S3 ShardStore service was recently augmented with *lightweight verification* [5] using reference models written in Rust, which are simplified instantiations of program components that can be used to track program state under different input conditions. Like CCF, the primary goals are usability, and the ability to ensure correctness as both the implementation and the spec evolve over time. Unlike the CCF approach however, state exploration is limited to what the test harness is able to reach.

10 Conclusion

This paper details our journey with smart casual verification of the distributed protocols in CCF using TLA$^+$. Our experience demonstrates that TLA$^+$ can be used in industrial settings to verify extensive and nuanced distributed protocols, and that the verification process can be integrated into the development workflow of a production codebase. We have seen that TLA$^+$ can be effectively utilized to find bugs in both the design and implementation of these protocols and to communicate understanding of complex and subtle distributed systems like CCF.

References

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, 2017. URL: https://arxiv.org/abs/1712.01367, arXiv:1712.01367.

[2] Brandon Amos and Huanchen Zhang. 15-812 term paper: Specifying and proving cluster membership for the Raft distributed consensus algorithm, 2015. [Last accessed: 2023-Nov-03]. URL: https://www.cs.cmu.edu/~aplatzer/course/pls15/projects/bamos.pdf.

[3] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. Azure SQL database always encrypted. In *Proceedings of the 2020 International Conference on Management of Data*, SIGMOD '20, pages 1511–1525, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3386141.

[4] Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 1–12, 2021. doi:10.1109/SRDS53918.2021.00010.

[5] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483540.

[6] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 157–168, USA, 2017. IEEE Computer Society. URL: https://arxiv.org/pdf/1701.00981.pdf, doi:10.1109/DSN.2017.45.

[7] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 463–478, USA, 2020. USENIX Association. URL: https://www.usenix.org/system/files/nsdi20-paper-brooker.pdf.

[8] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association. URL: https://www.usenix.org/legacy/event/osdi06/tech/full_papers/burrows/burrows.pdf.

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association. URL: https://pmg.csail.mit.edu/papers/osdi99.pdf.

[10] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, mar 1996. doi:10.1145/226643.226647.

[11] Horatiu Cirstea, Markus A. Kuppe, Benjamin Loillier, and Stephan Merz. Validating Traces of Distributed Programs Against TLA$^+$ Specifications, September 2024. arXiv:2404.16075.

[12] TLA Community. TLA+ google group. URL: https://groups.google.com/g/tlaplus.

[13] Coq. Coq formal proof management system, 2024. [Last accessed: 2024-Mar-25]. URL: https://github.com/coq/coq.

[14] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016. URL: https://eprint.iacr.org/2016/086.

[15] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA$^+$ Proofs. *arXiv:1208.5933 [cs]*, August 2012. arXiv:1208.5933.

[16] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. eXtreme Modelling in Practice. *Proceedings of the VLDB Endowment*, 13(9):1346–1358, May 2020. URL: https://dl.acm.org/doi/10.14778/3397230.3397233, doi:10.14778/3397230.3397233.

[17] Antoine Delignat-Lavaud, Cédric Fournet, Kapil Vaswani, Sylvan Clebsch, Maik Riechert, Manuel Costa, and Mark Russinovich. Why should I trust your code? Confidential Computing enables users to authenticate code running in TEEs, but users also need evidence this code is trustworthy. *Queue*, 21(4):94–122, sep 2023. doi:10.1145/3623460.

[18] Cambridge English Dictionary. Definition for smart casual, 2024. [Last accessed: 2024-May-02]. URL: https://dictionary.cambridge.org/dictionary/english/smart-casual.

[19] Open Enclave. Open Enclave SDK, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/openenclave/openenclave.

[20] Eli Gafni and Leslie Lamport. Disk Paxos. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, page 330–344, Berlin, Heidelberg, 2000. Springer-Verlag. URL: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Disk-Paxos.pdf.

[21] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 159–175, Vancouver BC Canada, January 2023. ACM. doi:10.1145/3575693.3575695.

[22] Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. Understanding inconsistency in Azure Cosmos DB with TLA+. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '23, page 1–12. IEEE Press, 2023. doi:10.1109/ICSE-SEIP58684.2023.00006.

[23] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. URL: https://www.usenix.org/system/files/osdi20-hance.pdf.

[24] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, jun 2017. doi:10.1145/3068608.

[25] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. doi:10.1145/78969.78972.

[26] Lorin Hochstein. Reading the Herlihy & Wing linearizability paper with TLA+, 2022. [Last accessed: 2024-May-05]. URL: https://github.com/lorin/tla-linearizability.

[27] Heidi Howard and Ittai Abrabam. Raft does not guarantee liveness in the face of network faults, 2020. [Last accessed: 2024-Feb-23]. URL: https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/.

[28] Heidi Howard, Fritz Alder, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cédric Fournet, Andrew Jeffery, Matthew Kerner, Fotios Kounelis, Markus A. Kuppe, Julien Maffre, Mark Russinovich, and Christoph M. Wintersteiger. Confidential Consortium Framework: Secure multiparty applications with confidentiality, integrity, and high availability. *Proc. VLDB Endow.*, 17(2):225–240, oct 2023. doi:10.14778/3626292.3626304.

[29] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:14, Dagstuhl, Germany, 2017. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2016.25, doi:10.4230/LIPIcs.OPODIS.2016.25.

[30] Guanzhou Hu. Practical SMR-style TLA+ specification of the MultiPaxos protocol, 2024. [Last accessed: 2024-Apr-26]. URL: https://www.josehu.com/technical/2024/02/19/practical-MultiPaxos-TLA-spec.html.

[31] Intel. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more, January 2020. [Last accessed: 2023-Oct-06]. URL: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[32] Andrew Jeffery, Julien Maffre, Heidi Howard, and Richard Mortier. LSKV: A confidential distributed datastore to protect critical data in the cloud, 2024. URL: https://arxiv.org/abs/2407.12623, arXiv:2407.12623.

[33] Chris Jensen, Heidi Howard, and Richard Mortier. Examining Raft's behaviour during partial network failures. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, HAOC '21, page 11–17, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447851.3458739.

[34] Jepsen-io. Jepsen, 2024. [Last accessed: 2024-May-02]. URL: https://github.com/jepsen-io/jepsen.

[35] Shubhra Sinha Kamath. Integrity protect your Azure blob storage data with Azure Confidential Ledger, 2024. [Last accessed: 2024-Feb-09]. URL: https://techcommunity.microsoft.com/t5/azure-storage-blog/integrity-protect-your-azure-blob-storage-data-with-azure/ba-p/4050754?WT.mc_id=DT-MVP-5001664.

[36] Martin Kleppmann. A critique of the CAP theorem. *CoRR*, abs/1509.05393, 2015. URL: http://arxiv.org/abs/1509.05393, arXiv:1509.05393.

[37] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi:10.1145/3360549.

[38] Igor Konnov, Markus Kuppe, and Stephan Merz. Specification and Verification with the TLA$^+$ Trifecta: TLC, Apalache, and TLAPS. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*, volume 13701, pages 88–105. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-031-19849-6_6.

[39] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), jan 2010. doi:10.1145/1658357.1658358.

[40] Markus A. Kuppe. Teaching TLA$^+$ to Engineers at Microsoft. In Catherine Dubois and Pierluigi San Pietro, editors, *Formal Methods Teaching*, pages 66–81, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-27534-0_5.

[41] Markus A. Kuppe. The TLA$^+$ Debugger. In Paolo Masci, Cinzia Bernardeschi, Pierluigi Graziani, Mario Koddenbrock, and Maurizio Palmieri, editors, *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops*, volume 13765, pages 174–180. Springer, 2023. doi:10.1007/978-3-031-26236-4_15.

[42] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The TLA+ Toolbox. In Rosemary Monahan, Virgile Prevosto, and José Proença, editors, *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019*, volume 310 of *EPTCS*, pages 50–62, 2019. doi:10.4204/EPTCS.310.6.

[43] Hyperledger Labs. Hyperledger private data objects, 2022. URL: https://github.com/hyperledger-labs/private-data-objects.

[44] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems*, volume 35, pages 26337–26349, 2022. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/a8901c5e85fb8e1823bbf0f755053672-Paper-Conference.pdf.

[45] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, may 1994. doi:10.1145/177492.177726.

[46] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133 – 169, may 1998. doi:10.1145/279227.279229.

[47] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, Pearson Education, USA, 2002. [Last accessed: 2023-Oct-06]. URL: https://lamport.azurewebsites.net/tla/book.html.

[48] Leslie Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, oct 2006. doi:10.1007/s00446-006-0005-x.

[49] Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, page 211–224, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.5555/2075029.2075058.

[50] Leslie Lamport. Byzantine Paxos, 2020. [Last accessed: 2023-Nov-03]. URL: https://lamport.azurewebsites.net/tla/byzpaxos.html.

[51] Leslie Lamport, Markus A. Kuppe, Stephan Merz, Andrew Helwer, William Schultz, Jeff Hemphill, Mariusz Ryndzionek, Igor Konnov, Thanh Hai Tran, Josef Widder, Jim Gray, Murat Demirbas, Guanzhou Hu, Giuliano Losa, Ron Pressler, Younes Akhouayri, Luming Dong, Zhi Niu, Lim Ngian Xin Terry, Gaurav Gandhi, Isaac DeFrain, Martin Harrison, Santhosh Raju, Cherry G. Mathew, Fransisca Andriani, and Ludovic Yvoz. TLA+ Examples. Version 1.0.0. URL: https://github.com/tlaplus/Examples.

[52] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, pages 108–117, Monterey California, July 2002. ACM. doi:10.1145/571825.571840.

[53] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC*

Conference on Computer and Communications Security, CCS '23, page 1615–1629, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576915.3623097.

[54] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, Berlin, Heidelberg, 1987. Springer-Verlag. doi:10.1007/3-540-48184-2_32.

[55] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering Shared Objects Without Stable Storage. In Andréa Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2017.36, doi:10.4230/LIPIcs.DISC.2017.36.

[56] Microsoft. LSKV, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/LSKV.

[57] Microsoft. Microsoft Azure Confidential Ledger, 2022. [Last accessed: 2023-Oct-06]. URL: https://learn.microsoft.com/en-us/azure/confidential-ledger/overview.

[58] Microsoft. scitt-ccf-ledger, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/scitt-ccf-ledger.

[59] Microsoft. W3C DID for Confidential Consortium Framework, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/did-ccf.

[60] Microsoft. CCF - more efficient support for linearizable reads (#5636), 2023. [Last accessed: 2024-Apr-19]. URL: https://github.com/microsoft/CCF/issues/5636.

[61] Microsoft. Confidential Consortium Framework, 2023. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/CCF.

[62] Microsoft. Privacy sandbox key management system (kms) for azure, 2023. [Last accessed: 2024-Apr-19]. URL: https://github.com/microsoft/azure-privacy-sandbox-kms.

[63] Microsoft. SQL Server - ledger overview, 2023. [Last accessed: 2023-Nov-03]. URL: https://learn.microsoft.com/en-us/sql/relational-databases/security/ledger/ledger-overview.

[64] Microsoft. TLA+ specifications for the Confidential Consortium Framework, 2023. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/CCF/tree/main/tla.

[65] Microsoft. CCF - document non-linearizability of read-only transactions in rare system condition (#6167), 2024. [Last accessed: 2024-May-07]. URL: https://github.com/microsoft/CCF/pull/6167.

[66] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2517350.

[67] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. Learning-based controlled concurrency testing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, November 2020. doi:10.1145/3428298.

[68] Chris Newcombe. Why Amazon chose TLA+. In *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 8477*, ABZ 2014, page 25–39, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-662-43652-3_3.

[69] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, March 2015. doi:10.1145/2699417.

[70] Zhi Niu, Luming Dong, Yong Zhu, and Li Chen. Verifying Zookeeper based on model-based runtime trace-checking using TLA$^+$. In *Proceedings of the 7th International Conference on Cyber Security and Information Engineering*, pages 13–18, Brisbane QLD Australia, September 2022. ACM. doi:10.1145/3558819.3558822.

[71] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford, 2014. [Last accessed: 2023-Oct-06]. URL: https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf.

[72] Diego Ongaro. Email to raft-dev mailing list, subject line: bug in single-server membership changes, 2015. [Last accessed: 2023-Nov-03]. URL: https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E.

[73] Diego Ongaro. TLA+ specifications for raft, 2020. [Last accessed: 2023-Nov-03]. URL: https://github.com/ongardie/raft.tla.

[74] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305 – 320, USA, 2014. USENIX Association. URL: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf.

[75] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1990.

[76] Ron Pressler. Conjunction Capers: A TLA+ Truffle. https://conf.tlapl.us/2020/, September 2020. URL: https://www.youtube.com/watch?v=TP3SY0EUV2A&list=PLWLcqZLzY8u8EA8UlsZ5xKMvtUoeGr5R_&index=1.

[77] Mark Russinovich. Confidential computing: Elevating cloud security and privacy. *Commun. ACM*, 67(1):52–53, December 2023. doi:10.1145/3624577.

[78] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward Confidential Cloud Computing. *Commun. ACM*, 64(6):54–61, May 2021. doi:10.1145/3453930.

[79] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990. doi:10.1145/98163.98167.

[80] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Scalable, interpretable distributed protocol verification by inductive proof slicing, 2024. URL: https://arxiv.org/abs/2404.18048, arXiv:2404.18048.

[81] William Schultz, Ian Dardik, and Stavros Tripakis. Formal verification of a distributed dynamic reconfiguration protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 143–152, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3497775.3503688.

[82] Pierre Sutra. On the correctness of Egalitarian Paxos. *Information Processing Letters*, 156:105901, 2020. URL: https://www.sciencedirect.com/science/article/pii/S002001901930184X, doi:https://doi.org/10.1016/j.ipl.2019.105901.

[83] Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. SandTable: Scalable distributed system model checking with specification-level state exploration. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 736–753, Athens Greece, April 2024. ACM. doi:10.1145/3627703.3650077.

[84] Serdar Tasiran, Yuan Yu, Brannon Batson, and Scott Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*, 2002. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2002/06/mtv2002-alpha.pdf.

[85] Tendermint. Tendermint specifications, 2021. [Last accessed: 2024-Apr-26]. URL: https://github.com/tendermint/tendermint/tree/master/spec.

[86] TLA+. High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB, 2023. [Last accessed: 2023-Nov-03]. URL: https://github.com/tlaplus/azure-cosmos-tla.

[87] TLA+. TLA+ specifications for Paxos, 2023. [Last accessed: 2023-Nov-03]. URL: https://github.com/tlaplus/Examples/tree/master/specifications/Paxos.

[88] TLA+. TLA+ proof manager (tlapm), 2024. [Last accessed: 2024-Mar-25]. URL: https://github.com/tlaplus/tlapm.

[89] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 127–143, Rome Italy, May 2023. ACM. doi:10.1145/3552326.3587442.

[90] Michael Whittaker. EPaxos dependency set compaction bug, 2021. [Last accessed: 2023-Nov-03]. URL: https://github.com/mwhittaker/bipartisan_paxos/blob/master/epaxos_bugs/epaxos_dependency_bug.pdf.

[91] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, jun 2015. doi:10.1145/2813885.2737958.

[92] Jia-Qi Yin, Hui-Biao Zhu, and Yuan Fei. Specification and verification of the Zab protocol with TLA+. *J. Comput. Sci. Technol.*, 35(6):1312–1323, nov 2020. doi:10.1007/s11390-020-0538-7.

[93] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In Laurence Pierre and Thomas Kropf, editors, *10th IFIP WG 10.5 Conf. Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer. URL: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Model-Checking-TLA-Specifications.pdf.

VEP: A Two-stage Verification Toolchain for Full eBPF Programmability

Xiwei Wu[1], Yueyang Feng*[1], Tianyi Huang*[1], Xiaoyang Lu*[1], Shengkai Lin[1], Lihan Xie[1], Shizhen Zhao†[1], Qinxiang Cao†[1]

[1]Shanghai Jiao Tong University

Abstract

Extended Berkeley Package Filter (eBPF) is a revolutionary technology that can safely and efficiently extend kernel capabilities. It has been widely used in networking, tracing, security, and more. However, existing eBPF verifiers impose strict constraints, often requiring repeated modifications to eBPF programs to pass verification. To enhance programmability, we introduce VEP, an annotation-guided eBPF program verification toolchain. VEP consists of three components: VEP-C, a verifier for annotated eBPF-C programs; VEP-compiler, a compiler targeting annotated eBPF bytecode; and VEP-eBPF, a lightweight bytecode-level proof checker. VEP allows users to verify the correctness of their programs with appropriate annotations, thus enabling full programmability. Our experimental results demonstrate that VEP addresses the limitations of existing verifiers, i.e. the Linux verifier and PREVAIL, and provides a more flexible and automated approach to kernel security.

1 Introduction

Recently, eBPF [15] has gained significant popularity as a versatile technique. This innovative technology enables users to load programs into the Linux kernel dynamically, proving to be highly advantageous across various domains, including networking [2, 6, 29, 34, 46, 49],tracing [4, 43], security [3, 23, 47], storage [7, 51, 52], and more. In contrast to traditional Linux kernel modules, eBPF programs offer enhanced stability, as each program undergoes a rigorous verification process before attachment to the kernel. This verifier meticulously screens programs for unsafe behaviors, such as infinite loops and out-of-bound memory access, thus preventing potential crashes.

Existing eBPF verifiers (e.g., the Linux verifier [19] and PREVAIL [26]) often compromise programmability for security, sometimes rejecting safe eBPF programs. The Linux verifier [19] uses *register value tracking* to simulate the execution paths but limits programmability by imposing constraints such as maximum program size and loop complexity to prevent path explosion. PREVAIL [26] addresses this issue using *abstract interpretation* to merge paths, allowing variable-sized loops in eBPF programs. However, when loops involve complex, dynamic behaviors—such as termination conditions dependent on runtime data—PREVAIL's analysis could become imprecise, leading to safe programs being incorrectly rejected.

The eBPF maintainers have acknowledged this issue and have developed methods to mitigate these restrictions on program size and loop complexity. Since kernel version 3.18, the Linux verifier has employed *pruning* to eliminate redundant code paths [18, 19]. Kernel version 5.2 raised the instruction limit from 4096 to 1 million, and version 5.3 added support for bounded loops [16]. In version 5.13, map iterators were introduced to allow iteration through map elements [17], as loops are always unrolled by the verifier, which can lead to excessive code lengths when iterating over large maps. However, map iterators require a function pointer, which, while suited for functional languages, is cumbersome in C. Despite all these ad-hoc features, some safe programs remain impossible to pass the Linux verifier.

State-of-the-art verification tools have the potential to achieve full programmability. We broadly categorize modern verification tools into three types based on their design approaches when dealing with intricate algorithms, complex program properties, and the need for

*These authors contributed equally to this work.
†Corresponding authors.

high levels of automation: (1) Fully automatic verifiers. CBMC [12], a bounded model checker, verifies array bounds, pointer safety, and other properties in C programs by unrolling loops a fixed number of times. Infer [10], a static analysis tool developed by Meta, automatically checks memory safety in mobile and server-side code. These tools can automatically handle millions of lines of source codes. But their users have to tolerate false positives and negatives, especially when verifying complex programs with intricate algorithms and data structures. (2) Verifiers based on interactive theorem provers. This method can verify complex programs, but it requires significant manual effort in developing machine-checkable *proof code*. VST [24], as a representative of this type of tool, verifies properties of programs in the theorem prover Coq [48] by requiring users to describe the desired properties and manually write proof code to complete the proof. This approach lacks automation, and the time and memory consumption can be significant [1]. (3) Annotation verifiers based on SMT solver. This approach relies heavily on the SMT solver. VeriFast [31] allows users to provide assertions within the program that describe relevant properties, which are then evaluated by the SMT solver for correctness. Similarly, Dafny [33] leverages the Z3 [8] SMT solver to automate the verification of annotated programs. Unfortunately, even powerful and industry-standard SMT solvers such as Z3 and cvc5 [5] still encounter numerous unsolvable or incorrectly solved problems. Moreover, these SMT solvers are often of considerable scale, requiring substantial amounts of time and memory. In short, it is possible for verification tools to verify complex programs without posing any limitation (i.e. achieve full programmability) but with an additional cost such as requiring users to write annotations. Meanwhile, the tools must make trade-offs among efficiency, resource consumption, and potential false negatives and false positives (see Table 1).

For eBPF verification, the verifier must operate within the kernel, necessitating high efficiency, minimal resource usage, and the absence of false negatives. These requirements imply several constraints: First, despite the advanced capabilities of modern SMT solvers, their overhead and risk of false results make them unsuitable for kernel verification. Second, most eBPF programs are compiled with unverified compilers such as LLVM or GCC. We aim to minimize our Trusted Computing Base (TCB), avoiding including such monolithic components. Finally, verification tools must be highly automated and user-friendly to encourage widespread adoption by eBPF developers in practical settings.

Inspired by these considerations, we propose **v**erification toolchain for **e**BPF **p**rograms (VEP) based on an annotation-guided verification approach. The VEP toolchain comprises a C-level verifier, an annotation-aware compiler, and a bytecode-level proof checker. First, verification is conducted on the annotated C programs. To ensure they are free of undefined behavior(UB) per the C standard and only use limited resources. This phase also generates additional annotations and proof terms for later bytecode-level verification. Next, the verified C program is transformed into an annotated bytecode program by an annotation-aware compiler. In this step, the code, the annotations, and the proof terms are all converted to the bytecode level. Finally, a bytecode proof checker re-evaluates the proofs and the annotated bytecode to ensure compliance with eBPF standards, thereby completing the verification process. Our two-stage verification framework distributes substantial time and memory overhead to user space, while the kernel space contains only a proof checker with minimal time and space requirements.

In summary, we have developed a verification tool VEP for annotated eBPF programs, which offers the following advantages:

1. VEP achieves full programmability for eBPF programs, going beyond memory safety to verify the functional correctness of eBPF programs.
2. In user space, the verifier is highly automatic.
3. In kernel space, the proof checker is secure and efficient with lower time and memory consumption.
4. Our TCB is small, which includes only the proof checker in kernel space, excluding the C verifier and the compiler.

The structure of this paper is as follows: In Section 2, we describe the design of VEP, including the general framework and underlying design principles. In particular, we explain the design necessity of the two-stage verification scheme. In Section 3, 4 and 5, we introduce the detailed design of the three components of VEP respectively. In Section 6, we show our experiments and the performance of the proposed method. Section 7 and 8 introduce our future works and make a conclusion.

2 Overall Architecture of VEP

In this section, we will introduce the overall design of VEP and discuss the design choices of the two-stage

[1]The resource requirements of theorem-prover-based tools, such as VST, vary with implementation. Tools built entirely on theorem provers such as Coq and Isabelle [40] tend to have high time and memory demands, while implementations in languages such as C or Python such as VST-A [53] can greatly reduce these costs.

Tool	Full Programability	Automation	Small TCB	Low Time/Memory Cost
CBMC/Infer	-	+++	- - -	+++
VST	+	- -	+++	depend on implementation
VeriFast	+	+	- - -	+

Table 1: Comparison of different verification tools.

verification framework.

2.1 Two-stage Verification

We have developed the verification tool VEP for annotated programs to achieve full programmability, enabling users to write any safe and valid program. Specifically, the VEP toolchain verifies whether an eBPF program (1) is free from aborting behaviors such as null-pointer dereferences and division by zero and (2) adheres to strict memory safety properties, including program-owned memory (e.g., stack, eBPF maps) and kernel-owned memory (e.g., pointers passed via helper functions or context structures).

VEP begins with a C-level verifier, VEP-C, which performs initial checks on user-annotated C programs. Developers can iteratively refine their code using VEP-C until it passes verification. In this process, VEP-C applies symbolic execution to compute the strongest postcondition for each program statement and leverages an SMT solver to derive assertions and generate corresponding proofs automatically. Here, postconditions define the conditions that must hold after executing a program segment, while preconditions specify the assumptions made before execution begins. Symbolic execution systematically computes the strongest postcondition based on the given preconditions, ensuring that the program adheres to its safety and resource constraints.

```
/*@ 0 ⩽ x < 100 */
x = x + 1;
/*@ 0 ⩽ x ⩽ 100 */
```

In this example, the precondition is $P \triangleq 0 \leqslant x < 100$, and the postcondition to be checked is $Q \triangleq 0 \leqslant x \leqslant 100$. We need to verify whether, after executing $c \triangleq$`x=x+1` under the condition P, the postcondition Q holds. The first step is to compute the *strongest* postcondition; that is, the best we can tell about the program state after executing c given the initial condition P. In this case, the strongest postcondition is $\exists x_0, x = x_0 + 1 \wedge 0 \leqslant x_0 < 100$, where intuitively x_0 represents the value of x before the increment operation. During symbolic execution, such computation of strongest postconditions has its logic foundation in Hoare logic rules [28, 41]. The time complexity of the strongest postcondition computation is approximately linear with assertion length. So in general, the time complexity of symbolic execution is approximately linear with the product of the program length and the average assertion length.

Next, we use an SMT solver to determine whether $\exists x_0, x = x_0 + 1 \wedge 0 \leqslant x_0 < 100$ implies $0 \leqslant x \leqslant 100$. Although this example is simple, determining the validity of assertion derivations can be complex in practice. In fact, the decision problem is undecidable in general. Fortunately, existing research on SMT solvers has shown that modern solvers can produce correct results in many practical cases. VEP's built-in SMT solver is required not only to verify whether the derivations hold but also to provide (cvc5-style) proofs. Each proof consists of a list of proof steps. Every proof step derives a new conclusion from known propositions (including assumptions and conclusions proved earlier). Formally, a proof step contains one proof rule name, the associated parameters t_i, the premises φ_i, the resulting conclusion ψ, and its side condition C.

$$\text{RULENAME} : \frac{\varphi_1, \ldots, \varphi_n \mid t_1, \ldots, t_m}{\psi} \text{ if } C$$

In the example of $(\exists x_0, x = x_0 + 1 \wedge 0 \leqslant x_0 < 100) \Rightarrow 0 \leqslant x \leqslant 100$, the VEP's built-in SMT solver produces a 'YES' with a 138-line proof. It means that the code has been successfully verified by VEP-C, with both the corresponding proof and strongest postcondition generated.

Computed strongest postconditions and SMT-generated proofs are inserted back into the original annotated program. The annotation-aware compiler (VEP-compiler) then compiles the elaborated C program into annotated eBPF bytecode. Unlike traditional compilers, relevant assertions and proofs will also undergo corresponding compilation passes.

Finally, VEP uses a bytecode-level proof checker (VEP-eBPF) to perform a final verification. VEP-eBPF similarly uses symbolic execution to compute the strongest postcondition for each statement and then checks the corresponding proofs to derive the assertions, but without requiring SMT solvers: it only checks the proofs based on basic logic rules, using a minimal proof

checker. If the verification passes, the user-written program has successfully undergone VEP's verification process. In the end, the eBPF program will be loaded in the kernel, and its execution is guaranteed to be safe. This security assurance is achieved solely by verifying that the programs and proofs checked by VEP-eBPF are consistent. Therefore, the entire TCB of VEP is effectively reduced to the lightweight proof checker, VEP-eBPF.

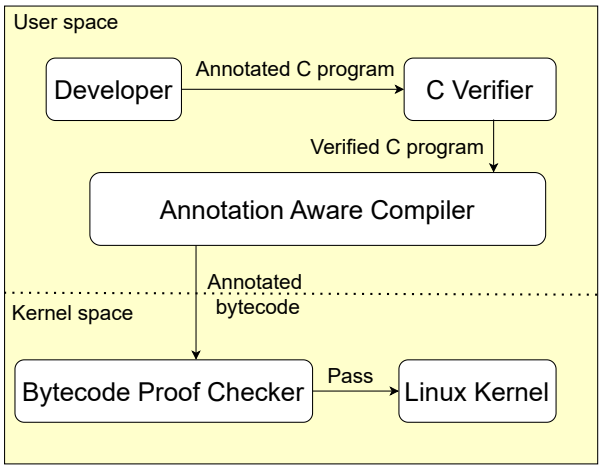


Figure 1: The framework of VEP.

All components of VEP were developed independently, without relying on existing tools. We aim for VEP to replace the current eBPF verification mechanism in Linux. Figure 1 shows the whole workflow of VEP. Our two-stage verification addressed the three requirements mentioned in Section 1.

- **Small and Efficient Kernel Integration.** Our two-stage verification process takes advantage of this by offloading complex SMT solver calls to VEP-C in user space while keeping a lightweight proof checker, VEP-eBPF, in kernel space. VEP-eBPF performs only straightforward logical deductions without complex SMT operations, ensuring the kernel component remains simple and efficient.
- **Minimal Final Trusted Computing Base (TCB).** The safety of loading eBPF programs depends solely on the correct implementation of VEP-eBPF. By isolating the critical verification logic in this small component, we minimize the TCB, thereby reducing risks associated with complex verification algorithms and enhancing overall reliability.
- **Highly Automated and User-Friendly.** While users must provide assertions to prove program safety, our approach leverages lessons from tools like VeriFast and VST by offering an annotation syntax that closely resembles C. This familiar syntax reduces the learning curve, making it easier for users to write and understand assertions.

2.2 Verification Capabilities

VEP is designed to verify critical properties of eBPF programs, ensuring they adhere to both safety and correctness guarantees. Here, safety refers to kernel safety, while correctness pertains to functional correctness verification—the assurance that program implementations precisely match their specified behaviors.

Safety Guarantees: Under the classification framework of KFlex [14], kernel safety consists of two core components: *kernel-interface compliance* and *extension correctness*. Since eBPF programs extend kernel functionality, *extension correctness* ensures safe memory accesses and guarantees program termination. Additionally, as eBPF programs often invoke kernel functions to perform specific tasks, *kernel-interface compliance* is equally critical. This requires that all kernel function calls are valid and that any resources acquired through these calls are properly released, preventing resource leaks.

VEP not only ensures *extension correctness* but also enforces *kernel-interface compliance*. VEP features an annotation language that enables users to formally specify program properties for verifying the following critical properties:

1. Enforcing memory access permissions for load/store operations and function calls.
2. Guaranteeing that all programs terminate within a finite number of steps.
3. Ensuring proper acquire-release ordering discipline for synchronization primitives (e.g., locks and semaphores).
4. Preventing resource leaks through comprehensive lifecycle tracking.

Correctness Guarantees: While the annotation requirements for functional correctness verification are indeed significantly more complex than those for safety verification, our annotation language has been specifically designed to facilitate the precise description of these complex properties. Since functional correctness verification fundamentally builds upon the foundation of safety verification, VEP enforces the requirement that all user-provided annotations must satisfy the four fundamental properties of safety guarantees. This ensures that verified programs maintain both safety guarantees and correctness guarantees throughout the verification process.

2.3 Discussion

In short, we believe an eBPF verifier should be powerful enough to reject unsafe programs and accept correct complex programs, and still be simple enough to be built in the kernel.

Why not use only the C verifier? In a single-phase C verification approach, users need to place a high level of trust in not only the C verifier but also the compiler. But even widely-used compilers such as GCC and LLVM continue to receive numerous bug reports daily. For instance, the LLVM bug tracker has numerous reports of bugs affecting various components of the compiler [22,50], such as the front end, the optimizer, and the code generator. These reports illustrate the necessity for users to trust the compiler while being aware of potential issues. Instead, we have a trustworthy bytecode verifier to ensure the final validation before loading, which is our only TCB.

Why not use only the bytecode verifier? Most importantly, if there is only one bytecode verifier, then it needs a strong SMT solver, which contradicts the simplicity goal of the whole verifier discussed above.

Second, the C verifier provides better computer-human interaction. An optimizing compiler may dramatically change the structure of a C program, making it impossible to convert verification results at the bytecode level back to the source level. If we only report failures at the bytecode level, users must understand how the whole toolchain works only to debug their code. Instead, the C verifier can directly generate C-level feedback, which is more understandable for developers.

A further reason is that writing an annotated bytecode program directly is challenging. Bytecode is very hard to comprehend, and thus it is even more difficult to write suitable assertions for it. Therefore, VEP provides a transformation from C assertions to bytecode assertions through an annotation-aware compiler, which enables users to write more readable assertions based on C code.

Unsafe C programs may be compiled to a safe bytecode program. Can such programs pass VEP? As the compiler is a black box for eBPF developers, we believe VEP should reject undefined behavior (UB) in C at the C verification level. For example, Figure 2 shows an unsafe eBPF program. Due to the insufficient space (size of 20) in *buffer* to store the information of comm, the Linux verifier rejects the function. However, a C program (shown in Figure 3) has out-of-bound array access. After being compiled into bytecode, that access may be a valid

```
1  int badhelpercall()
2  {   char buffer[1];
3      return bpf_get_current_comm(buffer, 20); }
```

Figure 2: Bad helper call A from PREVAIL benchmark.

```
1  int badhelpercall()
2  {   char buffer[1];
3      char buffer2[20];
4      return bpf_get_current_comm(buffer, 20); }
```

Figure 3: Bad helper call B from PREVAIL benchmark.

albeit unintended location, say, *buffer2*. The C standard does not require *buffer2* to be placed right after *buffer*. Compilers could choose to swap them for optimization. Advanced compilers may even detect this undefined behavior (UB) and aggressively exploit this fact by eliminating the whole C function. This program is accepted by version 5.10 of the Linux Kernel but is rejected by version 5.15. We believe that these programs need to be rejected as early as possible. VEP-C will reject all such programs at the C level. Even if such a program were to bypass VEP-C, VEP-eBPF would still reject it during bytecode verification due to the absence of required proofs generated from the C-level assertions.

How much additional efforts are needed for writing annotations? Typically, annotation-based verification tools require users to provide at least function specifications and loop invariants. For complex assertion derivations, users might also need to supply additional assertions to elaborate on the derivation steps. VEP offers some automation support for generating specifications and loop invariants, and we have found that, with this basic support, users can often complete program verification without *any* additional annotations. However, for complex programs, manual input is still required, particularly for verifying functional correctness.

If one manually modifies the generated annotated bytecode, is it possible to trick the VEP-eBPF checker and break the safety guarantee? Simply put: no. It is like you cannot modify the proof of $0 = 0$ to prove $0 = 1$.

In scenarios where a user modifies the annotated bytecode and/or its corresponding proofs, VEP-eBPF still meticulously verifies the alignment between them. If the modifications result in a correct match, meaning that the proof still validates the safety and correctness of the altered bytecode, VEP-eBPF will permit the bytecode to be loaded into the kernel. This ensures that even after modifications, as long as the integrity of the proofs is

maintained, the program remains secure and is considered safe for kernel execution.

Conversely, if a user alters the annotated bytecode and introduces incorrect or inconsistent proofs, VEP-eBPF will detect these discrepancies during the verification process. The proof checker is designed to ensure that only bytecode with valid, accurate proofs can be executed within the kernel. When erroneous proofs are encountered—those that fail to substantiate the safety or correctness of the bytecode—VEP-eBPF will reject the bytecode, preventing it from being loaded into the kernel.

How can we make sure that the pre/postconditions correctly describe the properties that we care? Regarding whether the modified code functions as the user intends, additional proofs related to functional correctness can be provided by the user to ensure this aspect. VEP-eBPF's primary focus is to guarantee that any code it approves is safe to execute. However, the tool will not automatically verify that the modified code behaves as desired; it will only ensure that the code can be safely run without introducing security vulnerabilities. Therefore, it is up to the user to include further assertions and proofs to confirm that the program's functionality aligns with their expectations.

2.4 Related Work

To the best of our knowledge, VEP is the first annotation-guided eBPF verification toolchain. In theory, for any correct eBPF program written by programmers, there exists a way to add annotations to pass VEP.

Most existing eBPF verifiers, including the Linux verifier [19], PREVAIL [26], etc., are automatic verifiers. Admittedly, automatic verifiers are easy to use. But in theory, checking whether an eBPF program can be safely executed and terminated is an undecidable problem. Indeed, both the Linux verifier and PREVAIL encounter problems when handling loops with complex data structures and may generate false positive results for certain eBPF programs, which limits eBPF programs from achieving full programmability.

We have identified ExoBPF [38] and Serval [37], which necessitate users to add specifications at the bytecode level. These tools conduct symbolic execution utilizing theorem provers and subsequently employ an SMT solver to resolve constraints. Specifically, ExoBPF utilizes Lean, while Serval leverages Rosette. Although these tools are indeed powerful, they require users to be familiar with eBPF bytecode and theorem provers to effectively write bytecode specifications.

Previous research in annotation-based verification has laid important theoretical foundations. Necula established a proof-carrying code (PCC) framework where code producers generate safety proofs that can be efficiently verified by the code consumer [35, 36]. While these PCC concepts have inspired later verification systems, existing annotation-based tools like Vale [9] and Ironclad Apps [27] demonstrate limitations when adapted for eBPF verification. Vale leverages Dafny's Z3-based verification to analyze annotated assembly code for cryptographic implementations, while Ironclad Apps streamlined Z3's capabilities for OS-level verification. Though effective for assembly code verification in specific domains, these approaches impose substantial learning overhead for eBPF developers who must reason about low-level memory semantics and instruction sets. Our methodology builds upon PCC's foundational principles but shifts the verification abstraction layer - rather than requiring assembly-level annotations or specialized proof generators, we enable developers to annotate directly at the C source level. This creates an end-to-end verification chain from high-level semantics to eBPF bytecode, crucially mediated by a lightweight verification engine optimized for kernel integration.

3 Detailed Design of VEP-C

VEP-C is an annotation-based verifier for annotated C programs, implemented based on traditional ***symbolic execution*** algorithm with an ***entailment solver*** to check the validity of the verification conditions. It uses separation logic assertions to represent program states and performs symbolic execution. The entailment solver is based on a separation logic elimination solver and an SMT solver. During the verification process, it generates proofs for assertion derivations. In this section, we will detail the design of VEP-C, focusing on the assertion syntax, the symbolic execution process, and the verified program.

3.1 Verification Process of VEP-C

Throughout this subsection, we use the annotated program *memset* in Figure 4 as an example[2]. In this example, we intentionally write "`i = 0; for (; i < n;) { ...; i ++ }`" (respectively in line 5, 8, and 19) rather than the commonly used "`for (i = 0; i < n; ++ i) { ... }`" to illustrate the verification steps more conveniently. This is not a requirement or restriction in realistic

[2]To facilitate the reader's understanding, we present a version of the annotated program that leans more towards functional correctness. Assertions in actual code are more straightforward.

verification tools.

```
1  void memset(char *p1, __u32 n, char v)
2  /*@ With l1
3      Require chars(p1,n,l1)
4      Ensure ∃ l2, chars(p1,n,l2) */
5  { __u32 i = 0;
6   /*@ i == 0 && chars(p1,n,l1) */
7   /*@ Inv: ∃ l2, 0 ⩽ i ⩽ n && chars(p1,n,l2) */
8   for (; i < n; ) {
9   /*@ ∃ l2, 0 ⩽ i < n && chars(p1,n,l2) */
10  p1[i] = v;
11  /*@ ∃ l3 l2, 0 ⩽ i < n &&
12        l3[0:i] == l2[0:i] &&
13        l3[i] == v &&
14        l3[i+1:n] == l2[i+1:n] &&
15        chars(p1,n,l3) */
16  i++;
17  /*@ ∃ l3 l2, 0 ⩽ i - 1 < n &&
18        l3[0:i-1] == l2[0:i-1] &&
19        l3[i-1] == v &&
20        l3[i:n] == l2[i:n] &&
21        chars(p1,n,l3) */ }
22  /*@ ∃ l2, i == n && chars(p1,n,l2) */
23  return ; }
```

Figure 4: Annotated memset.

Line 2-4: In the beginning, the user needs to provide a function specification, which indicates the condition that the arguments together with the initial program state need to satisfy when entering the function and the condition that needs to be satisfied when exiting the function. A function specification consists of a **With** clause, a **Require** clause, and an **Ensure** clause. The **Require** clause indicates the function precondition; the **Ensure** clause indicates the function postcondition; and the **With** clause indicates the list of logical variables mentioned in precondition which will be used in the whole program assertions. In this example, the precondition is `chars(p1, n, l1)` (line 3), which indicates that `p1` is an array of length `n`, with its data `l1` being a list of characters. The postcondition provided in line 4 specifies that the array stored at `p1` will be modified to a new character list `l2`.

Line 5-6: To verify the program in Figure 4, VEP-C initiates symbolic execution based on the preconditions of the function. For instance, after processing the variable declaration and initialization at line 5, the strongest postcondition in line 6 is computed. In this example, assertions generated through symbolic execution are highlighted in red, distinguishing themselves from the assertions provided by the user, which are displayed in blue.

Line 7-22: Before entering the loop, the user needs to provide a loop invariant, which indicates the property that the program state needs to satisfy at the beginning and end of each loop iteration. VEP-C will check whether the assertion before entering the function (lines 6) implies the loop invariant (line 7). If the above check passes, VEP-C will continue to symbolically execute the loop condition testing from the loop invariant. Thus, in its strongest postcondition (line 9), an additional proposition $i < n$ is added, comparing to the `Inv`. Moreover, symbolic execution will generate the strongest postcondition of the assignment statement (lines 11-15), and the strongest postcondition of the incremental step (lines 17-21). VEP-C will check whether the assertion entails the loop invariant `Inv`. The check is successful, thereby proving the correctness of the loop invariant. Subsequently, we can derive the strongest postcondition at the end of the loop, which is the strongest postcondition of `Inv` when the loop condition evaluates to false (line 22).

Line 23: Finally, we perform symbolic execution on the return statement. Unlike other statements, the return statement requires the calculation of the return value (although there is no return value in this example), followed by the deallocation of all declared local variables and corresponding updates of the assertions. In this case, the variable `i` is deallocated, and the assertion is updated accordingly into `∃ l2, chars(p1,n,l2)`. The final step is to check whether the updated assertion implies the postcondition of the function. The proof here is straightforward, thus completing the verification of this example.

3.2 Assertion Language of VEP-C

In the example of Figure 4, we used first-order logic to describe program states. For some C programs, such assertions are sufficient to capture time costs and the changes made to memory. However, most programs manipulate more than one data structure (array, linked list, tree, etc). These data structures are stored in disjoint memory locations and such disjointness is critical in verification. For example, the C standard says that *strncpy* copies strings from source to destination, while the source and destination memory space are disjoint [30, Section 7.24.2.4]. Therefore, some naive specification like the one in Figure 5 does not correctly describes strncpy's behavior.

```
1  void strncpy (char *p1, char *p2, __u32 n)
2  /*@ With l1 l2
3      Require chars(p1,n,l1) && chars(p2,n,l2)
4      Ensure chars(p1,n,l2) && chars(p2,n,l2) */;
```

Figure 5: A naive strncpy specification.

This specification appears to be very concise, but it overlooks a critical issue: whether the two arrays, `p1` and `p2`, overlap. If there is an overlap, the behavior of the program could differ significantly, leading to potential unintended side effects that are not accounted for by this

simple specification. Therefore, merely using first-order logic is insufficient to meet our needs.

State-of-the-art research has provided us with new insights, particularly through the introduction of separation logic into assertions. Tools such as VST, VeriFast, and Hip/Sleek [39] have demonstrated that using separation logic is an effective method for clearly describing properties and facilitating symbolic execution, especially in proofs related to memory properties.

Separation logic [41] uses a new connective *separating conjunction* to ensure that different names duplicate no identical addresses. The separating conjunction $P * Q$ represents the existence of two disjoint portions of the state, one that satisfies P and one that satisfies Q. Specifically,

$$m \models P * Q \Leftrightarrow \exists m_1, m_2 . m = m_1 \uplus m_2 \wedge m_1 \models P \wedge m_2 \models Q.$$

Here $\uplus$ means the disjoint union[3] and $m \models P$ means m satisfies P. A distinction between $*$ and boolean conjunction && is that $P * P \neq P$ where $P \&\& P = P$. In particular, if $\mathbf{store}(p, v)$ means that the value v is stored at address p, $\mathbf{store}(p, v) * \mathbf{store}(q, u)$ implies $p \neq q$, and thus $\mathbf{store}(p, v) * \mathbf{store}(p, v)$ is always false: there is no way to divide a heap that a cell p goes to both partitions.

Separation logic predicates can also be used to represent kernel resources [13,41], like semaphores and locks. Intuitively, when a helper function is used to gain a piece of resource, the corresponding predicate will be added to the assertion. When a helper function is used to release a piece of resource, the predicate are required to appear beforehand (which will be checked by VEP) and will then be removed. In separation logic, those resources act like memory. We show an example in Appendix E.

VEP-C uses separation-logic assertions in the canonical form of a symbolic heap. A symbolic heap is in the form $\exists \vec{x}.(P_1 \wedge \cdots \wedge P_n \wedge Q_1 * \cdots * Q_m)$, where the pure part P describes memory-irrelevant properties between terms (e.g. `e1`, `e2`), and the spatial part Q is a separating conjunction of spatial predicates. For example, `e1==e2` and `e1>e2` can appear as memory-irrelevant conjuncts; the empty heap predicate `emp` and the points-to predicate `store(e1,e2)` can appear as spatial conjuncts. Additionally, users can define their own predicates according to their specific needs. For example, a user can define a string array predicate such as `chars(a,e1,e2)`.

With the help of separation logic, we can write the specification shown in Figure 6, which describes the scenario stipulated by the C standard where `p1` and `p2` do not overlap in memory.

[3] $A \uplus B = A \cup B$ if $A \cap B = \varnothing$, Otherwise, $A \uplus B$ is undefined. For example, $\{1,2,3,5\} = \{1,3\} \uplus \{2,5\}$, but $\{1,3\} \uplus \{1,2\}$ is undefined.

```
1  void strncpy (char *p1, char *p2, __u32 n)
2  /*@ With l1 l2
3      Require chars(p1,n,l1) * chars(p2,n,l2)
4      Ensure ∃ l3,
5              chars(p1,n,l3) * chars(p2,n,l2) */;
```

Figure 6: A correct strncpy specification.

Traditionally, these terms (`e1`, `e2`, etc) above in pure parts and spatial parts should be memory-irrelevant expressions. For example, if `x` is a C variable of a struct type, then `&(x.tail)` is a memory-irrelevant expression, because computing its value does not include a load from memory. Many verification tools use this setting internally because this assertion language prevents a lot of ambiguity. For example, it may be unclear whether the predicate `store(*x, 0)` only claims the memory permission at address `*x`, or it claims the memory permission at addresses `*x` and `x` – if we take the address of storing variable `x` into consideration, it becomes even more complicated. VEP-C extends traditional symbolic heap and allows users to use memory-related expressions, which makes assertions more concise. VEP-C automatically transforms the expression to the traditional symbolic heap. In this way, users avoid a large portion of boilerplate[4]. For example, VEP-C allows users to write `*y==x→tail`, which is equivalent to:

```
∃ v yp,
store((field_addr(x,tail),v) *
store(y, yp) * store(yp, v)
```

Through the application of separation logic assertions, users gain the capability to formally specify function properties while simultaneously verifying both the fundamental safety guarantees (Properties 1 and 2 as detailed in Section 2.2) and comprehensive functional correctness properties within a unified verification framework.

Verification when helper functions involve in The acquisition and release of kernel resources of eBPF programs relies on helper functions, making the symbolic execution of helper function calls a critical component in the verification process. The VEP framework incorporates predefined specifications for most helper functions, formally characterizing their interactions with memory subsystems and kernel resource management. These specifications are derived from eBPF's official documentation [1]. Figure 7 presents the specifications for `bpf_sk_lookup_udp` and `bpf_sk_release`, while Figure 8 illustrates the symbolic execution behaviors of these two helper functions. The complete program is provided

[4] The detailed syntax and transformation algorithm can be found in Appendix A and B.

in Appendix E.

```
struct bpf_sock * bpf_sk_lookup_udp
    (void *ctx, struct bpf_sock_tuple *tuple,
     __u32 tuple_size, __u64 netns, __u64 flags)
/*@ xdp_version
    Require flags == 0 &&
      valid_size(tuple_size, tuple) &&
      store_xdp(ctx) * store_sock_tuple(tuple)
    Ensure store_sock(__return) * store_xdp(ctx) *
      store_sock_tuple(tuple) */;
```

```
int bpf_sk_release (void * sock)
/*@ sock_version
    Require sock != NULL && store_sock(sock)
    Ensure ∃ v, __return == v */;
```

Figure 7: The specification of `bpf_sk_lookup_udp` and `bpf_sk_release`

In Figure 7, the `valid_size` predicate ensures that the value of `tuple_size` correctly corresponds to `tuple`. Specifically, if `tuple` represents an IPv4 socket, then `tuple_size` must be `sizeof(tuple→ipv4)`, and similarly for IPv6. The `store_sock(sk)` predicate ensures that if `sk` is a null pointer, it corresponds to an empty memory region. Otherwise, if `sk` is non-null, it provides access to a valid socket. This definition guarantees that memory operations on `sk` can only be performed after checking whether `sk` is null.

```
1  /*[auto generated pre]
2      ipv4(tuple) && store_xdp(ctx) *
3      store_sock_tuple(tup) */
4  sk = bpf_sk_lookup_udp(ctx, &tup,
5          sizeof(tup.ipv4), 0, 0);
6  /*[auto generated]
7      store_xdp(ctx) * store_sock_tuple(tup) *
8      store_sock(sk) */
9  ...
10 /*[auto generated pre]
11     sk != NULL && store_xdp(ctx) *
12     store_sock_tuple(tup) * store_sock(sk) */
13 bpf_sk_release(sk);
14 /*[auto generated]
15     store_xdp(ctx) * store_sock_tuple(tup) */
```

Figure 8: The symbolic execution result of `bpf_sk_lookup_udp` and `bpf_sk_release`

In Figure 8, we assume that `tup` is an IPv4 socket before the helper function call in line 4. Therefore, the function call on line 4 satisfies the function specification of `xdp_version`, allowing us to obtain a `store_sock(sk)`. When execution reaches line 12, we know that `sk` is non-null. Consequently, the function call on line 12 satisfies the function specification of `sock_version`, enabling us to properly reclaim the allocated socket memory.

By leveraging these built-in helper function specifications, VEP automatically enforces the verification of safety guarantee Properties 3 and 4 (as specified in Section 2.2) during symbolic execution. The full list of helper function specifications can be found in our GitHub repository.

3.3 Output of VEP-C

After the symbolic execution and entailment solver processes, VEP-C completes the verification of the input program. Based on the verification process, VEP-C generates a verified C program, which is then passed to the VEP-compiler to be compiled into bytecode. Figure 9 illustrates the verified C program corresponding to the program in Figure 4. Due to space constraints, we have omitted the specific assertions and proof content.

```
1  void memset(char *p1, __u32 n, char v)
2  /*@ With l1
3      Require chars(p1,n,l1)
4      Ensure chars(p1,n,repeat(v,n)) */
5  { __u32 i = 0;
6   /*@ Assertion\_1 with Proof\_1 */
7   /*@ Inv with Proof\_2 and Proof\_6 */
8   for (; i < n; ) {
9   /*@ Assertion\_2 with Proof\_3 */
10  p1[i] = v;
11  /*@ Assertion\_3 with Proof\_4 */
12  i++;
13  /*@ Assertion\_4 with Proof\_5 */ }
14  /*@ Assertion\_5 with Proof\_7 */
15  return ; /*@ Proof\_8 */ }
```

Figure 9: Verified memset.

In this example, `Proof_1`, `Proof_3`, `Proof_4`, `Proof_5`, and `Proof_7` pertain to the verification of various safety checks during symbolic execution. These include range checks for array writes, as well as checks to ensure that there are no undefined behaviors during assignments. `Proof_2` and `Proof_6` are associated with the verification of the validity of loop invariants. `Proof_8` ensures that the function postcondition is satisfied upon completion of the function.

4 Detailed Design of VEP-compiler

An annotated C program undergoes several compilation passes before being converted into an annotated eBPF bytecode. While most of these passes are standard, modifications to assertions and proofs are necessary throughout the process. It's worth noting that the VEP-compiler contains only a few optimization passes at present, primarily for simplicity, which may not generate the most efficient bytecode. How to incorporate additional compilation optimizations while ensuring accurate transformation of assertions and proofs is a future work.

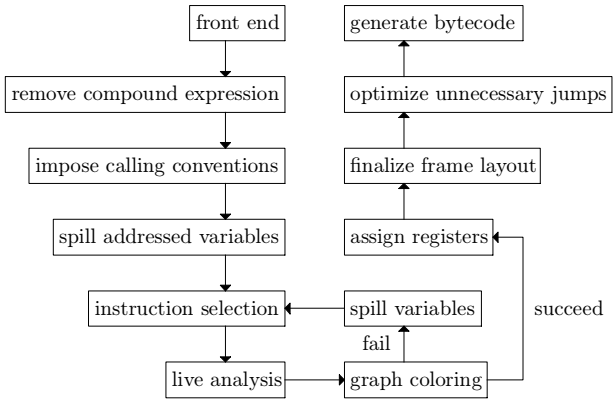


Figure 10: The nanopass-style compilation process

```
//@ ∃ v, store((field_addr(x,tail),v)
                ⇓ compile
        //@ ∃ v, store(x + 8,v)
```

Figure 11: IR Generation example

Figure 10 shows the whole compilation process. Every pass has a single simple and clear purpose, as in the nanopass framework [44]. We discuss the transformation during relevant passes in the rest of this section.

4.1 IR Generation

In our intermediate representation, we replace structure-member access by dereferencing the address of the structure plus the offset of the member. We do the same for assertions, as illustrated in Figure 11.

4.2 Calling Conventions

The BPF calling convention is defined as follows.

1. `R0` stores the return value.
2. `R1` to `R5` are used to pass arguments.
3. `R6` to `R9` are callee saved. Others are caller-saved.

All the helper functions obey the suggested calling convention. We adopt the same convention in compiling in-program functions, so the compiler modifies function specifications to reflect the calling convention as follows.

1. Return value `__return` is replaced by `R0`.
2. Function parameters are replaced by `R1` to `R5`.
3. State that `R6` to `R9` are preserved.

The generated code saves used callee-saved registers on the stack at the beginning of a procedure and restores them before returning. To convince the symbolic executor that callee-saved registers are indeed unchanged, we also modify each assertion in the procedure. They state that certain portions of the stack contain the original values of callee-saved registers.

```
strncpy:
/*@ With l1 l2 _R1 _R2 _R3 _R6 _R7 _R8 _R9
    Require
      _R1 == R1 &&
      _R2 == R2 &&
      _R3 == R3 &&
      _R6 == R6 && _R7 == R7 &&
      _R8 == R8 && _R9 == R9 &&
      chars(R1, R3, l1) * chars(R2, R3, l2)
    Ensure
      _R6 == R6 && _R7 == R7 &&
      _R8 == R8 && _R9 == R9 &&
      chars(_R1,_R3,l2) * chars(_R2,_R3,l2) */
```

Figure 12: Transformation of the function specification for strncpy. `p1`, `p2`, `n` and `__return` are replaced by `R1`, `R2`, `R3`, and `R0`, respectively.

Figure 12 shows the specification in Figure 5 produced by this pass. Besides the substitution defined previously, the compiler introduces auxiliary logic variables `_Ri`-s to relate the register values at function entry and those at the return point.

4.3 Register Allocation

We adopt a standard "iterated register coalescing" algorithm [25]. Most of the time, simply substituting variables in assertions with their corresponding registers (or their location on the stack, if it is spilled) is enough. The only exception is when the variable is not live (the value of the variable is not used later) at that point. In that case, the register does not necessarily hold the value of the variable: maybe another variable is using it. To make the assertion valid while retaining information, the variable should be substituted by an existentially quantified logic variable. Figure 13 shows a complete example. The third line in the source code takes the address of `x`, so `x` is spilled to the stack. `p` and `q` are assigned to the same register `R1` because they are unused.

In annotated code, register allocation can sometimes yield interesting results. Figure 14 shows one example.

In the previous example, the C code clearly fails VEP-C verification, but the compiler generates valid eBPF bytecode for it. This is due to liveness analysis determining that the value of x is not live at that point. We believe rejecting such code at the VEP-C level is reasonable and does not compromise programmability. Even if one only uses VEP-compiler and VEP-eBPF, he/she can still trust the safety of the C code, because (1) the precondition is unchanged, and (2) propositions related to resources (time, memory) are unaffected.

```
int x, y, *p, *q;
x = 0; y = 1;
p = &x; q = 0;
//@ y == x + 1 && p != q
return y;
```

⇓ compile

```
*(R10 - 4) = 0
R0 = 1
R1 = R10
R1 -= 4
R1 = 0 // p and q are not used
/*@ R1 == *(R10-4) + 1 &&
    ∃ _p _q, _p != _q */
ret
```

Figure 13: An assertion after register allocation.

```
int x = 0;          R0 = 0
//@ x == 1    ⇒     //@ ∃ _x, _x == 1
return;             ret
```

Figure 14: Example of register allocation.

4.4 Frame Layout

In annotated C programs, a variable is available after we declare it. But in bytecode, there are no such declarations. So the frame—which contains spilled scalar variables, structures, and so on—should be specified in the precondition so that the symbolic executor knows which addresses are valid. Knowing how much stack space a procedure consumes is also necessary to verify that the whole program does not exceed the stack space limit.

Figure 15 shows a complete example. In the example, we assume the compiler places `l` at `R10-16`. Consequently, `R10-16` stores an integer, and `R10-8` stores a pointer to a `struct list`. The underscores indicate that their actual values are not needed.

5 Detailed Design of VEP-eBPF and Proof Check

VEP-eBPF processes the annotated bytecode produced by the VEP-compiler and performs symbolic execution. During this execution, when it encounters assertion entailment or safety checks typically requiring an entailment solver, VEP-eBPF distinguishes itself from VEP-C by not invoking an SMT solver. Instead, it verifies whether the proofs, generated by the SMT solver and transformed by the VEP-compiler, correctly establish the required entailments. If all entailments can be validated through their corresponding proofs, it indicates that the bytecode is safe, allowing the program to be loaded into the kernel.

VEP-eBPF's proof language is designed to include

```
struct list {
  int x;
  struct list *next;
};
int f()
/*@ Require emp
   Ensure __return == 0 */
{ struct list l;
  l.head = 0;
  return l.head; }
```

⇓ compile

```
f: /*@ Require
     *(R10-16) == _ *
     *(R10-8)  == _
   Ensure R0 == 0 */
...
```

Figure 15: A precondition decorated with frame information. Assume `l` is located at `R10-16`, then the compiler guarantees that `R10-16` stores an integer and `R10-8` stores a pointer to a `struct list`. The underscores indicate that they are uninitialized.

both spatial parts derivation proofs and pure propositional parts derivation proofs. The spatial parts proofs are primarily syntactic transformations of the entailment using established separation logic properties. These properties have been formalized and proved within the Coq proof assistant, ensuring their correctness. On the other hand, the pure parts proofs are based on a proof language inspired by cvc5, a well-known SMT solver. In this process, no SMT solving is required. The spatial syntax transformation and pure proof checking are handled by highly efficient algorithms.

As a result, VEP-eBPF is an efficient, low-memory proof checker ideally suited for kernel-space deployment. The lightweight nature of these algorithms contributes to the overall performance, making VEP-eBPF a robust tool for verifying the safety of eBPF bytecode without the overhead of traditional SMT-based methods.

6 Evaluation

We evaluate the time cost, memory efficiency, and verification accuracy of VEP compared to the Linux verifier (Kernel 5.15) and PREVAIL (August 2024 version). Our benchmark selected a total of 41 programs across four categories of programs: Linux kernel samples, PREVAIL test cases, C standard library string functions, and unsafe programs with known vulnerabilities, such as out-of-bounds memory accesses and invalid function call. Detailed per-program results are provided in Appendix C (Tables 4 and 5), and all data is publicly available in our

GitHub repository[5].

6.1 Performance Analysis

The Linux verifier demonstrates superior stability with consistent sub-millisecond verification times (avg. 0.94 ms) and low memory consumption (<5.2 MB) across all benchmarks (Table 2). In contrast, VEP-eBPF achieves 3–5× faster verification than both PREVAIL (avg. 13.88 ms) and VEP-C (avg. 39.46 ms), while maintaining predictable memory usage of 2–3 MB – a 75% reduction compared to VEP-C's 8.5–32.6 MB footprint. This efficiency gain stems from our novel design choice discussed in Section 2.

6.2 Verification Accuracy

While all tools successfully reject 100% of unsafe programs (10/10 PR), critical deficiencies emerge in their handling of safe implementations. The Linux verifier demonstrates particularly severe limitations, accepting only 60% of safe programs (6/10 PR) with a 40% rejection rate - primarily stemming from inadequate loop verification (e.g., StringLib functions). PREVAIL exhibits even more erratic behavior, achieving wildly inconsistent acceptance rates from 10% (1/10 PR) on StringLib to 80% (8/10 PR) for its own test suite, a consequence of its SMT solver dependency introducing non-determinism.

This version-induced instability compounds the problem - 30% of PREVAIL samples and 20% Linux kernel programs that previously passed verification get rejected after updates, exposing fundamental flaws in their architectural design. Such brittleness severely undermines their reliability for production deployment.

In stark contrast, VEP achieves perfect verification outcomes across all benchmarks: 100% acceptance (10/10 PR) for properly annotated safe programs and 100% rejection of unsafe implementations. As long as the user provides correct annotations, VEP ensures there are no false positives and no false negatives. This deterministic behavior persists across kernel versions, demonstrating VEP's architectural superiority in balancing verification rigor with practical stability.

6.3 Annotation Efficiency

Tables 2 and 3 present the total lines of code, the number of user-written assertions, and the corresponding proof obligations generated by VEP. The comparison highlights that VEP requires a relatively modest number of assertions to verify programs effectively. For example, in the Linux samples with 618 lines of code, only 76 assertions were needed, resulting in 64,840 proof obligations. This indicates that a small number of high-level assertions can generate comprehensive proof coverage. Although the generated proofs are extensive, the verification process remains efficient because our proof checker is highly lightweight, ensuring minimal verification time.

Regarding the effort required to write annotated programs, we observe that adding annotations does not demand extensive modifications to the original code. Most assertions involve standard properties such as memory safety, resource bounds, and function preconditions, which can be specified with basic knowledge of program verification concepts. The expertise required is comparable to writing simple function contracts or preconditions in languages with contract support.

Furthermore, VEP supports the generation of specifications for several functions through lightweight methods, reducing manual effort. As VEP evolves, we aim to automate more of the assertion generation process, further minimizing the manual effort required and lowering the expertise barrier for new users.

6.4 Case Study : Key_Connection

The *Key_Connection* example (last row in Tables 2-3) demonstrates VEP's ability to verify non-eBPF network functions – specifically, an L7 filter mapping string keys to backend servers. Detailed code and the annotations for this example can be found in Appendix D. For this program, VEP generates 350 lines of annotated bytecode and 5,800 lines of proof.

This capability addresses critical limitations in current eBPF frameworks like Cilium [11], which must offload complex L7 logic to external proxies (Envoy [21]) due to verification challenges. By enabling verifiable L7 processing, VEP could eliminate proxy dependencies while maintaining safety guarantees.

7 Future Work

In advancing eBPF program verification, several critical areas offer promising research opportunities. This section outlines key directions for future work designed to enhance and expand the capabilities of our verification framework, aiming to make it more robust and applicable to real-world scenarios.

[5] https://github.com/yashen32768/NSDI25-VEP-535

Programs Source	Total Code Lines	Linux verifier				PREVAIL			
		PR	MaxT (ms)	AvgT (ms)	MaxM (KB)	PR	MaxT (ms)	AvgT (ms)	MaxM (KB)
Linux samples	618	9/10	1.13	0.94	4196	10/10	48.37	12.48	7918
PREVAIL samples	252	6/10	1.08	0.71	4200	8/10	74.67	13.88	5279
StringLib	321	3/10	2.69	1.83	5168	1/10	39.89	39.89	7267
Unsafe Programs	195	10/10	0.051	0.038	4340	10/10	34.88	6.05	5312
Key_Connection	63	0/1	0.025	0.025	4304	0/1	2.312	2.312	5004

Table 2: Evaluation results for Linux verifier and PREVAIL.

Programs Source	Total Assertion Lines	Total Proofs Lines	VEP-C				compiler	VEP-eBPF		
			PR	MaxT (ms)	AvgT (ms)	MaxM (KB)	AvgT (ms)	MaxT (ms)	AvgT (ms)	MaxM (KB)
Linux samples	76	64840	10/10	158.12	39.46	32569	1.73	21.69	8.42	8034
PREVAIL samples	49	41931	10/10	40.23	9.38	12422	0.47	4.40	2.76	3047
StringLib	112	56117	10/10	36.60	13.47	12612	0.66	3.54	2.63	2970
Unsafe Programs	32	-	10/10	5.36	3.32	2955	-	-	-	-
Key_Connection	17	5819	1/1	16.24	16.24	8534	0.56	2.48	2.48	2440

Table 3: Evaluation results for VEP. 'PR' refers to the acceptance rate for safe programs and the rejection rate for unsafe programs.

Towards Functional Correctness As our demands for programs increase, memory safety alone no longer meets our development requirements. We aim to go further by supporting the verification of the functional correctness of eBPF programs. From this perspective, VEP needs to enable users to directly write proofs within C annotations or provide an interface to incorporate external proofs.

Towards Less Annotations Given that VEP currently requires users to provide necessary C annotations, including function preconditions, postconditions, and loop invariants, we aim to introduce additional tools to automate the generation of these annotations and reduce the user's burden. Traditional verification tools [20, 32] have already made progress in this area, and recent advancements have been achieved by integrating large language models [42, 45]. In the future, VEP will be able to adopt similar approaches to automate the generation of some annotations, thereby minimizing the user's workload.

Towards Compilation Optimization Advanced optimization passes may dramatically change the control flow and data flow of a program, and assertion annotations should be modified accordingly. In our current compiler, one such compilation pass is register allocation, in which multiple variables may be represented by one register (as long as their lifetime does not overlap) — we translate all these variables to the register in assertions. In the future, optimization passes may be added to eliminate some redundant instructions or redundant variables. Then, corresponding algorithms need to be designed to compile assertion annotations.

8 Conclusion

Since its proposal, eBPF is widely used in various domains while its verification often comes with a trade-off between safety and programmability. In this paper, we propose a two-stage automatic formal framework that addresses this issue between verification and programmability. It empowers eBPF developers to write annotations regarding the memory to guide automatic verification. As far as we are concerned, this is the first approach that could verify the safety and resource constraints of eBPF programs without compromising programmability. Specifically, a safe eBPF program can always pass VEP's check, provided that it includes sufficient annotations.

Based on this, we implement a prototype toolchain VEP. Our evaluation further demonstrates that VEP can verify the safety of complex programs with moderate overhead, which highlights the potential of the two-stage framework as a practical solution for kernel security. We believe this will enable eBPF to support more flexible and powerful programs across a wider range of domains.

Acknowledgements This work was supported by the NSF China (No. 62272292 and 62472274). We also thank our shepherd Rishabh Iyer and the NSDI reviewers.

References

[1] ebpf docs. https://docs.ebpf.io/linux/helper-function/.

[2] The Cilium Authors. Cilium. https://cilium.io/.

[3] The Falco Authors. falco. https://falco.org/.

[4] The Pixie Authors. pixie. https://pixielabs.ai/.

[5] Haniel Barbosa, Andrew Reynolds, Tim King, Mudathir Mohamed, Andres Noetzli, Andrew V. Sutherland, Cesare Tinelli, Clark W. Barrett, Morgan Deters, and Mathias Preiner. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[6] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[7] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, 2019. USENIX Association.

[8] Nikolaj Bjørner and Leonardo de Moura. Proofs and refutations, and z3. In *Proc. 7th International Workshop on Implementation of Logics (IWIL), CEUR Workshop Proc*, volume 418, pages 123–132, 2008.

[9] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-Performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 917–934, Vancouver, BC, August 2017. USENIX Association.

[10] Cristiano Calcagno, Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Moving fast with software verification. In *NASA Formal Methods*, pages 3–11. Springer, 2015.

[11] Cilium. Cilium Service Mesh. https://cilium.io/use-cases/service-mesh, 2024. Accessed: 2024.

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[13] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, page 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 249–264, New York, NY, USA, 2024. Association for Computing Machinery.

[15] eBPF Development Team. ebpf. https://ebpf.io/, 2022.

[16] The eBPF Development Team. Bounded loops in bpf for the 5.3 kernel. https://lwn.net/Articles/794934/.

[17] The eBPF Development Team. bpf: add bpf_for_each_map_elem helper. https://lwn.net/Articles/846504/.

[18] The eBPF Development Team. bpf: add search pruning optimization and tests. https://lwn.net/Articles/614226/.

[19] The eBPF Development Team. ebpf verifier. https://docs.kernel.org/bpf/verifier.html.

[20] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. Ilinva: Using abduction to generate loop invariants. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2019.

[21] Envoy. Envoy is an open source edge and service proxy, designed for cloud-native applications. https://www.envoyproxy.io, 2024. Accessed: 2024.

[22] Yuyou Fan and John Regehr. High-throughput, formal-methods-assisted fuzzing for llvm. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 349–358, 2024.

[23] William Findlay, Anil Somayaji, and David Barrera. Bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 91–103, 2020.

[24] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[25] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, may 1996.

[26] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *PLDI*, pages 1069–1084, 2019.

[27] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-End security via automated Full-System verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[29] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *CoNEXT*, pages 54–66, 2018.

[30] International Organization for Standardization. *ISO/IEC 9899:2018: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, 2018.

[31] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[32] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. SLING: using dynamic analysis to infer program invariants in separation logic. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 788–801. ACM, 2019.

[33] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[35] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[36] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 229–243, New York, NY, USA, 1996. Association for Computing Machinery.

[37] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *SOSP*, 2019.

[38] Luke Nelson, Xi Wang, and Emina Torlak. A proof-carrying approach to building correct and flexible in-kernel verifiers. https://lpc.events/event/

11/contributions/944/attachments/893/1707/2021-09-23-lpc21.pdf.

[39] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 251–266. Springer, 2007.

[40] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[41] Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[42] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[43] Alastair Robertson. bpftrace. https://github.com/iovisor/bpftrace.

[44] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. *SIGPLAN Not.*, 39(9):201–212, sep 2004.

[45] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7762–7773, Red Hook, NY, USA, 2018. Curran Associates Inc.

[46] Calico Development Team. Calico. https://www.tigera.io/project-calico/.

[47] The Aqua Tracee Development Team. tracee. https://www.aquasec.com/products/tracee/.

[48] The Coq Development Team. Coq. https://coq.inria.fr/.

[49] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 245–257, 2021.

[50] Qiushi Wu, Zhongshu Gu, Hani Jamjoom, and Kangjie Lu. Gnnic: Finding long-lost sibling functions with abstract similarity. *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[51] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[52] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. Bpf for storage: an exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 128–135, 2021.

[53] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. Vst-a: A foundationally sound annotation verifier, 2023.

A Syntax of VEP-C Assertions

In this section, we discuss the syntax of VEP-C assertions, which are divided into two main categories: user assertions and internal assertions. User assertions enable developers to annotate C code with specific directives, whereas internal assertions support the tool's symbolic execution and reasoning processes.

A.1 Syntax of User Assertions

The primary design principle of user assertions is to align as closely as possible with the habits of C programmers, while also ensuring sufficient expressiveness for formal reasoning at the logical level. We begin by defining the expressions used in user assertions, followed by an explanation of how to construct a complete assertion. Lastly, we will discuss the structure of function specifications within this framework.

The expressions are inductively defined in Figure 16. The only thing unfamiliar is custom functions. They are functions in the *logic* world. For example, we provide a built-in function `rev`(l) that reverses a list (a list of values, not a data structure in C).

A user assertion can consist of several branches, each of which is divided into a section that describes pure facts and another that describes memory aspects. The syntax tree is defined in Figure 17. The semantics of user assertions is a little different from textbook separation logics, because of its HCI nature. To convince yourself that our transformation explained later makes sense, imagine we normalize every memory-related expression to a form $*l$. We assume two ls are different simply by their syntax. Then, each proposition—whether pure or spatial—describes such memory locations in addition to their spatial part, if any. When they are connected through conjunction `&&` or separating conjunction `*`, we union these locations; when connected through `||`, we keep them local to each case.

Just like custom functions, users can introduce custom predicates. For example, our built-in predicate `Ebpf_map`(m) denotes that an eBPF map is stored at a location m.

A function specification consists of three parts: logical variable list (**With** clause), precondition (**Require** clause), and postcondition (**Ensure** clause).

With clause. An implicit "∀" around the specification. They are sometimes essential to relate the program state at the function exit point to the one at the function entry. They can even appear in annotations inside the function so that developers have convenient access to the initial state. In general, their values should be "determinable" from the precondition.

Require clause. The precondition describes properties that the program state at function entry should satisfy.

Ensure clause. The postcondition describes properties that the program states when the function exits should satisfy. We use the keyword `__return` to refer to the return value, if any.

The syntax of function specifications is formally defined in Figure 18:

A.2 Syntax of Internal Assertions

We use a canonical form of assertions, internal assertions, actual symbolic execution, constraint solving, and everything. They have a simpler structure and clearer semantics (paradoxically, user-friendly assertions have *complex* semantics). Each internal assertion consists of several branches; each of which includes four parts: **Exist**, **Local**, **Prop**, and **Sep**. **Exist** is a list of existential variables, either explicitly written by developers or automatically generated during transformation (explained later). **Local** describes the relations between program variables and the logic world. For example, in C, `&`$v = e$ means that the address of the program variable v is expression e; in eBPF bytecode, $r = e$ means the register r stores value e. **Prop** stores pure facts. **Sep** stores memory-related facts. The syntax of internal assertions is defined in Figure 19.

We will use an example to clearly illustrate the distinction between user assertions and internal assertions. Consider the following annotation in a C program:

```
exists v, y = &x && x == 2 * v &&
          forall n, v != n * n
```

An equivalent internal assertion looks like this:

```
Ex    : v vx vy
Local : &x = px,  &y = py
Prop  : vy == px && vx == 2 * v &&
        forall n, n != n * n
Sep   : store(px, int, vx)
      * store(py, int, vy)
```

In this example, the user assertion describes an integer variable x, whose value is an even number that is not a perfect square; y is a pointer to x. The internal assertion introduces two new existential variables vx and vy to represent the current values of the program variables x

Expression	e	::=	$z \in \mathbb{Z}_{64}$	integer literal
		|	$v_p \mid v_l$	C variable, logic variable
		|	$\&e \mid *e$	address, dereference
		|	$\odot e \mid e_1 \oplus e_2$	unary and binary operation
		|	$e_1[e_2]$	array indexing
		|	$e.m \mid e\texttt{->}m$	member access
		|	$\texttt{sizeof}(t)$	type size
		|	$f(e_1, \ldots, e_n)$	custom function
Unary operator	$\odot$	::=	! | ~ | - | + | ...	
Binary operator	$\oplus$	::=	+ | - | * | / | ...	

Figure 16: Syntax of expressions.

Pure proposition	P	::=	$e_1 \oplus e_2$	comparison
		|	P_1 && P_2	conjunction
		|	P_1 || P_2	disjunction
		|	P_1 -> P_2	implication
		|	$!P$	negation
		|	exists x, P	existential quantification
		|	forall x, P	universal quantification
		|	$p(e_1, \ldots, e_n)$	custom predicate
Spatial proposition, assertion	S	::=	P	
		|	P && S	conjunction
		|	S_1 * S_2	separating conjunction
		|	S_1 || S_2	disjunction
		|	exists x, S	existential quantification
		|	$p(e_1, \ldots, e_n)$	custom predicate
Comparison operator	$\oplus$	::=	> | >= | < | <= | = | !=	

Figure 17: Syntax of user assertions.

Function specification F ::= $(\forall v_1, \ldots, v_n . S_{pre}, S_{post})$

Figure 18: Syntax of function specification.

Assertion	A	::=	H	singleton
		|	$A \vee H$	disjunction
Heap	H	::=	$\exists x_1, \ldots, x_n . L \wedge P \star S$	
Local	L	::=	$\bullet$	empty
		|	$\&v = e, L$	address
Pure proposition	P	::=	$e_1 \oplus e_2$	comparison
		|	$\neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \rightarrow P_2$	connective
		|	$\exists x.P \mid \forall x.P$	quantification
		|	$p(e_1, \ldots, e_n)$	custom predicate
Spatial proposition	S	::=	$\bullet$	empty heap
		|	$S_1 \star S_2$	separating conjunction
		|	$e_1 \mapsto e_2$ type t	store
		|	$e_1 \mapsto -$ type t	uninitialized store
		|	$p(e_1, \ldots, e_n)$	custom predicate
Expression	e	::=	$z \in \mathbb{Z}$	integer constant
		|	x	logic variable
		|	$\odot e \mid e_1 \oplus e_2$	arithmetic
		|	$\&e_1 \rightarrow m$	member offset
		|	$f(e_1, \ldots, e_n)$	custom function

Figure 19: (Abstract) Syntax of C internal assertions.

Normal form	N	::=	$\overrightarrow{H}$
Heap	G,H	::=	$\exists \overrightarrow{x}.P \,\&\&\, \overrightarrow{B}$
Body	B	::=	$p(e_1,\ldots,)$

Figure 20: Syntax of DNF.

and y, respectively. It is more structured and facilitates symbolic execution more conveniently.

B Transformation To Internal Assertion

In this section, we show how user assertions are transformed into internal assertions.

B.1 Step 1: Normalization

This phase puts an assertion in a sort of disjunctive normal form (DNF). Such a normal form is defined in Figure 20. The normalization procedure is described in Figure 21.

B.2 Step 2: Convert Expressions

In this phase, we convert expressions in user assertions to expressions in internal assertions, at the same time generating auxiliary **Local** and **Sep**. The procedure is defined in Figure 22. The judgement $\mathcal{V}(e,L,S,x) = (e',L',S',x')$ means that under context L,S,x, the value of e is e' and the context is updated to L',S',x'. $\mathcal{A}$ is similar, computing addresses (if valid) instead of values. The conversion is naturally extended to pure propositions $\mathcal{T}(P)$ and spatial propositions $\mathcal{T}(B)$, replacing each expression e in them with $\mathcal{V}(e)$, which we omit here.

B.3 Step 3: Sanity Check

Our pure propositions P can contain spatial expressions, so they are not really pure. Consequently, some user assertions cannot be transformed into internal ones. For example, the following user assertion is invalid.

$$\texttt{forall}\ i,\ 0 <= i \ \&\&\ i < 4 => a[i] == 0$$

A simple scope checking on internal assertions is enough to exclude such invalid cases. That is, every logic variable appearing in S should be bound by existential variables introduced in `exists` x,S.

C Full Evaulations

Full test results of previous work and VEP are shown in Table 4 and Table 5, respectively.

D Case Study : Key_connection

Figure 23 presents the complete Key_connection program, though only the assertions used to verify memory safety are shown here. In this program, `connt→conn_list` is a linked list of servers to connect to. To describe the memory structure related to this linked list, we introduce two predicates: `ConnListrep` and `ConnListseg`. These predicates are defined as follows:

```
ConnListrep(x) := x == 0 && emp ||
    ∃ n l num mark_v length,
      0 ⩽ n && n < 128 && l[n] == 0 &&
      x → num_packet == num &&
      x → mark == mark_v &&
      x → lengthsofar == length &&
      store_char_array(&(x→key), 128, l) *
      ConnListrep(x → next)

ConnListseg(x,y) := x == y && emp ||
    ∃ n l num mark_v length,
      0 ⩽ n && n < 128 && l[n] == 0 &&
      x → num_packet == num &&
      x → mark == mark_v &&
      x → lengthsofar == length &&
      store_char_array(&(x→key), 128, l) *
      ConnListseg(x → next, y)
```

In this definition, `store_char_array` corresponds to `chars` in Figure 4, representing a character array. This is purely a recursive definition describing the server list.

E Case Study : Acquire/Release example

Figure 24 presents an example that uses helper functions for acquiring and releasing resources. In this example, line 13 invokes the helper function `bpf_sk_lookup_udp` to obtain a socket. The success of this operation is checked at line 17, and if successful, the socket is released at line 19 using the helper function `bpf_sk_release`. As a result, this program maintains kernel-interface compliance level safety.

In this example, the user only needs to provide the annotations in lines 3 to 5 to successfully verify the program. VEP automatically generates the assertions in lines 13-15 and 19-20. Suppose the user omits the null check for sk at line 16 and directly returns. In that case, VEP will report an error due to the presence of an unverified

$$norm(P) = P \qquad norm(p(e_1,\ldots,e_n)) = p(e_1,\ldots,e_n) \qquad \frac{norm(S) = \exists\overrightarrow{x_1}.P_1 \,\&\&\, \overrightarrow{B_1},\ldots,\exists\overrightarrow{x_n}.P_n \,\&\&\, \overrightarrow{B_n}}{norm(\exists x,S) = \exists\overrightarrow{x,x_1}.P_1 \,\&\&\, \overrightarrow{B_1},\ldots,\exists\overrightarrow{x,x_n}.P_n \,\&\&\, \overrightarrow{B_n}}$$

$$\frac{norm(S) = \exists\overrightarrow{x_1}.P_1 \,\&\&\, \overrightarrow{B_1},\ldots,\exists\overrightarrow{x_n}.P_n \,\&\&\, \overrightarrow{B_n}}{norm(P \;\&\&\; S) = \exists\overrightarrow{x_1}.(P \,\&\&\, P_1) \,\&\&\, \overrightarrow{B_1},\ldots,\exists\overrightarrow{x_n}.(P \,\&\&\, P_n) \,\&\&\, \overrightarrow{B_n}} \qquad \frac{norm(S_1) = \overrightarrow{H_1} \qquad norm(S_2) = \overrightarrow{H_2}}{norm(S_1 \;||\; S_2) = \overrightarrow{H_1 H_2}}$$

$$\frac{norm(S_1) = H_1,\ldots,H_n \qquad norm(S_2) = G_1,\ldots,G_m}{norm(S_1 \;*\; S_2) = merge(H_1,G_1),\ldots,merge(H_1,G_m),\ldots,merge(H_n,G_m)}$$

$$merge(\exists\overrightarrow{x_1}.P_1 \,\&\&\, \overrightarrow{B_1}, \exists\overrightarrow{x_2}.P_2 \,\&\&\, \overrightarrow{B_2}) = \exists\overrightarrow{x_1 x_2}.(P_1 \,\&\&\, P_2) \,\&\&\, \overrightarrow{B_1 B_2}$$

Figure 21: Normalization procedure.

$$\mathcal{V}(z,C) = (z,C) \qquad \mathcal{V}(v_l,C) = (v_l,C) \qquad \mathcal{V}(\texttt{sizeof}(t),C) = (\mathsf{sizeof}(t),C) \qquad \mathcal{V}(\&e,C) = \mathcal{A}(e,C)$$

$$\frac{\mathcal{V}(e,C) = (e',C')}{\mathcal{V}(\odot e,C) = (\odot e',C')} \qquad \frac{\mathcal{V}(e_1,C) = (e_1',C') \qquad \mathcal{V}(e_2,C') = (e_2',C'')}{\mathcal{V}(e_1 \oplus e_2,C) = (e_1' \oplus e_2',C'')}$$

$$\frac{\mathcal{V}(e_1,C) = (e_1',C_1) \qquad \cdots \qquad \mathcal{V}(e_n,C_{n-1}) = (e_n',C_n)}{\mathcal{V}(f(e_1,\ldots,e_n),C) = (f(e_1',\ldots,e_n'),C_n)} \qquad \frac{\mathcal{A}(e,C) = (a,L,S,x) \qquad a \in \mathrm{dom}(S)}{\mathcal{V}(e,C) = (S(a),L,S,x)}$$

$$\frac{\mathcal{A}(e,C) = (a,L,S,x) \qquad a \notin \mathrm{dom}(S) \qquad v\ \mathsf{fresh}}{\mathcal{V}(e,C) = (v,L,S\{a \mapsto v\},x \cup \{v\})} \qquad \mathcal{A}(*e,C) = \mathcal{V}(e,C) \qquad \mathcal{A}(e_1\,[e_2\,],C) = \mathcal{V}(e_1{+}e_2,C)$$

$$\frac{\mathcal{A}(e,C) = (e',C')}{\mathcal{A}(e\,.\,m,C) = (e' + \mathsf{offset}(m),C')} \qquad \frac{\mathcal{V}(e,C) = (e',C')}{\mathcal{A}(e{-}{>}m,C) = (e' + \mathsf{offset}(m),C')} \qquad \frac{v_p \in \mathrm{dom}(L)}{\mathcal{A}(v_p,L,S,x) = (L(v_p),L,S,x)}$$

$$\frac{v_p \notin \mathrm{dom}(L) \qquad a\ \mathsf{fresh}}{\mathcal{A}(v_p,L,S,x) = (a,L\{v_p \mapsto a\},S,x \cup \{a\})}$$

$$\frac{\mathcal{T}(P,\{\},\{\},\{\}) = (P',C) \qquad \mathcal{T}(\overrightarrow{B},C) = (\overrightarrow{B'},L,S,y)}{\mathcal{T}(\exists\overrightarrow{x}.P \,\&\&\, \overrightarrow{B}) = \exists x_1,\ldots,x_n,y_1,\ldots,y_m.L \wedge P' \ast_i B_i' \ast_{a \in \mathrm{dom}(S)} a \mapsto S(a)} \qquad \mathcal{T}(\overrightarrow{H}) = \bigvee \overrightarrow{\mathcal{T}(H)}$$

Figure 22: Expression conversion.

```
#include "bpf.h"

struct connection{
    char key[128];
    unsigned int num_packet;
    unsigned int mark;
    unsigned int lengthsofar;
    struct connection * next;
};

struct identifier{
    int mark;
    char  name[32];
    char  pattern[512];
};

struct conntrack{
    int queuenum;
    int iden_num;
    struct identifier *iden_array;
    struct connection *conn_list;
};

struct connection * get_connection
    (struct conntrack * connt, char * key)
/*@ With n0 m0 l0 Connlist
    Require 0 ⩽ n0 && l0[n0] == 0 && n0 < m0 &&
        connt ⟶ conn_list == Connlist &&
        store_char_array(key, m0, l0) *
        ConnListrep(Connlist)
    Ensure ∃ v, __return == v && TT
*/
{
    struct connection * p ;
    if(connt == NULL || key == NULL)
        return NULL;
    p = connt⟶conn_list;
    /*@ Inv
        0 ⩽ n0 && l0[n0] == 0 && n0 < m0 &&
        connt ⟶ conn_list == Connlist &&
        store_char_array(key, m0, l0) *
        ConnListrep(p) * ConnListseg(Connlist, p)
    */
    while(p != NULL){
        if(strcmp(p⟶key , key) == 0)
            break;
        p = p⟶next;
    }
    if(p == NULL)
        return NULL;
    else
        return p;
}
```

Figure 23: Annotated Key_connection

memory region `store_sock(sk)`, whose validity is uncertain. Similarly, if the user fails to release sk at line 18, VEP will also raise an error at the final return statement.

```
SEC("xdp")
int prog(struct xdp_md *ctx)
/*@ Require store_xdp(ctx)
    Ensure ∃ v, __return == v && store_xdp(ctx)
*/
{
  struct bpf_sock_tuple tup;
  struct bpf_sock *sk;
  if (!check_ipv4_udp(ctx)) return XDP_DROP;
  init_sock_tuple(ctx, &tup);
  sk = bpf_sk_lookup_udp(ctx, &tup,
        sizeof(tup.ipv4), 0, 0);
  /*[auto generated]
    store_xdp(ctx) * store_sock_tuple(tup) *
        store_sock(sk) */
  if (!sk) return XDP_DROP;

  bpf_sk_release(sk);
  /*[auto generated]
    store_xdp(ctx) * store_sock_tuple(tup) */
  return XDP_PASS;
}
```

Figure 24: An example for Acquire/Release by helper function

Programs		Code Lines	Linux verifier		PREVAIL	
			Time(ms)	Memory(KB)	Time(ms)	Memory(KB)
Linux Samples	sockex1_kern	29	1.03	4194	2.05	4978
	syscall_tp_kern(enter)	38	1.03	4194	3.17	4931
	cpustat_kern(frequency)	93	1.13	4190	11.30	6377
	cpustat_kern(idle)	116	1.11	4196	23.52	7918
	xdp_adjust_tail_kern	47	0.64	4196	6.69	5539
	syscall_tp_kern(exit)	35	0.99	4190	3.02	4927
	lathist_kern(on)	77	1.09	4144	10.18	6025
	trace_event_kern	65	fail	-	48.37	6730
	tcp_iw_kern	68	0.65	4152	9.82	5664
	tcp_rwnd_kern	50	0.77	4155	6.69	5404
PREVAIL Samples	twomaps	34	fail	-	2.43	5056
	twotypes	33	fail	-	3.80	5278
	map_in_map	36	fail	-	5.17	5054
	stackok	13	1.02	4114	74.67	5279
	loop	21	fail	-	fail	-
	packet_start_ok	14	1.08	4200	1.19	4942
	twostackvars	47	0.53	4140	20.9	5267
	packet_access	28	0.58	4153	2.74	5216
	bpf2bpf	13	0.51	4154	0.16	4157
	dependent_read	13	0.55	4154	fail	-
Stringlib	strcpy	34	fail	-	fail	-
	strncpy	34	1.65	4212	fail	-
	strcat	44	fail	-	fail	-
	strncat	43	fail	-	fail	-
	strlen	19	fail	-	fail	-
	strncmp	31	2.69	4316	fail	-
	strcmp	32	fail	-	fail	-
	memset	28	1.14	5168	39.89	7267
	strchr	28	fail	-	fail	-
	memchr	28	fail	-	fail	-
Unsafe Program	badhelpercall	6	reject	-	reject	-
	badmapptr	24	reject	-	reject	-
	badrelo	20	reject	-	reject	-
	ctxoffset	21	reject	-	reject	-
	nullmapref	23	reject	-	reject	-
	badhelpercall2	22	reject	-	reject	-
	packet_overflow	14	reject	-	reject	-
	wronghelper	20	reject	-	reject	-
	mapunderflow	23	reject	-	reject	-
	packet_reallocate	22	reject	-	reject	-
Key_connection		63	fail	-	fail	-

Table 4: Time cost and memory usage of samples by Linux verifier and PREVAIL

Programs		Assertion Lines	Proof Lines	VEP-C		VEP-compiler	VEP-eBPF	
				Time(ms)	Memory(KB)	Time(ms)	Time(ms)	Memory(KB)
Linux Samples	sockex1_kern	3	1140	4.06	5882	0.35	2.53	2284
	syscall_tp_kern(enter)	7	2253	5.33	6391	0.37	2.57	2396
	cpustat_kern(frequency)	11	8357	50.33	15887	2.62	7.89	4292
	cpustat_kern(idle)	11	19390	158.12	32569	6.09	21.32	7167
	xdp_adjust_tail_kern	10	739	4.27	5852	0.32	2.41	2236
	syscall_tp_kern(exit)	7	2253	5.28	6396	0.35	2.47	2379
	lathist_kern(on)	9	4180	23.47	18502	2.47	21.69	8034
	trace_event_kern	6	7136	67.33	18841	1.15	5.79	3353
	tcp_iw_kern	6	161	12.57	12150	1.75	9.11	4182
	tcp_rwnd_kern	6	19231	63.88	19154	1.84	8.44	4342
PREVAIL Samples	twomaps	2	2310	6.07	7119	0.46	2.77	2521
	twotypes	4	9449	15.49	8288	0.73	4.40	2665
	map_in_map	3	261	2.97	5995	0.39	2.31	2348
	stackok	4	1848	4.32	5223	0.19	1.96	2028
	loop	8	4042	10.02	7682	0.55	2.84	2453
	packet_start_ok	3	1245	2.64	5174	0.21	2.03	2126
	twostackvars	12	18446	40.23	12422	1.43	3.96	3047
	packet_access	3	3669	8.85	7075	0.50	3.42	2659
	bpf2bpf	7	13	0.70	4436	0.11	1.76	1932
	dependent_read	3	648	2.56	4920	0.19	2.14	2106
Stringlib	strcpy	9	6051	12.09	8172	0.66	2.67	2510
	strncpy	11	4242	8.69	7888	0.59	2.61	2557
	strcat	17	12820	31.17	11778	1.39	3.41	2907
	strncat	17	14254	36.60	12612	1.16	3.54	2970
	strlen	9	2035	5.02	5946	0.29	2.23	2236
	strncmp	11	3976	8.45	7994	0.54	2.58	2470
	strcmp	9	5934	14.27	8528	0.76	2.25	2574
	memset	11	1925	3.78	6271	0.35	2.18	2281
	strchr	9	2329	6.94	6962	0.40	1.66	2463
	memchr	9	2551	7.64	6683	0.42	3.19	2517
Unsafe Program	badhelpercall	4	reject	2.82	1618	-	-	-
	badmapptr	3	reject	3.05	1701	-	-	-
	badrelo	3	reject	2.60	1575	-	-	-
	ctxoffset	3	reject	2.70	1436	-	-	-
	nullmapref	3	reject	4.38	2275	-	-	-
	badhelpercall2	4	reject	2.46	1402	-	-	-
	packet_overflow	3	reject	4.31	2115	-	-	-
	wronghelper	3	reject	2.73	1652	-	-	-
	mapunderflow	3	reject	2.78	1781	-	-	-
	packet_reallocate	3	reject	5.36	2955	-	-	-
Key_connection		17	5819	16.24	8534	0.56	2.48	2440

Table 5: Time cost and memory usage of samples by VEP

MeshTest: End-to-End Testing for Service Mesh Traffic Management

Naiqian Zheng Tianshuo Qiao Xuanzhe Liu Xin Jin
School of Computer Science, Peking University

Abstract

We present MeshTest, the first end-to-end testing framework for traffic management of service mesh. The key idea of MeshTest is to automatically generate input configurations with end-to-end semantics, and then create real test request suites on each input. There are two technical challenges. First, the input space of service mesh configurations is large and complex. The input configurations should be carefully orchestrated to form end-to-end service flow paths. Second, the abstract output network behavior cannot be directly checked for correctness, and we need to generate a set of real requests that are capable of checking possible behaviors. To address these challenges, we model the service flows of traffic management in service mesh, and propose a novel *Service Flow Exploration* technique to enumerate all possible configuration resources and interactions between them in the input configuration. We design and implement MeshTest, which contains an automatic input configuration generator based on *Service Flow Exploration* and a *Service Mesh Oracle* which leverages formal methods to generate test request suites. MeshTest has found 23 *new* bugs (19 confirmed and 10 fixed) in two popular service mesh systems, Istio and Linkerd.

1 Introduction

Cloud computing systems are growing with larger scale and more complex functionalities. Many cloud computing systems choose to be architected in a microservice-alike [1] manner, with each collection of microservices performing some discrete business functions. Service mesh systems are designed to process network requests and manage the communication between microservices. The widely-used open-source and commercial service mesh systems, such as Istio [2], Linkerd [3], Canal Mesh [4] and AWS App Mesh [5], provide rich traffic management functionalities for service traffic. Traffic management decides how service requests are transmitted through the service mesh system, and establishes the foundation for other functionalities. It provides declarative API for developers to define service routing, load balancing, A/B testing [6], and access control with just a few lines of configurations. Service meshes have been widely-used in industry [7], and there are also academic proposals such as application-defined networks [8, 9] to improve performance and simplify management of service meshes.

The rapid evolution and increasingly popular use of service mesh systems have raised higher demands on their reliability. Reliable traffic management is critical in service meshes, as it directly controls the service traffic and thus affects the correctness and performance of the entire application. The complexity in both implementation and functionality of service mesh traffic management makes it difficult to systematically test or verify. For example, Istio [2], one of the most widely-used service mesh systems, contains over 1,000 components and 3×10^5 lines of code. It provides more than 10 custom resources with extensive configuration options to define traffic management policies. Each custom resource is able to define a specific network function, and an actual service mesh configuration contains multiple custom resources. These resources are orchestrated to describe full end-to-end network functions, including traffic entrance, service routing, and workload dispatching (shown in Figure 1). The interactions between resources create more complex functionalities, such as priority competition on resources with the same matching conditions.

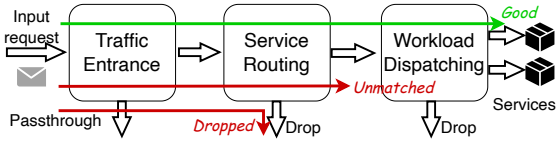


Figure 1: Network request processing stages in service mesh traffic management.

Figure 2 shows a real bug that our technique detects in Istio [10]. When two `service entries` (a custom resource in Istio to define service entrance rules) are defined with the same hostname but different ports and workloads, one of them will be unreachable. This bug is caused by the incorrect implementation in service entrance, which fails to merge the two `service entries` correctly and makes the service listening on port 9080 unreachable. Similar to this bug, traffic management bugs lead to severe issues, such as service unavailability and incorrect service routing, which subsequently degrade the entire application. These bugs escape from existing testing suites because existing service mesh systems rely most on unit tests which can not check traffic management correctness end to end. Some systems include a few end-to-end tests, but they only check basic functionalities and do not cover complex resource interactions.

Creating end-to-end tests for traffic management of service mesh is not trivial, since the test input should be carefully orchestrated to form service flow paths from the sources to the destinations. Figure 1 shows three stages of request processing in the traffic management of service mesh systems. Each stage receives and forwards requests based on the fields and

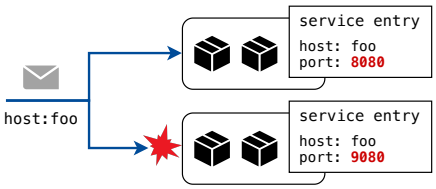


Figure 2: A bug [10] discovered by MeshTest. When two `service entries` are defined on the same host but different ports and workloads, one of them will be unreachable.

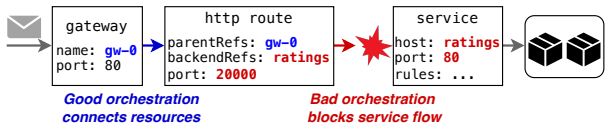


Figure 3: An example configuration snippet of Linkerd [3]. It demonstrates that resource orchestration is necessary for end-to-end test cases; otherwise, it would break the service flow and make the cases not end-to-end.

metadata of the request (*e.g.*, hostname, destination). An end-to-end test case should build up service flow paths through these stages, while a basic test input may drop the request in some stages or fail to hand over between stages. To orchestrate resources and forward the service flow from one resource to another, developers need to set up connectors fields in the entry and exit of each resource to specify what kind of service flow to accept and where to forward. Figure 3 shows a good and a bad example of service flow forwarding between different resources based on a configuration snippet of Linkerd [3]. In the good case (highlighted in blue), the `gateway` hands over the service request to the `http route` via the matching connector fields. In the bad case (highlighted in red), the `http route` fails to forward the request to the service due to a mismatch in port, and blocks the service flow, causing the test case to be not end-to-end. Notably, the real end-to-end test cases can be much more complex than this example, as the types of resources and their connections are more diverse, allowing service flow splitting and merging in various ways.

We seek for a systematic approach to automatically perform end-to-end testing for traffic management of service mesh. Existing techniques like symbolic execution [11] and fuzzing [12] do not model service flows, and cannot be directly used to generate end-to-end input configurations for service mesh systems because of state explosion and low validation pass rates. Besides, the automatic testing tool requires an automatic testing oracle to check the correctness on each input configuration, which existing tools cannot achieve.

Our solution. In this paper, we present MeshTest, the first automatic end-to-end testing framework for traffic management of service mesh. MeshTest automatically generates end-to-end input configurations based on an explicit model on service flows and a novel *Service Flow Exploration* technique. MeshTest's *Service Mesh Oracle* checks whether the service mesh correctly implements functionalities specified in the input configuration by sending real network requests. MeshTest is highly usable. It does not require hypotheses about vulnerable regions in the code nor implementation details. MeshTest is not tied to any specific service mesh system, and can be retrofitted among different systems by its generic API.

MeshTest is a service flow centric approach. It gradually models service flow at a finer granularity on a smaller scale in four stages: service flow exploration, service flow filling, fine-grained service flow modeling and service flow execution. The first two stages compose the input generator, which automatically creates end-to-end input configurations. The last two stages compose the *Service Mesh Oracle*, which automatically generates real network requests to check whether the service mesh system behaves correctly under the input configuration. These stages are designed to be loose-coupled so that they can work individually and can be extended to other tools.

The key insight of MeshTest's input generator is that in an end-to-end input configuration, the resources are orchestrated by end-to-end service flows. Specifically, MeshTest models the input space of end-to-end configurations as *Service Flow Skeletons* (representing interactions among configuration resources) and *Service Flow Bodies* (representing the detailed rules inside each configuration resource). The MeshTest input generator firstly conducts a domain-specific *Service Flow Exploration* technique to create service flow skeletons (§ 4.1), which depict end-to-end layouts of resource orchestration. *Service Flow Exploration* additionally objectives testing vulnerable resource interactions, comprehensively covering all possible resource interactions in the service flow skeleton suites. Then, MeshTest fills in the service flow bodies based on the skeleton with fuzzing-based domain-specific techniques to generate ready-to-use end-to-end input configurations with various network configuration options (§ 4.2).

The *Service Mesh Oracle* in MeshTest is a powerful testing oracle to check whether the service mesh system correctly implements the input configuration. The key idea of *Service Mesh Oracle* is to select a comprehensive set of requests to represent possible service traffic based on a domain specific model, and then check whether the results are as expected. To do so, *Service Mesh Oracle* first supplements the service flow model with more fine-grained and accurate rules (§ 4.3), and transforms the input configuration into a control flow graph (CFG) which is capable of inferring the correct effects on arbitrary requests. Then, *Service Mesh Oracle* conducts symbolic execution on the control flow graph, and generates a set of test requests where each request represents a unique path in the CFG, thereby representing a specific behavior of the input configuration (§ 4.4). These requests are then sent to and captured from the service mesh system hosted in a testbed to check whether the system behaves as expected under the input configuration. Besides, *Service Mesh Oracle* also checks negative cases, such as rejecting invalid inputs, and inspects whether the system raises exceptions.

Key results. We implement MeshTest for service mesh systems. MeshTest requires no expert knowledge on the implementation code base. MeshTest's input generator can effectively enumerate possible layouts of configuration resources orchestrated by service flows, and generate corresponding end-to-end input configurations. MeshTest's *Service Mesh Oracle* can automatically generate a real network request suite which is comprehensive to represent arbitrary service traffic and check behaviors of the service mesh system under the input configuration. We evaluate MeshTest on two most widely-used service mesh systems, Istio [2] and Linkerd [3]. To date, MeshTest has detected 23 *new* bugs that have not been reported previously. 19 out of these 23 bugs have been already confirmed and 10 have been fixed by developers. Notably, a large number of these bugs occur due to interactions between resources, leading to deep semantic violations that are difficult to detect using existing approaches. MeshTest can generate 2500 end-to-end input configurations per second, and check correctness with an average of 29 varying network requests for each input configuration.

Contributions. The paper makes three main contributions.

- We propose a service flow centric approach to model the end-to-end semantics of traffic management of service mesh in gradually finer granularity, and design a novel *Service Flow Exploration* technique to create input configurations with end-to-end service flows.
- We design and implement MeshTest, the first automatic end-to-end testing technique for traffic management of service mesh. MeshTest consists of an input generator empowered by *Service Flow Exploration* and a *Service Mesh Oracle* based on formal methods.
- MeshTest has already improved the quality of two most widely-used service mesh systems by finding 23 *new* bugs. The code of MeshTest is open-source and publicly available `https://github.com/pkusys/meshtest`.

2 Background and Motivation

2.1 Service Mesh Systems

Traffic management of service mesh. Service mesh systems provide a set of network functions to manage and control network traffic in a microservice architecture. Traffic management is the core and most important function, which includes fine-grained service routing, load balancing, and advanced traffic control such as A/B testing. The input of the service mesh is a high-level configuration file that describes the desired network behaviors with custom resources, such as service routing rules and workload discovery. The configuration file (usually written in YAML or JSON) is declarative and highly-abstracted, *i.e.* it only specifies the desired network behaviors, but not how to implement them. The main goal and challenge of service meshes is to correctly implement the abstract network behavior specifications into concrete request processing logics on each nodes.

Interactions between input configuration resources. Although each configuration resource specifies an individual network function, the end-to-end network traffic transmission requires orchestrated interactions between these resources to build end-to-end service flows. The resources define different network functions in different request processing stages, and interact with other resources via connector fields. The blue resource orchestration in Figure 3 allows the service traffic to be forwarded from traffic entrance rules to service routing rules. In contrast, the red resource interaction blocks the service flow, and drops the service request before reaching the destination workload. Besides the orchestration, competition and conflicts also appear between configuration resources. For example, two resources on routing can share the same host key but conflicting routing behaviors. In this case, the service mesh system has to resolve the priority competition and arrange the correct routing rule for arbitrary service traffic. This problem orients from the declarative nature of the input configuration, which makes resource interactions implicit.

Abstract service mesh output. The goal of service mesh is to correctly implement desired network functions specified in the input configuration for arbitrary service traffic. The output of service mesh is the abstract traffic processing behaviors in the system, which are not directly observable or measurable. Judging the correctness of the output has to rely on sending and capturing real network requests to infer the abstract traffic processing behaviors. Thus, a reliable testing oracle has to select a comprehensive network request suite which is limited in size but capable of representing various service traffic in the service mesh system. In particular, it should cover all possible individual network functions and their complicated interactions specified in the input configuration.

2.2 Testing Service Mesh Systems

The lack of existing end-to-end tests. Most popular service mesh systems already have a mature ecosystem, including unit tests and continuous integrations, but they usually have only few end-to-end tests. For example, Istio [2] has 10891 unit tests and has already been widely-used in production. However, it only has 168 end-to-end tests and 217 assertions. Most of these end-to-end tests are simple and the assertions are basic. They do not reason about complex network behaviors caused by the interactions of resources. Another popular service mesh system, Linkerd [3], also has only 31 simple end-to-end tests. The lack of high-quality end-to-end tests is a common problem in traffic management of service mesh systems, mainly due to the difficulties of creating test inputs with end-to-end semantics and the challenges of creating a strong testing oracle. On one hand, the end-to-end service flow path costs a lot of manual work to generate configuration resources and connect them with proper connector fields to form reasonable service flow paths. On the other hand, the extensive functionality combinations raise the complexity of

determining the vulnerable input patterns and code regions. Thus, the lack of end-to-end tests reveals an urgent need to develop a comprehensive end-to-end testing framework for traffic management of service mesh, more specifically, to create more complex end-to-end input configurations and stronger correctness checkers.

Difficulties on applying existing techniques. Existing testing techniques, such as fuzzing [12, 13], symbolic execution [11], cannot be directly applied to service mesh systems. The main reason is that the input configuration is composed of a set of configuration resources, which are very large and have to be carefully orchestrated to form end-to-end service flow paths. Symbolic execution suffers from the path explosion problem, since each resource has a large number of detailed rule options and the combination of these resources explodes the input space exponentially. Specifically, `virtual service` includes more than one million of different configurations, and the number explodes when combined with other resources. Fuzzing does not model the service flows, so the generated resources are usually isolated and do not form end-to-end service flow paths. Another solution is to fuzz individual resources and then combine them by setting connector fields. On one hand, it is not easy to decide what resources are generated and how to connect them. On the other hand, the modified resources may not pass the configuration validation for violating constraints.

Recent works on reliability of Kubernetes controllers [14] and operators [15] cannot be directly applied to service mesh. The main reason is that these works rely on the state-centric property of Kubernetes, while service mesh traffic management is service flow centric. Specifically, state reconciliation is not the main concern in service mesh, since the network policies are completely recalculated on configuration changes. Besides, the input generation technique proposed by Acto [15] is not suitable for traffic management of service mesh, since many of the core resources in service mesh systems do not have a direct mapping to Kubernetes resources.

2.3 Challenges

Our goal is to develop a comprehensive end-to-end testing framework for service mesh traffic management. It should (1) automatically generate orchestrated input configurations with end-to-end service flow paths and (2) automatically checks the correctness of the abstract network behaviors by sending and checking real network requests. We identify two main challenges:

Challenge 1: End-to-end input generation. The first challenge is to create input configurations that have end-to-end semantics. More specifically, the resources in the input configuration have to be orchestrated and carefully connected to form complete service flow paths. Additionally, the input generator should be able to handle complex interactions and interleaving between resources, and strive to create complex combinations of configuration resources, such as service flow priority competition.

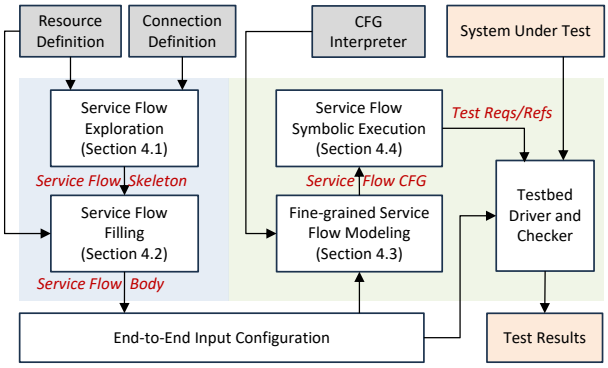


Figure 4: The workflow of MeshTest. The manual parts are gray, the automatic parts are white, and the system input and output are orange.

Challenge 2: Oracle for complex end-to-end inputs. The second challenge is to create a strong testing oracle that can judge whether the service mesh system correctly implements the input network configuration. The testing oracle has to create a limited but comprehensive network request suite which is able to represent various service traffic in the service mesh system. It also has to infer the expected output of arbitrary input requests as a reference to compare with the actual output.

3 Overview

MeshTest centers on service flows, which indicate how the service traffic flows through network functions specified in configuration resources. Specifically, MeshTest explicitly models the service flow in different granularity levels, in order to balance the need of covering larger input space and judging the correctness of service mesh behaviors. First, MeshTest's input generator models the input space of end-to-end configurations as service flow skeletons and service flow bodies, respectively representing the high-level combination layouts and detailed rules of resources. Second, MeshTest's oracle generates a comprehensive set of real requests based on a precise model of the service flow, and checks the correctness of the target system.

3.1 Workflow

Figure 4 shows the full workflow of MeshTest. MeshTest gradually creates and completes test cases in four stages: (1) it generates service flow skeletons by leveraging a domain specific *Service Flow Exploration* technique, indicating the high-level layout of service flows; (2) it fills service flow bodies with detailed rules and options and creates ready-to-use input configurations; (3) it automatically models the input configuration into a fine-grained service flow CFG; and (4) it checks the service mesh system with real requests generated. The first two stages focus on the input generation, and the last two stages focus on the output checking.

Through the four stages, the service flow modeling is gradually refined in finer granularity, and corresponds to a smaller scope. Each service flow skeleton specifies a high-level combination layout of resources, but does not contain detailed rules inside resources. Each service flow body (and its equivalent input configuration) is one of the possible fuzzing results of the service flow skeleton, and fills detailed rules inside resources. Each service flow CFG is a formal model for one input configuration, and contains precise priority selection and any other implicit effects of configuration. Each test request corresponds to a specific service flow path in the CFG, and is used to check a detailed effect. The less detailed modeling in MeshTest's input generator focuses on service flow layouts and reduces the complexity of input space without loss of generality. The fine-grained modeling in *Service Mesh Oracle* provides better correctness checking ability and covers more subtle explicit and implicit effects.

Usage. To use MeshTest, one needs to provide (1) the definition of the configuration resources used in the service flow, including constraints and the range of values, (2) connection rules between different resources, such as possible predecessor and successor resource types, and (3) an interpreter to automatically convert the input configuration to its equivalent service flow CFG, which includes system-specific detailed rules such as priority selection and implicit routes. On this basis, MeshTest automatically generates ready-to-use input configurations, creates a comprehensive service request suite, and applies the test on a testbed environment. Figure 4 illustrates the various components, system inputs, and outputs of MeshTest, where the manual parts are indicated in gray and the automatic parts are shown in white. The definition of the configuration resources and connection rules can be directly obtained from the documentations of the service mesh system. The interpreter requires users to utilize system-specific knowledge to develop and provide to MeshTest, thus requiring some manual efforts when updating the target service mesh or applying MeshTest to a new service mesh system. MeshTest provides a user-friendly interface, facilitating generalization across various service mesh implementations. These interfaces include configuration template definition and connection rule matrix for the input generator, and domain specific primitives used in CFG and symbolic execution for the testing oracle. In our practice, adapting MeshTest to another service mesh system costs less than two person-weeks. After the adaptation, MeshTest can work automatically and continuously until the input constraints or rules change.

3.2 Technique

Collecting service flow skeletons. The goal of this stage is to generate service flow skeletons which depict a high-level layout of service flows. We model a service flow skeleton as a directed acyclic graph (DAG). Each node represents a configuration resource, and each edge represents a connection between two resources. To ensure the end-to-end service flow, the service flow skeleton contains end-to-end service flow paths for all internal resources, making it ready for end-to-end testing. To do so, we leverage a domain specific *Service Flow Exploration* technique to systematically build service flow skeletons with end-to-end service flows. The *Service Flow Exploration* starts from a skeleton seed with resource interactions, and gradually connects more resources on the predecessor side and successor side. In this way, the skeleton seed is extended to larger size, and eventually connects the start and the end points to form a complete end-to-end service flow skeleton. Additionally, we heuristically explore all possible interactions of three typical types for any two resources to be the skeleton seed, in order to check the service meshes on more service flow orchestrations.

Filling service flow bodies. The goal of this stage is to fill detailed rules and options for each resources in the service flow skeleton, and then create ready-to-use input configurations. The service flow body describes (1) the detailed rules and options for each resource in the service flow skeleton, and (2) which pair of fields connects the resources. The main challenge of this stage is how to realize the connectivity defined in the service flow skeleton into real field values. To do so, MeshTest leverages a two-stage approach in which the first stage fills the connector fields to ensure connectivity, and the second stage extends the configuration with more detailed rules and options with fuzzing. As shown in the second part of Figure 4, one abstract service flow skeleton is filled with various detailed options, and becomes a set of corresponding input configurations. The configurations realize service flows designed in the skeleton, and cover more options on non-connector fields.

Fine-grained service flow modeling. The goal of this stage is to build a precise fine-grained model for each input configuration. The model describes the expected behaviors of arbitrary service request under the input configuration. The model depicts the fine-grained control flow of processing the service requests in the service mesh, and considers all explicit and implicit effects of the input configuration. Compared to the service flow skeleton and body, the fine granularity is mainly reflected in considering the complete role of resources, the priority competition between resources, and other implicit intrinsic rules in the service mesh. The design of the structure of the service flow CFG is based on the intrinsic three-stage structure of the service processing model (traffic entrance, service routing, and workload dispatching). We embed each resource's effects to these three stages, and resolve the priority competition and implicit routes carefully.

Checking by real requests. The goal of this stage is to check if the service mesh system correctly realizes the input configuration by sending and checking real service requests. The service request suite is a comprehensive set of real service requests, which is capable to represent arbitrary possible request. Specifically, it should cover all individual and combined rules

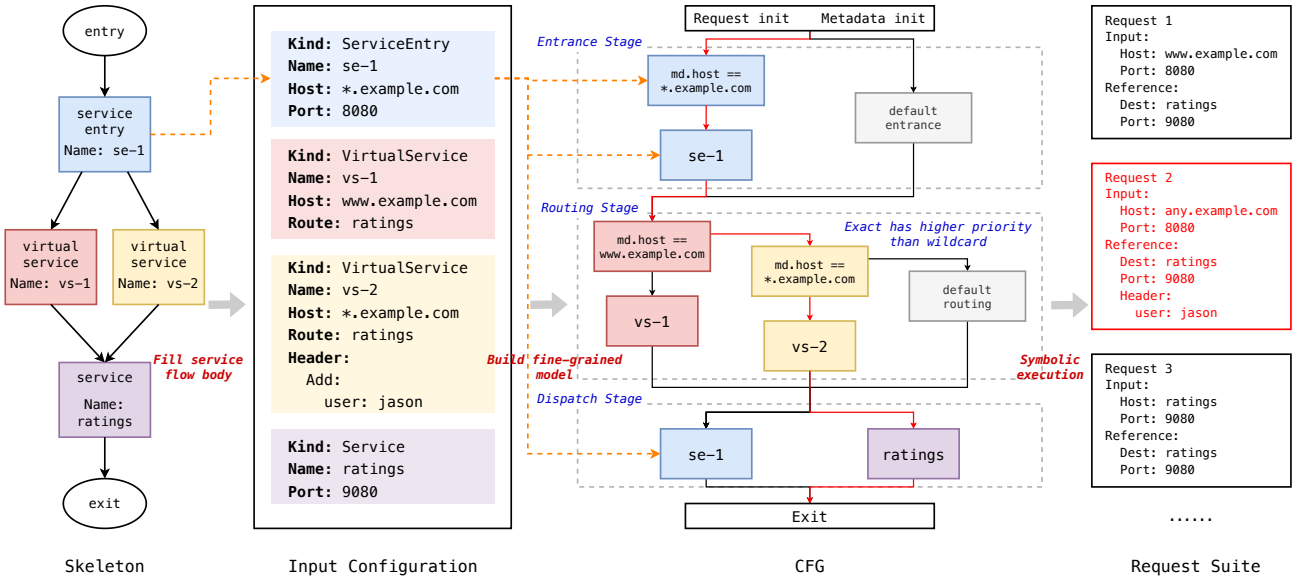


Figure 5: An illustrative example on the creation and evolvement of test cases for Istio [2]. The process starts from a service flow skeleton, evolves to a detailed input configuration, models the full effects to a CFG, and finally composes a real test request suite.

specified in the input configuration, and infer the expected outputs of those requests. To get a comprehensive test request suite, MeshTest leverages symbolic execution and traverses all possible paths on the service flow CFG. Each service flow path in the CFG corresponds to a specific test request, and further corresponds to a specific rule in the input configuration. After that, these test requests are sent to and captured from the testbed environment, and compared with the expected results.

4 Design and Implementation

This section explains the design of MeshTest and how we implement it. Figure 5 shows an illustrative example of the whole test campaign. It shows how a test case is created and evolved in the testing workflow. Specifically, we explain how MeshTest explores the resource combinations to orchestrate end-to-end service flow skeletons(§ 4.1), fills all fields to compose ready-to-use end-to-end input configurations (§ 4.2), models full fine-grained effects of the configuration (§ 4.3), and tests the target system with real service requests (§ 4.4).

4.1 Service Flow Exploration

Generating end-to-end service flows to guide the orchestration of resources is the first step of the workflow and the key to end-to-end test input generation. As shown in the first part of Figure 5, the service flow skeleton depicts how the service flow passes through resources. Further, it serves as a backbone indicating the resource orchestration structure. Service flow skeletons are high-level and not ready for system input, since they do not contain detailed rules and options. To find all possible service flow structures, MeshTest models and then explores the service flow skeletons with a domain specific *Service Flow Exploration* technique.

Service flow model. To get a formal definition, we model the resources as nodes, and the service flow between resources as directed edges. Further, we model the service flow skeleton as a directed acyclic graph (DAG) $G = (V, E)$. Each resource is of one resource type $R(v) \in \{R_0, R_1, \cdots, R_n\}$. The connectability between two kinds of resources is modeled by an adjacency matrix $\mathcal{A} = [A_{ij}]$. The acyclic property of the graph is intrinsic in the service mesh traffic management, because any request has to be processed in bounded steps. We model the end-to-end service flow path as a sequence of resources connected by edges, which starts from the start point and ends at the end point. We define that the service flow skeleton does not contain any dangling resources, *i.e.* each resource is covered by at least one end-to-end service flow path.

Generating service flow skeletons. The goal of *Service Flow Exploration* is to generate end-to-end service flow skeletons which depict the layouts of resource orchestration. The key insight of *Service Flow Exploration* is to first explore resource interactions as skeleton seeds and then explore the end-to-end service flow paths traversing the seeds to compose end-to-end service flow skeletons. The skeleton seed describes the resource interactions used in the service flow skeleton. Other resources are orchestrated to establish end-to-end service flow paths that traverse through the skeleton seed, thereby enabling the composition of complete service flow skeletons. The inputs of *Service Flow Exploration* are the types of resources $\mathcal{R}$ and the adjacent matrix $\mathcal{A}$ describing the connectability of resources. They can be inferred from the documentations and specifications of traffic management of service mesh. The output is a set of end-to-end service flow skeletons showing orchestrations and interactions of resources.

Algorithm 1 *Service Flow Exploration*

Inputs:
- $\mathcal{R} = \{R_0, \cdots, R_n\}$: resources for traffic management of service mesh
- $\mathcal{A} = [A_{ij}]$: adjacency matrix for resource connection compatibility

Outputs:
- $\mathcal{S}keleton$: end-to-end service flow skeletons

```
1:  function ExploreSkeletonSeeds(R, A)
2:      // Enumerate resource interactions as seeds
3:      seeds ← ∅
4:      for res1, res2 ∈ R do
5:          if A_{res1,res2} then
6:              seeds.add(Connect(res1, res2))
7:          if ∃res3 s.t. A_{res3,res1} ∩ A_{res3,res2} then
8:              seeds.add(Split(res3, res1, res2))
9:          if ∃res3 s.t. A_{res1,res3} ∩ A_{res2,res3} then
10:             seeds.add(Merge(res3, res1, res2))
11:     return seeds
12: function ServiceFlowExploration(R, A)
13:     skeletonSeeds ← ExploreSkeletonSeeds(R, A)
14:     Skeleton ← ∅
15:     for seed ∈ skeletonSeeds do
16:         partialSkeleton ← seed
17:         seedResourceSet ← seed.getResourceSet()
18:         queue ← Queue(seedResourceSet) // init with seed resources
19:         while queue ≠ ∅ do
20:             res ← queue.pop()
21:             if res ≠ R_entry and res has no predecessor then
22:                 pred ← GetPossiblePred(res, R, A)
23:                 partialSkeleton.connect(pred, res)
24:                 queue.push(pred)
25:             if res ≠ R_exit and res has no successor then
26:                 succ ← GetPossibleSucc(res, R, A)
27:                 partialSkeleton.connect(res, succ)
28:                 queue.push(succ)
29:         Skeleton.add(partialSkeleton)
30:     return Skeleton
```

Algorithm 1 shows the *Service Flow Exploration* algorithm, which consists of two steps: exploration of skeleton seeds and exploration of end-to-end service flow paths. First, *Service Flow Exploration* explores the skeleton seeds by enumerating all possible interactions between resources (line 1-11). We limit the interactions to be pairwise, since the documentation and implementation of traffic management of service mesh typically split large interactions into several pairwise interactions. *Service Flow Exploration* explores three types of interactions: direct connection (line 5), service flow splitting (line 7), and service flow merging (line 9), since the direct connection is the most common interaction in traffic management of service mesh, and the splitting and merging interactions challenge correctness of complex priority competition and conflict resolution. Notably, the interactions are not limited in scale and type, and they can be extended to more complex interactions in practice, such as increasing the number of interleaving resources or adding more complex interactions.

Second, *Service Flow Exploration* explores end-to-end service flow paths for each skeleton seed to construct end-to-end service flow skeletons (line 16-29). The key idea of the exploration is to extend the service flow path from the skeleton seed to the designated entry and the exit points. For the entry side, the algorithm initializes a queue with seed resources (line 18), recording resources without predecessors. It then iteratively pops a resource from the queue, appending any possible predecessor if the resource lacks one (lines 21-23). This predecessor is subsequently re-enqueued for further exploration. This iterative process drives the service flow path towards the entry point, continuing until every service flow path includes the entry point (line 21). The exit side is explored analogously, ensuring that the service flow paths remain connected from end to end. The correctness of the algorithm and its property of not falling into an infinite loop are guaranteed by an intrinsic property of traffic management for service mesh: any type of resource has an end-to-end service flow path that passes through it, and the length of the path is bounded. Thus, the algorithm will not fall into a loop, since a service flow with unbounded length is not permitted in the system. Importantly, the algorithm does not seek to enumerate all possible service flow skeletons for a given skeleton seed, since the interaction among resources outside the initial seed can be later considered as new seeds to construct new service flow skeletons. *Service Flow Exploration* prioritizes maintaining simplicity in service flow skeletons to facilitate bug determination and localization.

4.2 Filling Service Flow Bodies

Filling service flow bodies is the second stage of the testing workflow. It extends service flow skeletons to complete input configurations which are ready for the system input and following testing steps. Compared with the service flow skeleton, filling the service flow bodies requires more considerations on the configuration formats and detailed constraints. Besides, MeshTest needs to realize the connectivity described in the abstract service flow skeleton in concrete service mesh configurations. To do so, MeshTest leverages a two-step approach to fill the service flow bodies. The first step fills the connector fields to ensure the connectivity of resources. The second step extends the configuration with more detailed rules and options.

The main challenge of the first step is how to connect resources as specified in the service flow skeleton. The service flow skeleton only describes the connectivity of resources, but does not specify the specific connector fields. MeshTest automatically sets connector fields to realize the connectivity of resources. For example, MeshTest connects a routing resource and a workload dispatching resource by setting related values on connector fields in both resources. Further in some cases, connecting adjacent resource pairs does not guarantee the connectivity of the whole service flow path. Taking a practical Istio test case as an example, the service flow skeleton specifies a service flow path that sequentially passes through routing rules, workload dispatching rules, and service definitions. Since the routing rule specifies the host and port fields, the service definition has to specify the same host and port fields though they are not directly connected.

To ensure the connectivity of the whole service flow path, MeshTest tracks the core keys of service flow (such as host and port), propagates among resources from the entry, and fills the connector fields in a topological order.

When the connectivity is ensured, the second step fills other fields with a constraint-aware fuzzing technique. The constraints include ranges of values for each field (*e.g.*, the type of `StringMatch` has to be one of `EXACT`, `PREFIX`, `REGEX`) and field conflicting (*e.g.*, if `resolution` is set to `DNS`, host can not be wildcard). Besides, this step has to guarantee newly filled fields do not conflict with the connector fields. Moreover, we intentionally set some fields with invalid values (prohibited in the documents) or some special values (*e.g.*, empty string, wildcard domain) to challenge the robustness of the service mesh systems. To summarize, the service flow body filling transforms each abstract service flow skeleton into many concrete and detailed input configurations, meanwhile enriches the input configurations with more random options and rules.

4.3 Fine-grained Service Flow Model

The first two stages of the testing workflow focus on creating various end-to-end input configurations for the service mesh system. With the generated input configurations, the next two stages build a strong *Service Mesh Oracle* to check whether the service mesh system correctly realizes the network functions defined in the input configurations.

Gaps between skeletons and facts. Although first two stages are service flow centric, they do not build a precise and fine-grained model to reason about the accurate effects of the input configuration. The first three parts of Figure 5 show the gap between the service flow skeleton, input configuration and the real network behaviors. On one hand, the service flow skeleton does not model the full effects of resources. For example, in the service flow skeleton, the `se-1` is only used to define the traffic entrance, but in real service mesh systems the `service entry` also influences the dispatching stage as a supplementary endpoint provider. On the other hand, the service flow skeleton does not model fine-grained rules such as priority selection and default routes. For example shown in the third part of Figure 5, `vs-1` has higher priority than `vs-2` since exact matching has higher priority than wildcard matching. Another example is the grey blocks, which represent the default routes on port 80.

Modeling fine-grained semantics with CFG. In order to provide a precise and fine-grained model for the whole semantics specified in the input configuration, MeshTest uses a control flow graph (CFG) to describe the processing logics of an arbitrary network request. The CFG is a directed acyclic graph (DAG), which is composed of `action` nodes and `predicate` nodes. Each `action` node represents setting a field or metadata in the network request. Each `predicate` node represents a condition that determines whether the following nodes adapt to the specific request. The CFG inherits intrinsic three-stage structure of service mesh systems: traffic entrance, service routing, and workload dispatching. Each resource acts as a subgraph in the CFG, and the priority selections are modeled as the priority comparisons between different subgraphs.

An example of the fine-grained service flow CFG is shown in the third column of Figure 5. MeshTest uses interpreters (provided by developers) to convert the input configurations into service flow CFGs. The interpreters vary with different service mesh systems, require domain specific knowledge, and cost manual efforts. Notably, the manual efforts are inevitable: on one hand, different service mesh systems have varying input APIs and specific rules, making it infeasible to develop a universal interpreter; on the other hand, the interpreter must accurately represent the functionalities of the service mesh, which is difficult to infer automatically from documentation or code.

MeshTest provides general CFG APIs to aid the users and reduce manual efforts in developing the interpreter. First, MeshTest provides a set of universal CFG primitives, which are common in all service mesh systems. The primitives include nodes and edges in the control flow graph, basic three-stage CFG backbone structure, and expressions and statements to describe the semantics of service mesh resources. Second, MeshTest provides universal abstraction of network requests and service mesh metadata, which is applicable to all service meshes. The network request abstraction includes host, headers and other contents in the TCP and HTTP request. The metadata depict the status of the request, such as source and destination. MeshTest also provides APIs to effectively encode hosts and URIs by splitting them into substrings and further encoding them to integers. Third, MeshTest provides common utility functions, which are frequently used in all service mesh systems. The utility functions include string matching, priority comparisons, default routes, and so on.

MeshTest also provides strategies and guides to build the interpreter. First, the interpreter should build up the backbone structure with its CFG API, set entry and exit nodes, and initialize request fields and metadata. Second, the interpreter needs to encode each resources in the input configuration to individual subgraphs. Each subgraph contains a `predicate` node as the entry guardian, and other following nodes for the detailed semantics of the resource. Third, the interpreter should connect the subgraphs into the holistic CFG, resolve priority competitions between different resources, and add default routes to the CFG. In the example of Figure 5, the `vs-1` and `vs-2` are in different priority levels due to different matching types. The interpreter also utilizes other detailed rules to resolve priority conflicts, such as resource sequence and resource name. After initial development, the interpreter might be incorrect due to bugs or misunderstandings. To address this issue, MeshTest adopts an iterative development approach, conducting tests on the interpreter and the target service mesh system simultaneously, ultimately obtaining an interpreter without false positives (§ 5.4).

With these APIs and strategies, developing and maintaining the interpreter for specific service mesh systems become much easier for the developers. In practice, we have developed interpreters for Istio and Linkerd, and the manual effort is less than two person-weeks. After that, the interpreter is reusable for further development and evolvement

4.4 Testing with Real Network Requests

Judging whether the service mesh system correctly realizes the input configuration is the final step of the testing workflow. The realization is highly abstracted network behavior and cannot be directly observed or compared. The only way is to send real network requests into the mesh network and judge the correctness of output requests. Because of the infinity of network requests, we have to select a finite set of network requests which is comprehensive enough to represent arbitrary network requests. In particular, the test request suite should cover all detailed rules and corner cases appeared with subtle interactions of resources. It should also produce reference results for each output request. After that, we design a test driver which is able to set up the testbed environment, apply the input configuration to the service mesh system, send the network requests, capture output requests, and check results with the expected behaviors.

To produce the test request suite, we leverage symbolic execution [11, 16] on the service flow CFG. The symbolic execution enumerates all types of network requests by exploring all possible paths in the CFG. During the execution, the symbolic execution engine tracks and updates the states and path conditions on each nodes. When the execution reaches a `predicate` node, it evaluates the condition with a SMT solver [17] and decides whether the path is reachable. When the execution reaches an `action` node, it updates the request fields and metadata based on the action. After exploring all paths in the CFG, the symbolic execution engine evaluates the corresponding input requests and reference results, and thus forms a comprehensive test request suite. Notably, since the input configurations obtained from the MeshTest's input generator are not large in scale, symbolic execution here will not encounter scalability issues (§ 5.3).

After that, the test driver automatically conducts tests on a real service mesh testbed. The input of each test campaign contains an input configuration and a set of network requests and reference results. The test driver converts the test suite into real requests by supplementing arbitrary values (unrestricted in the symbolic execution), attaching a test magic number to distinguish test requests and other network requests, and setting indexes to classify individual tests. It then sends the requests to the service mesh system, captures and parses the results, and compares the results with the reference results. The test driver also analyzes system status and logs to check for crashes and internal errors. Finally, the test driver reports the test results, identifying the failed input configurations and their corresponding network requests.

4.5 Implementation

We implement MeshTest with roughly 8300 lines of JAVA and Python code, among which 1400 lines for configuration resource descriptions (850 lines for Istio, 350 lines for Linkerd and 200 lines of shared codes), 1800 lines for the input generator, 3600 lines for CFG modeling and symbolic execution (1700 lines for Istio, 500 lines for Linkerd and 1400 lines of shared codes) and 1500 lines for the test driver. To enhance the generality and extensibility of MeshTest, we architect the system components in a loosely-coupled manner and delineate a suite of domain-specific interfaces within the shared codebase. Specifically, the shared code responsible for resource descriptions incorporates a parser tailored for domain-specific resource templates. Similarly, the shared code dedicated to CFG modeling furnishes a comprehensive set of interfaces for constructing the CFG, including nodes and edges, domain-specific expressions and constraints, and utility functions designed to encapsulate common semantic operations such as string matching and priority selection.

5 Evaluation

We conduct extensive evaluation on two most popular service mesh systems, Istio [2] and Linkerd [3], to demonstrate the effectiveness of MeshTest in finding new bugs and improving the reliability of service mesh systems. Our main evaluation results are summarized as follows:

- MeshTest has found 23 *new* bugs that escape from existing test cases. Among them, 19 have been confirmed and 10 has been fixed.
- MeshTest covers all properties specified by single resource and pairwise interactions between resources.
- MeshTest is efficient in end-to-end test case generation and oracle checking. The input generator produces 2500 end-to-end test cases per second. On average, the *Service Mesh Oracle* checks each test case using 29 distinct real-world requests within 15 seconds.
- MeshTest reports no false positives. Each test alarm during the test campaigns points to a violation of the specification of the target service mesh system.

5.1 Finding New Bugs

As shown in Table 1, MeshTest has found 23 previously unknown bugs in two most popular service mesh systems. We reported the bugs to developers and they confirmed 19 of them and fixed 10. Many of them stand in the critical path of request processing and can lead to severe failures, including incorrect state, load imbalance and security vulnerabilities. Figure 6 shows a real bug found by MeshTest, where the service request mistakenly skips the routing stage. This bug may lead to imbalanced load, and more seriously, it may disable security policies in the routing stage, and cause security vulnerabilities. It is noteworthy that the primary reasons for the majority of bugs being found in Istio [2] are twofold. On one hand, Istio

Index	Implementation	Bug Description	Status
1 [18]	Istio 1.19–1.21	Empty prefix in specific fields causes an internal error	Fixed
2 [19]	Istio 1.19–1.21/dev	Port 80 is not open by default when Istio gateways are not installed	Reported
3 [20]	Istio 1.19–1.21/dev	Traffic passthroughs cluster when `service entry endpoints` set to an internal IP	Reported
4 [21]	Istio 1.19–1.21/dev	`Service entry` with wildcard host makes traffic skip service routing	Confirmed
5 [22]	Istio 1.19–1.21/dev	`Service entry` defined on port 80 disables `virtual service`	Confirmed
6 [23]	Istio 1.22dev	Routing fails under multiple interleaved resources	Fixed
7 [24]	Istio 1.19–1.21/dev	Traffic is not dropped when port not matched in `virtual service`	Confirmed
8 [25]	Istio 1.19–1.21	`WithoutHeaders` matching fails without target header	Fixed
9 [26]	Istio 1.19–1.21	Delegation influences the priority between `virtual services`	Fixed
10 [27]	Istio 1.19–1.21/dev	Match conditions influence the choice of `virtual service` for gateway	Confirmed
11 [28]	Istio 1.19–1.21/dev	`Service` defined on port 80 disables `virtual service`	Reported
12 [29]	Istio 1.19–1.21	Update on `targetPort` does not trigger update on `EDS`	Fixed
13 [30]	Istio 1.19–1.21/dev	Wildcard matching fails on destination host	Reported
14 [10]	Istio 1.19–1.21	Collision between `service entries` with same host but different workloads	Fixed
15 [31]	Istio 1.19–1.21/dev	`EDS` missing for `service entry` defined on the same host as `service`	Confirmed
16 [32]	Istio 1.19–1.21/dev	`WorkloadSelector` takes effect at wrong place	Confirmed
17 [33]	Istio 1.19–1.21/dev	Header manipulation fails when the value is empty string	Confirmed
18 [34]	Istio 1.19–1.21/dev	Special headers are not ignored in match conditions	Confirmed
19 [35]	Istio 1.19/1.20	Header manipulation fails on pseudo headers	Fixed
20 [36]	Linkerd 2.14	Linkerd extension drives specific pods crash	Fixed
21 [37]	Linkerd 2.14	Routing error under rules with the same matching conditions	Fixed
22 [38]	Linkerd 2.14	Routing error under `http routes` bound on the same `gateway`	Fixed
23 [39]	Linkerd 2.14/dev	Incorrect `hostnames` effects	Confirmed

Table 1: Bugs found by MeshTest. Istio [2] and Linkerd [3] are two most widely-used service mesh systems. All bugs are newly discovered in supported versions or main branch (dev version). All bugs had escaped from existing test suites.

provides more complex and flexible features compared to Linkerd [3], which inherently raises the risk of encountering bugs. Istio exhibits significantly more issues than Linkerd. On the other hand, due to Istio's widespread adoption and rich feature set, MeshTest has implemented more configuration resources and more detailed service flow modeling specifically for Istio. Table 2 demonstrates categories and sources of these new bugs: entrance error, routing error, dispatching error, internal error and others.

Entrance error. These bugs indicate that requests are not captured correctly by the service mesh system. Entrance errors can cause the system to capture the wrong traffic or miss some traffic. Issue #2 in Table 1 shows a typical entrance error found by MeshTest. When no gateway is explicitly set, the default traffic entrance on port 80 is not added for internal requests to the virtual service. These bugs usually result in traffic loss and disable other functionalities.

Routing error. These bugs are rooted in the complex routing rules, including those specified by users and those generated by default. They are usually related to some complicated functionalities, such as priority competition and wildcard matching. Figure 7 describes a bug when two `virtual services` share the same host but have different routing behaviors. The correct priority should follow the creation order of the `virtual services`, but the `delegation` causes incorrect priority. Previous approaches cannot detect this bug, since it is triggered by specific combinations of routing rules. MeshTest generates test cases to cover the priority competition of routing rules with the *Service Flow Exploration* technique, and successfully detected this bug.

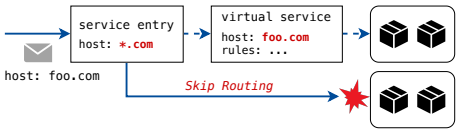


Figure 6: A real bug [21] in Istio discovered by MeshTest, where `service entry` with wildcard host makes the request skip the routing stage.

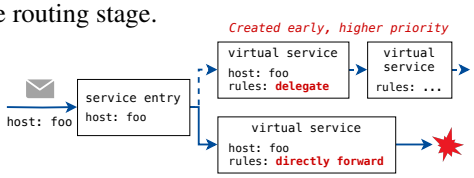


Figure 7: A real bug [26] in Istio discovered by MeshTest, where delegation causes incorrect priority.

Dispatching error. These bugs occur in dispatching the service requests to the concrete workloads. They are usually caused by incorrect assignment of the traffic to the workloads or wrong attributes of the services. Figure 2 shows a severe dispatching error, where two `service entries` with the same host collide with each other when they refer to different workloads. This bug is caused by the incorrect transformations from workloads to services, and it makes the system fail to dispatch traffic to partial workloads. It is confirmed by the developers and fixed in the next release.

Internal error. These bugs occur with an unhandled exception or an undesired state due to the absence of exception handling or internal conflicts. Although internal errors are not the primary target of MeshTest, we still detect some of them by simple but effective techniques such as setting invalid values for configuration properties, checking logs, and

	Istio	Linkerd	Total
Entrance error	1	0	1
Routing error	9	3	12
Dispatching error	5	0	5
Internal error	1	1	2
Others	3	0	3
Total	19	4	23

Table 2: Classification of the bugs found by MeshTest.

monitoring component liveness.

Others. Besides the above four types of bugs, MeshTest also finds some other bugs, such as manipulation failure and inconsistency between specification and implementation. We reported them to developers and received positive feedback.

5.2 Coverage

MeshTest achieves 100% coverage on the functionalities specified in single resource and functionalities indicated by the interactions between pairwise resources. Both coverages are critical for the correctness of service mesh traffic management, and are not fully covered with existing test suites. Even the coverage of functionalities in resource interactions is rarely considered in the design of test suites. MeshTest traverses all possible value types (exact, wildcard, etc.) for each resource field when filling the service flow bodies. MeshTest leverages the domain specific *Service Flow Exploration* technique to enumerate all possible pairwise interactions between resources to cover the functionalities rooted within. For the results, MeshTest detects many bugs related to resource interactions based on the full coverage, including Issue #4, Issue #5 and Issue #9 in Table 1. Besides, MeshTest also finds some bugs related to functionalities specified by single resource, such as Issue #8 in Table 1.

MeshTest improves code coverage by enumerating all possible resource interactions with the *Service Flow Exploration* technique and fuzzing detailed options with service body filling. We conduct code coverage analysis on the controller of Istio (`pilot-discovery`). The results show that: (1) MeshTest achieves 41.2% statement coverage for traffic management packages and 32.2% statement coverage for the entire controller; (2) MeshTest increases the statement coverage of existing test suites from 74.1% to 78.8% for traffic management packages and from 73.1% to 77.0% for the entire controller; and (3) *Service Flow Exploration* is effective in testing resource interactions, increasing the corresponding statement coverage from 70.9% to 79.4%.

5.3 Test Efficiency

The efficiency of MeshTest is evaluated in two aspects: how fast MeshTest can generate diverse end-to-end test inputs and how long it takes to execute and check each test case. We evaluate MeshTest in a virtual Kind [40] cluster on a lightweight host machine. On one hand, the input generator of MeshTest is efficient and effective. MeshTest leverages *Service Flow Exploration* to create service flow skeletons and fills service flow bodies to generate diverse end-to-end test cases with different service flows. MeshTest produces 2500 end-to-end test cases per second, and the cases need to be tested with an average of 29 different service requests. On the other hand, the *Service Mesh Oracle* checks whether the target system behaves consistently with the input configuration efficiently. For each test input, the *Service Mesh Oracle* builds the CFG, performs symbolic execution, and sends and receives requests in less than 15 seconds on average. As shown in Table 3, more than 99% of time is spent on environment setting, request sending and receiving, which is inevitable in the end-to-end testing. The environment setting time includes applying and deleting the end-to-end test configuration on the testbed environment, and cannot be reduced by testing tools. It is possible to set up multiple test environments in parallel to significantly improve the efficiency of *Service Mesh Oracle*.

	Set Env	Build CFG	Symbolic Execution	Send/Recv Requests	Total
Istio (Sidecar)	5.405	0.008	0.046	5.913	11.372
Istio (Ambient)	4.793	0.009	0.03	5.605	10.437
Linkerd	9.92	0.08	0.043	4.895	14.938

Table 3: Average time (in seconds) for each test input spent in *Service Mesh Oracle*.

5.4 False Positives

MeshTest prevents false positives under the assumptions of *Service Mesh Oracle* correctness. Each test alarm during the test campaign points to an inconsistency between the exprected behavior (specified in documentation) and actual behavior of the target system, and further indicates a bug or undefined behavior. To make sure the correctness of *Service Mesh Oracle*, we develop the service flow CFG model iteractively when we are detecting bugs in the target system. Specifically, we develop a basic service flow CFG model based on the documentation of the target system, and then we run the test campaigns to find violations. Then we determine whether the violations are caused by the bugs in the target system or the incorrectness of the CFG model. If the violations are caused by the CFG model, we refine the model and iteratively repeat the process until there are no more model errors. It is not feasible to develop a universal CFG model for all service mesh systems, since they have different functionalities and no standard specifications. We respectively developed the CFG model for Istio and Linkerd with less than two person-weeks of effort and finally made it reliable. We argue that although we cannot prove the correctness of the CFG model, it stops producing false positives quickly just after several iterations.

6 Discussion

Limitations and future work. MeshTest is a first step towards automatic end-to-end testing for traffic management of service mesh. Like other testing techniques, MeshTest is

incomplete and it can miss bugs. MeshTest does not cover all possible input configuration space. Both the input generator and the testing oracle rely on the domain specific service flow model, which can not cover implementation specific details. This design aims to balance efficiency and coverage—it covers all pairwise interactions between resources and requests specified in the documentation. The results show that MeshTest can find bugs in real-world service mesh systems, but the coverage and scope of testing can be further improved. MeshTest relies on domain knowledge to provide the adjacency matrix in *Service Flow Exploration* and build the fine-grained CFG model for *Service Mesh Oracle*. Although most of the knowledge can be obtained from the documentation, there are some details that are not well documented and require experience as system users. The soundness of test results rely on the correctness of the service flow CFG model. MeshTest does not test service mesh performance, security, or network topologies. MeshTest does not aim to test the state reconciliation of the service mesh system.

Generality. The approach of MeshTest is not tied to the specific implementation of service mesh systems and can be generalized to other service mesh systems with different functionality models and configuration resource definitions. The generalization requires domain knowledge of service mesh for *Service Flow Exploration* and service flow CFG model construction. The generalization cost is inevitable due to the absence of a standard for service mesh systems. To reduce the effort of generalization, MeshTest introduces the universal principles of service mesh traffic management into the design and implementation, and provides a set of domain specific interfaces to facilitate generalization. These interfaces includes template resource definitions, domain specific expressions and constraints, and utility functions designed to encapsulate common semantic operations. Besides service mesh, the approach of MeshTest can be extended to other service flow centric systems with a high level declarative input specification for network behaviors, such as application defined networks [8] and software defined networks [41].

7 Related Work

Application defined networks and service meshes. Both service mesh [42] and application defined network (ADN) [8, 9] provide a way to decouple the network communication and the application logic, and releases the burden of development and maintenance. Besides industry practices [7], service mesh systems can be used to empower secure microservice applications [43], API managements [44], tracing systems [45], and so on [46]. Some works focus on the performance and efficiency of service mesh systems [9, 47, 48], which is orthogonal to our work. Application defined networks (ADN) [8] share the similar goal with service meshes, and it has more flexibility and better performance than service mesh [9]. ADN is also service flow centric and is expected to be a potential alternative of service meshes. We believe MeshTest can be easily retrofitted to ADN systems.

Reliability of microservice systems. There is rich literature on improving microservice systems by testing clusters [14, 49–52], RPC communication [53], system operators [15], and so on [54–64]. Among them, Sieve [14] and Acto [15] focus on the state reconciliation in the microservice systems. MeshTest focuses on the service flow and the end-to-end communication in service mesh systems, which is orthogonal to the state change reconciliation. The input generation technique in Acto [15] does not suit for service mesh systems since it relies on a mapping between the input resources and the system state resources. The traffic management of service mesh mainly focus on network communication rules and do not have a direct mapping to the system state.

Checking system correctness with formal techniques. Formal methods have been used to improve the system correctness in various domains, such as verification on system codes [65–75], system configuration verification [76–84]. Ucheck [85] is a tool to check whether a given invariant would hold based on modular microservice models. Ucheck [85] mainly focuses on the correctness of application logic, which is orthogonal to the communication correctness in service mesh systems. Model based testing builds a model of the system and creates test cases covering the model rather than the system code. It is lightweight and has been widely-used in testing large and complex systems [86–91]. *Service Mesh Oracle* is inspired by the art of model based testing, and builds a common model for service mesh functionalities.

8 Conclusion

We present MeshTest, an automatic end-to-end testing framework for traffic management of service mesh. MeshTest centers around an explicit service flow model, generates end-to-end test inputs with a domain specific *Service Flow Exploration* technique, and checks correctness with a *Service Mesh Oracle* which leverages formal methods to generate test request suites. MeshTest is proved to be practical for detecting 23 *new* bugs in real service mesh systems.

Acknowledgments. We thank our shepherd, T. S. Eugene Ng, and the anonymous reviewers for their valuable feedback. We thank all the Istio, Linkerd, and Kubernetes developers who engaged with us and reviewed our reports. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700, the National Natural Science Foundation of China under the grant numbers 624B2007, 62172008 and 62325201, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xin Jin is the corresponding author. Naiqian Zheng, Tianshuo Qiao, Xuanzhe Liu, and Xin Jin are also affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[2] Istio Authors, "The istio service mesh." https://istio.io, 2024.

[3] Linkerd Authors, "The linkerd service mesh." https://linkerd.io, 2024.

[4] E. Song, Y. Song, C. Lu, T. Pan, S. Zhang, J. Lu, J. Zhao, X. Wang, X. Wu, M. Gao, *et al.*, "Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture," in *ACM SIGCOMM*, 2024.

[5] Amazon Web Services, "The aws app mesh." https://aws.amazon.com/app-mesh, 2024.

[6] H. Gui, Y. Xu, A. Bhasin, and J. Han, "Network a/b testing: From sampling to estimation," in *ACM Web Conference*, 2015.

[7] I. Authors, "Istio case studies." https://istio.io/latest/about/case-studies/, 2024.

[8] X. Zhu, W. Deng, B. Liu, J. Chen, Y. Wu, T. Anderson, A. Krishnamurthy, R. Mahajan, and D. Zhuo, "Application defined networks," in *ACM SIGCOMM HotNets Workshop*, 2023.

[9] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, *et al.*, "Dissecting overheads of service mesh sidecars," in *ACM SOCC*, 2023.

[10] "Collision between `service entries` with the same host." https://github.com/istio/istio/issues/49550, 2024.

[11] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, 1976.

[12] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[13] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy*, 2018.

[14] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *USENIX OSDI*, 2022.

[15] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *ACM SOSP*, 2023.

[16] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[17] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[18] "Empty prefix causes routing error." https://github.com/istio/istio/issues/48534, 2023.

[19] "Port 80 is not open by default when istio gateways are not installed." https://github.com/istio/istio/issues/49991, 2024.

[20] "Http traffic passthroughs cluster when `service entry endpoints` set to internal address." https://github.com/istio/istio/issues/49430, 2024.

[21] "Wildcard host in `service entry` results in routing error." https://github.com/istio/istio/issues/49485, 2024.

[22] "`Service entry` disables `virtual service` on port 80." https://github.com/istio/istio/issues/49508, 2024.

[23] "Routing error under multiple `virtual services` and `service entries`." https://github.com/istio/istio/issues/49509, 2024.

[24] "Traffic is not dropped when `virtual service HTTPMatch` is not matched by port." https://github.com/istio/istio/issues/49516, 2024.

[25] "`WithoutHeaders` denies packets without the target header." https://github.com/istio/istio/issues/49537, 2024.

[26] "Delegation influences which `virtual service` takes effect." https://github.com/istio/istio/issues/49539, 2024.

[27] "`HTTPMatch` affects which `virtual service` to be chosen for gateway." https://github.com/istio/istio/issues/49588, 2024.

[28] "Service defined on port 80 causes `virtual service` cannot capture traffic with specific host." https://github.com/istio/istio/issues/49673, 2024.

[29] "Update on `service entry targetPort` does not trigger update on eds." https://github.com/istio/istio/issues/49878, 2024.

[30] "Wildcard host in `service entry` cannot match the same host in the routing destination." https://github.com/istio/istio/issues/49482, 2024.

[31] "`CDS EDS` are missing for `service entry` defined on the same host and different port with another service." https://github.com/istio/istio/issues/50163, 2024.

[32] "`WorkloadSelector` takes effect in `service entry` with resolution `DNS`." https://github.com/istio/istio/issues/50164, 2024.

[33] "Header operation does not take effect if value is set to empty string." https://github.com/istio/istio/issues/49553, 2024.

[34] "Uri, scheme, method, authority are not ignored in `HTTPMatchRequest` header keys." https://github.com/istio/istio/issues/48555, 2023.

[35] "Inconsistent behaviors on pseudo headers." https://github.com/istio/istio/issues/48605, 2024.

[36] "`Viz` crashes `Pod`s created by ingress controller." https://github.com/linkerd/linkerd2/issues/12344, 2024.

[37] "Different routing behavior between gateway and sidecar under several same `HTTPRouteMatch`." https://github.com/linkerd/linkerd2/issues/12267, 2024.

[38] "Inconsistent routing choice between gateway and sidecar under `http route`s with same parent," 2024. Github issue id omitted for anonymity.

[39] "Linkerd implementation of gateway api does not follow `GAMMA` specification." https://github.com/linkerd/linkerd2/issues/12295, 2024.

[40] Kubernetes Community, "Kind," 2024.

[41] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, 2014.

[42] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019.

[43] R. Chandramouli, Z. Butcher, *et al.*, "Building secure microservices-based applications using service-mesh architecture," *NIST Special Publication*, 2020.

[44] F. Hussain, W. Li, B. Noye, S. Sharieh, and A. Ferworn, "Intelligent service mesh framework for api security and management," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019.

[45] D. Cha and Y. Kim, "Service mesh based distributed tracing system," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, 2021.

[46] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth, "Automated testing and resilience of microservice's network-link using istio service mesh," in *2022 31st Conference of Open Innovations Association (FRUCT)*, 2022.

[47] H. Saokar, S. Demetriou, N. Magerko, M. Kontorovich, J. Kirstein, M. Leibold, D. Skarlatos, H. Khandelwal, and C. Tang, "{ServiceRouter}: Hyperscale and minimal cost service mesh at meta," in *USENIX OSDI*, 2023.

[48] M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan, and E. Verplanke, "Challenges and opportunities in performance benchmarking of service mesh for the edge," in *2021 IEEE international conference on edge computing (EDGE)*, 2021.

[49] C. Lou, Y. Jing, and P. Huang, "Demystifying and checking silent semantic violations in large distributed systems," in *USENIX OSDI*, July 2022.

[50] H. Wu, J. Pan, and P. Huang, "Efficient exposure of partial failure bugs in distributed systems with inferred abstract states," in *USENIX NSDI*, 2024.

[51] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, "Understanding and detecting software upgrade failures in distributed systems," in *ACM SOSP*, 2021.

[52] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems," in *ACM ASPLOS*, 2020.

[53] Y. Chen, X. Sun, S. Nath, Z. Yang, and T. Xu, "Push-Button reliability testing for Cloud-Backed applications with rainmaker," in *USENIX NSDI*, 2023.

[54] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in *ACM ASPLOS*, 2019.

[55] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *ACM SOSP*, 2017.

[56] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *ACM SOSP*, 2021.

[57] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *ACM SOSP*, 2021.

[58] X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min, "Witcher: Systematic crash consistency testing for non-volatile memory key-value stores," in *ACM SOSP*, 2021.

[59] A. Quinn, J. Flinn, M. Cafarella, and B. Kasikci, "Debugging the {OmniTable} way," in *USENIX OSDI*, 2022.

[60] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan, "The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure," in *ACM SOSP*, 2019.

[61] Z. Chen, Y. Hua, Y. Zhang, and L. Ding, "Efficiently detecting concurrency bugs in persistent memory programs," in *ACM ASPLOS*, 2022.

[62] L. Tang, C. Bhandari, Y. Zhang, A. Karanika, S. Ji, I. Gupta, and T. Xu, "Fail through the cracks: Cross-system interaction failures in modern cloud systems," in *EuroSys*, 2023.

[63] Y. Hu, G. Huang, and P. Huang, "Automated reasoning and detection of specious configuration in large systems with symbolic execution," in *USENIX OSDI*, 2020.

[64] X. J. Ren, S. Wang, Z. Jin, D. Lion, A. Chiu, T. Xu, and D. Yuan, "Relational debugging—pinpointing root causes of performance problems," in *USENIX OSDI*, 2023.

[65] H. Ma, H. Ahmad, A. Goel, E. Goldweber, J.-B. Jeannin, M. Kapritsos, and B. Kasikci, "Sift: Using refinement-guided automation to verify complex distributed systems," in *USENIX ATC*, 2022.

[66] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "I4: incremental inference of inductive invariants for verification of distributed protocols," in *ACM SOSP*, 2019.

[67] N. Zheng, M. Liu, Y. Xiang, L. Song, D. Li, F. Han, N. Wang, Y. Ma, Z. Liang, D. Cai, E. Zhai, X. Liu, and X. Jin, "Automated verification of an in-production dns authoritative engine," in *ACM SOSP*, 2023.

[68] U. Sharma, R. Jung, J. Tassarotti, F. Kaashoek, and N. Zeldovich, "Grove: a separation-logic library for verifying distributed systems," in *ACM SOSP*, 2023.

[69] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *ACM ASPLOS*, 2023.

[70] X. Li, X. Li, W. Qiang, R. Gu, and J. Nieh, "Spoq: Scaling {Machine-Checkable} systems verification in coq," in *USENIX OSDI*, 2023.

[71] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *USENIX OSDI*, 2022.

[72] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu, "Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware," in *ACM SOSP*, 2021.

[73] J. Yao, R. Tao, R. Gu, and J. Nieh, "Mostly automated verification of liveness properties for distributed protocols with ranking functions," in *ACM POPL*, 2024.

[74] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *ACM SOSP*, 2019.

[75] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "{DistAI}:{Data-Driven} automated invariant learning for distributed protocols," in *USENIX OSDI*, 2021.

[76] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *USENIX NSDI*, 2017.

[77] S. Renganathan, B. Rubin, H. Kim, P. L. Ventre, C. Cascone, D. Moro, C. Chan, N. McKeown, and N. Foster, "Hydra: Effective runtime network verification," in *ACM SIGCOMM*, 2023.

[78] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime verification for real networks," in *USENIX NSDI*, 2020.

[79] K. Zhang, D. Zhuo, A. Akella, A. Krishnamurthy, and X. Wang, "Automated verification of customizable middlebox properties with gravel," in *USENIX NSDI*, 2020.

[80] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *USENIX NSDI*, 2020.

[81] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "NetSMC: A custom symbolic model checker for stateful network verification," in *USENIX NSDI*, 2020.

[82] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *ACM SIGCOMM*, 2020.

[83] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea, "Automated verification of network function binaries," in *USENIX NSDI*, 2022.

[84] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Bagpipe: Verified bgp configuration checking," in *ACM OOPSLA*, 2016.

[85] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *USENIX OSDI*, 2017.

[86] C. Li, Y. Jiang, C. Xu, and Z. Su, "Validating jit compilers via compilation space exploration," in *ACM SOSP*, 2023.

[87] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *ACM SOSP*, 2021.

[88] K. D. Albab, J. DiLorenzo, S. Heule, A. Kheradmand, S. Smolka, K. Weitz, M. Timarzi, J. Gao, and M. Yu, "Switchv: automated sdn switch validation with p4 models," in *ACM SIGCOMM*, 2022.

[89] N. Zheng, M. Liu, E. Zhai, H. H. Liu, Y. Li, K. Yang, X. Liu, and X. Jin, "Meissa: scalable network testing for programmable data planes," in *ACM SIGCOMM*, 2022.

[90] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," in *USENIX OSDI*, 2004.

[91] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX NSDI*, 2013.

Preventing Network Bottlenecks: Accelerating Datacenter Services with Hotspot-Aware Placement for Compute and Storage

Hamid Hajabdolali Bazzaz[1], Yingjie Bi[1], Weiwu Pang[1], Minlan Yu[2], Ramesh Govindan[3], Neal Cardwell[1], Nandita Dukkipati[1], Meng-Jung Tsai[1], Chris DeForeest[1], Yuxue Jin[1], Charlie Carver[4], Jan Kopański[1], Liqun Cheng[1], Amin Vahdat[1]

[1]*Google*, [2]*Harvard University*, [3]*University of Southern California*, [4]*Columbia University*

Abstract

Datacenter network hotspots, defined as links with persistently high utilization, can lead to performance bottlenecks. In this work, we study hotspots in Google's datacenter networks. We find that these hotspots occur most frequently at ToR switches and can persist for hours. They are caused mainly by *bandwidth demand-supply imbalance*, largely due to high demand from network-intensive services, or demand exceeding available bandwidth when compute/storage upgrades outpace ToR bandwidth upgrades. Compounding this issue is *bandwidth-independent task/data placement* by datacenter compute and storage schedulers. We quantify the performance impact of hotspots, and find that they can degrade the end-to-end latency of some distributed applications by over 2× relative to low utilization levels. Finally, we describe simple improvements we deployed. In our cluster scheduler, adding hotspot-aware task placement reduced the number of hot ToRs by 90%; in our distributed file system, adding hotspot-aware data placement reduced p95 network latency by more than 50%. While congestion control, load balancing, and traffic engineering can efficiently utilize paths for a fixed placement, we find hotspot-aware placement – placing tasks and data under ToRs with higher available bandwidth – is crucial for achieving consistently good performance.

1 Introduction

Distributed cloud applications and services rely on high-bandwidth networks to provide high-throughput, low-latency data transfers. Ideally, the datacenter network provides applications with the illusion of an *unconstrained network* — one in which applications receive high throughput independent of the placement of their components across the network. Techniques such as topology design [41] and/or topology and traffic engineering [32] can support this ideal.

In practice, however, the datacenter network is not unconstrained at all places and all times. In our study of production networks (§2), there exist *network hotspots* — links with persistently high utilization — both within the datacenter interconnect and at top-of-the-rack (ToR) switches (§3). At Google, most hotspots occur at ToR switches and can persist for tens of minutes to hours. ToRs are split between compute hosts managed by Borg [45] and storage hosts. A dominant fraction of traffic in our clusters[1] can be attributed to our distributed storage stack. This includes the Colossus distributed file system [17], the *RamStore*, an in-memory store similar to RamCloud [29], *QuerySys*, an internal query processing system similar to [5, 27, 36, 44], and Bigtable [18]. Colossus places chunks on storage racks, and *QuerySys* worker tasks (§4.2) access these over the network.

Datacenter network planning balances ToR bandwidth supply with anticipated compute and storage traffic demand. Due to the high cost of non-blocking bandwidth provisioning, statistical multiplexing is employed, resulting in an oversubscription ratio – the ratio of ToR downlink to uplink bandwidth. This ratio, set at deployment, typically remains fixed throughout a ToR's multi-year lifespan, based on initial workload forecasts. Consequently, hotspots emerge from two primary, interconnected causes (§3). The first is rack *bandwidth demand-supply imbalance*: an imbalance between bandwidth demand from compute or storage provisioned on a rack, and the provisioned ToR bandwidth supply. This imbalance may occur under specific extreme traffic patterns, such as large incasts, due to the inherent oversubscription ratio. Alternatively, it can arise unexpectedly from deviations in either demand or supply compared to initial provisioning forecasts. Demand-side changes might stem over time from inorganic increased bandwidth demands of network-intensive workloads or other workload efficiency improvements. Supply-side changes could be triggered by incremental upgrades that increase compute or storage capacity without corresponding ToR uplink upgrades, or by planned/unplanned ToR uplink outages that reduce available bandwidth temporarily. The second reason is *bandwidth-independent task/data placement* by datacenter compute or storage schedulers assuming an unconstrained network. Datacenter scheduling is primarily focused on host resources, like compute and storage, that can be evaluated independently for each scheduling decision (because the network was designed to typically not be a bottleneck). Scheduling for network load is a much harder problem, since a hotspot is a property of a set of hosts and not attributable to any individual host. Further, characterizing the bursty communication demands of a distributed job is

[1]In this paper, our scope is traditional compute and storage, excluding ML clusters.

challenging relative to the per-task CPU and memory requirements. Hence, for simplicity, our initial job schedulers did not consider network requirements in scheduling decisions. This could in turn lead to situations where the tasks or storage chunks on a rack had sufficient traffic demand to exceed the provisioned ToR bandwidth supply.

Highly utilized ToRs can degrade the performance of our distributed storage systems (§3.4). For instance, Colossus hard disk drive (HDD) write latencies are inflated by 2× relative to an unloaded ToR even at 40% ToR utilization. Higher levels of the storage stack are also sensitive to ToR hotspots; for example, query latencies in *QuerySys* are inflated by 1.5× at utilizations ranging from 75–95%. It is surprising that these systems are so sensitive to ToR utilization. They access high-latency storage devices like HDDs and can incur significant compute latency when, for example, aggregating query results. Clearly, the increased network latency due to high ToR utilization starts to dominate these other latency components at some utilization level.

Application impact from ToR hotspots cannot be avoided with traditional network performance techniques like congestion control and traffic engineering, since these solutions cannot provide more ToR bandwidth. Hotspots *can*, in theory, be addressed by maintaining ToR resource-balance at every instant but is hard and expensive to do in practice (§3.3). It is much easier to replace a few failed disks on a rack with new higher-capacity disks (resulting in an imbalance) than it is to upgrade ToR capacity to restore balance. Instead, we have to change the input traffic matrix by placing work in a hotspot-aware manner.

In this paper, we modified compute and storage schedulers (Borg and Colossus) to account for resource-imbalance and ToR utilization when making task and storage chunk placements. This is challenging because the scheduler and file system are complex software systems that simultaneously balance a range of competing objectives: ensuring performance for applications with diverse workloads, high utilization, load balance, failure tolerance, and so on.

Despite this, for two reasons, we were able to introduce relatively simple modifications to the scheduler and the file system to mitigate the latency impact of high ToR utilization. First, end-to-end application-perceived query performance is only sensitive to ToR utilization above 75% or so. Second, across our fleet, the *average* ToR utilization is low, so many ToRs are well below 75% utilization. Our *hotspot-aware* task placement (§5) on Borg proactively places tasks on low utilization ToRs, and reactively migrates them away from hotspot ToRs. Similarly, our *hotspot-aware* chunk placement biases storage chunk placement away from those imbalanced racks with less capacity.

These mechanisms reduced the number of hotspot ToRs by 90% in our network without regressing scheduler and file system objectives. Furthermore, they significantly reduce storage access and query latency. We reduced Colossus p95 network latency by 50–80% and p95 total latency by 30–60%. Similarly, *QuerySys* shuffle flush and materialize benchmarks achieve latency reductions of 13% and 9%, respectively.

This paper makes three key contributions:

1. We demonstrate that ToR hotspots remain an unsolved but critical problem, originating from rack bandwidth demand-supply imbalance in dynamic network environments with evolving workloads, incrementally increasing compute/storage resources, and ToR uplink bandwidth prone to planned/unplanned outages. The ToR ↔ aggregation block cross-section is the most common location for hotspots inside our datacenters, and these hotspots can persist for extended durations, ranging from minutes to days. (§3).
2. We show that these hotspots can significantly impact application performance, even for operations dependent on disk or compute latency. Specifically, ToR hotspots can double the latency of both simple storage read/write operations and complex application operations (§4).
3. We find that simple augmentations to existing compute and storage schedulers to better balance network load in a best-effort manner significantly reduce ToR hotspots and their impact on application latency without the added complexity of treating the network as a fully managed resource (§5).

2 Background

2.1 Datacenter Network

Clos topology. Google's datacenter network is based on a hierarchical design (Figure 1) that uses Clos topologies to provide high-bandwidth interconnectivity between servers [41]. The lowest level of the network consists of servers mounted on racks. A top-of-rack (ToR) switch connects to all the servers within the rack. Each ToR switch connects to an aggregation switch. A group of interconnected aggregation switches forms an aggregation block. In turn, aggregation blocks connect to the datacenter network interconnect (DCNI). The DCNI provides links that directly connect different aggregation blocks or indirectly connect them through spine blocks [32, 41]. The DCNI connects to a fabric border router (FBR) that provides connectivity to other datacenters or WAN.

Rack types. To simplify procurement, deployment, and subsequent lifecycle management, Google's datacenters contain different types of racks. A *Disk* rack contains only storage appliances, servers dedicated to I/O using hard disk drives (HDDs) or solid-state drives (SSDs). Optimized for high-capacity storage, they store and serve data for Google's entire suite of user-facing services, as well as analytics, backup, and archival applications. A *Compute* rack contains compute resources, such as servers, with minimal local storage. Finally, a *Mixed* rack contains both compute and storage resources.

2.2 The Storage Stack

Several storage systems provide Google's applications with a range of interfaces to storage hardware. These systems

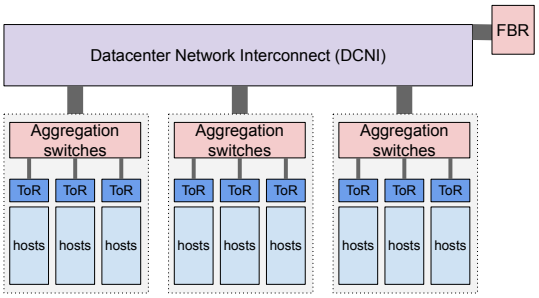


Figure 1: The typical topology of a datacenter network deployed at Google.

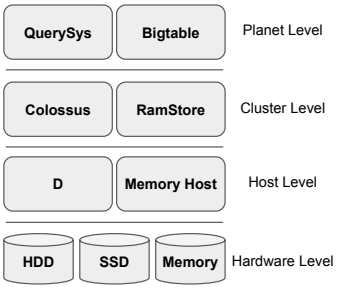


Figure 2: The storage stack at Google.

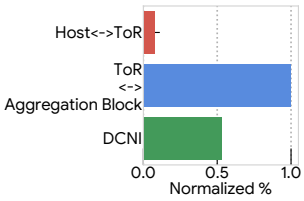


Figure 3: The percentage of hot links at different cross-sections of the network, normalized by the percentage at the ToR ↔ aggregation block layer.

collectively constitute the *storage stack* (Figure 2).

At the lowest level, *D* provides access to chunks of data residing on a single server, but it does not have an API to access user-level files or provide reliability guarantees.

Colossus [17], a distributed file system, is a successor to, and shares some architectural heritage with, the Google File System [16]. It breaks files up into encoded units called *stripes*. Each stripe has a particular encoding, perhaps replicated or erasure-coded, and is comprised of *chunks*, which it stores on *D*. Colossus transparently manages chunk replication and/or encoding, scalably updates filesystem metadata, and ensures adherence to consistency semantics during concurrent writes and appends. Clients access Colossus through a client library which obtains metadata from Colossus and directly accesses chunks, or parts of chunks, from *D* servers.

RamStore, a distributed in-memory file system (similar to [29]) built upon *memory hosts*, is primarily used to stage data during a *shuffle* [10]. Database grouping and join operations frequently use this form of collective communication, and an in-memory file system enables fast shuffles.

At the top of the stack, *QuerySys*, an internal query processing system similar to [5, 27, 36, 44], provides a query interface to distributed storage. *QuerySys* stores persistent data in Colossus, and uses *RamStore* to stage intermediate data during query execution. Multiple such systems with varying APIs and features exist, including Bigtable [18], another database service built on the same infrastructure.

3 Hotspot Characterization

In this section, we characterize the prevalence and duration of network hotspots in Google's network, discuss their causes, and present examples of their impact. We begin by defining hotspots and describing our methodology.

3.1 Methodology and Metrics

To characterize hotspots, we use telemetry data from switches, which provides the average utilization of each network link over 5-minute intervals. A link is a *hotspot* in an interval if its utilization during that interval is higher than 75%. Since a hotspot might occur on either of the two directions of a given link, we treat each direction separately in our characterization.

We choose a 75% threshold for two reasons. First, we [32, 41, 50] and others [14, 39, 47] maintain a 25% capacity headroom to minimize application impact during maintenance, upgrade, and failure mitigation. For this reason, we use a 75% threshold to determine a hotspot. Second, applications we study in this paper exhibit significant latency increases when network utilization exceeds this threshold (§4.1).

3.2 Prevalence

Figure 3 shows the percentage of hot links (i.e., the *hotspot rate*) across three types of network links in different cross-sections: host ↔ ToR, ToR ↔ aggregation block, and links within the DCNI, aggregated from the link utilization data over a month across the entire fleet.

ToR uplinks have the highest hotspot rate. As we have observed that ToR ↔ aggregation block links are ∼10x more likely to be hot than host ↔ ToR links and ∼2x more than DCNI links, ToR ↔ aggregation block hotspots have the highest traffic impact and are the main focus of this paper. Moreover, ToR hotspots are evenly divided across upstream and downstream directions (results omitted for brevity), so solutions cannot focus on a specific traffic direction. In what follows, the term hotspot always refers to a ToR uplink (i.e., ToR ↔ aggregation block) hotspot in either direction unless otherwise stated.

ToR hotspots span a large range of durations. Figure 4 shows the distribution of the duration for all ToR hotspots in a month. Here, the *duration* of a hotspot is defined as the total length of successive 5-minute time intervals during which the ToR uplink is consistently hot. About a third of the hotspots last less than 1 hour, while nearly 40% of them last 1–12 hours. Other timescales show non-trivial persistence as well; for example, about 13% of hotspots last between 1 day and 2 weeks. This wide range of persistence suggests that a variety of approaches might be necessary to address hotspots: reactive job migration at smaller time scales (minutes), proactive placement of data and network-intensive long-running jobs at medium timescales (hours), and capacity planning and re-provisioning at longer timescales (days or longer).

Storage traffic accounts for most ToR hotspots. Figure 5 shows the distribution of hotspots across different types of racks in the same month. *Mixed* racks, which contain both storage and compute (§2), account for nearly 50% of hotspots,

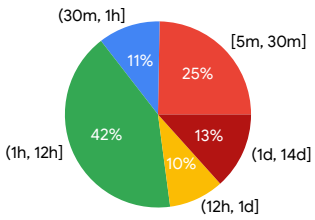


Figure 4: The distribution of the duration of hotspots.

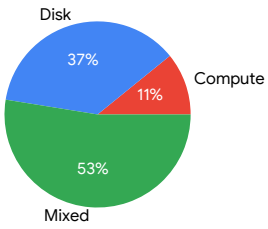


Figure 5: The distribution of hotspots by rack type.

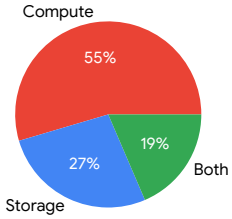


Figure 6: The attribution of hotspots to different workloads on *Mixed* racks.

and *Disk* racks for nearly 40%. Thus, between them, racks with storage devices account for about 90% of hotspots.

To understand which type contributes more to hotspots on *Mixed* racks, we first calculate the share of the capacity provisioned for storage and compute for a given rack based on the configured CPU and disks on the rack. Then, we attribute a hotspot on a *Mixed* rack to storage (compute) if the average throughput of storage (compute) traffic divided by the corresponding provisioned capacity is larger than 75%. From Figure 6, both compute and storage jobs contribute significantly to hotspots on *Mixed* racks; nearly 45% of hotspot traffic on a *Mixed* rack can be attributed to storage.

Taken together with Figure 5, this data suggests that storage traffic contributes significantly to hotspots. Prior work [49] also observes that storage read/write traffic constitutes 75% of the application traffic in Google's datacenters.

3.3 What Causes ToR Hotspots

ToR hotspots occur in our network because of a combination of two factors: *bandwidth demand-supply imbalance* on racks, and *bandwidth-independent task/data placement* by cluster schedulers and distributed storage systems.

Rack bandwidth demand-supply imbalance. Datacenter network planning involves *balancing* ToR bandwidth supply with anticipated traffic demand. For compute racks, bandwidth provisioning targets a specific bandwidth per normalized compute unit. For storage racks, link capacity is provisioned based on HDD capacity on the rack.

Over time, as datacenter ecosystems evolve, demand and supply can diverge from targets, resulting in *bandwidth demand-supply imbalance*. On a compute rack, new network-intensive applications with unprecedented network demands, efficiency improvements in existing applications, or server hardware refreshes can increase bandwidth demand beyond design targets and result in *compute bandwidth demand-supply imbalance*. On a storage rack, increases in installed storage capacity, or incremental replacement of existing or failed disks with faster ones, can increase storage supply without increasing bandwidth supply proportionally, likewise increasing bandwidth demand beyond design targets and resulting in *storage bandwidth demand-supply imbalance*.

These imbalances can persist for days, weeks, or months until network operators restore balance. We argue that these imbalances are inherent to cost-effective datacenter management. It is easier to upgrade or replace a few servers or disks on a rack, but harder to upgrade ToR capacity, since the latter requires taking all ToR-hosted compute and storage offline, and may require aggregation block upgrades as well.

Bandwidth-independent task and data placement. By themselves, rack resource imbalances do not necessarily cause ToR hotspots. ToR hotspots arise, additionally, because our cluster scheduler and our distributed file system make *bandwidth-independent task/data placement* decisions. This was a deliberate choice. At the scale of our datacenters, designing bandwidth-aware cluster scheduling and file systems while also satisfying performance, reliability, and utilization objectives was (and is) considered extremely hard. We have instead attempted to provide the illusion of an unconstrained network to these systems (§1).

As a result of bandwidth-independent task/data placement, these systems can create ToR hotspots. Consider our cluster scheduler, Borg, which schedules, on compute racks, worker tasks belonging to storage systems like *QuerySys*, *RamStore* and Bigtable. Making bandwidth-independent task placement decisions, Borg can schedule a large number of these network-intensive workers on a single rack, resulting in significant traffic demand and triggering a hotspot. Figure 7 shows the distribution of bandwidth per compute unit across all the jobs on our fleet. The distribution is wide, with some network-intensive jobs using more than 10× the median value. Figure 8 plots ToR utilization as a function of the ratio of the number of all network-intensive tasks, including those from *QuerySys*, *RamStore* and Bigtable, per unit network bandwidth for each rack. There is a clear correlation between utilization and the number of these tasks per bandwidth unit.

Bandwidth-independent data placement in a distributed file system can place a large number of storage chunks on racks. When applications access these chunks concurrently, bandwidth demand can approach or exceed supply, and ToR hotspots can arise. Figure 9 quantifies the storage bandwidth demand-supply imbalance in our network; the normalized amount of link capacity divided by HDD capacity for *Disk* racks across our fleet exhibits a more than 2× difference between the smallest and largest values.

3.4 Hotspot Incidents

Network hotspots can cause significant production incidents, including SLO (service-level-objective) violations, user expe-

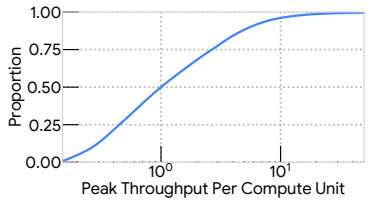


Figure 7: The distribution of outbound peak network throughput per compute unit for jobs across the fleet with non-trivial network usage, normalized by the median value.

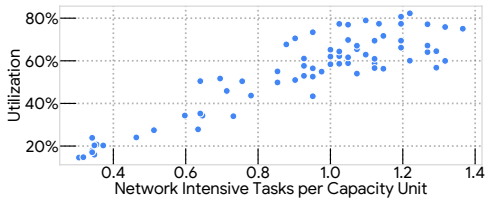


Figure 8: The correlation between ToR utilization (maximum of outbound and inbound directions utilization) and the number of network-intensive tasks on a *Compute* rack per unit of ToR capacity. The x-axis is normalized by the median value. ToRs with a higher number of network-intensive tasks run hotter.

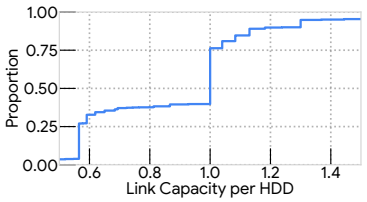


Figure 9: The distribution of the link capacity per HDD capacity for all *Disk* racks across the fleet, normalized by the median value.

rience degradation, and can require extensive troubleshooting effort. This section details several hotspot-related incidents.

Unintended capacity reduction causes ToR hotspots. Temporary hotspots can occur from reduced ToR uplink capacity during maintenance or failures. In a specific instance, a software rollout issue during network maintenance caused a 50% uplink capacity reduction on multiple ToRs. Unaware of this reduced capacity, Borg scheduled too many network-intensive jobs, including crucial batch jobs for Ads analysis, based only on compute resources. This demand-supply mismatch resulted in a 2–3x increase in task completion time.

Unintended network imbalances from scheduling constraints. While serving their intended purpose, Borg scheduling constraints can inadvertently create network imbalances. For instance, an I/O-intensive service implemented a custom constraint to avoid scheduling workers on machines with D servers, aiming to reduce network congestion. However, this constraint unintentionally excluded the majority of available machines, concentrating a large number of these same workers onto the remaining servers. This concentrated load resulted in persistent ToR hotspots and a 1.5x increase in read latency. This incident underscores the need for a more comprehensive approach to mitigating network hotspots.

Colossus degraded reads exacerbate ToR hotspots. During reads, a Colossus client initially attempts to read data directly from the disks storing the required chunk(s). If one or more of these are temporarily unavailable (due to overload on the disk, CPU, or network), and the file is erasure-encoded, the client performs a degraded read: retrieving the other chunks in the erasure-coded stripe and reconstructing the unavailable chunk. However, this mechanism is ineffective when the client resides on a ToR hotspot — it increases network demand on an already congested link. For example, in one incident, ToR congestion caused by incast traffic to Bigtable led to high Colossus read latencies. The resulting degraded reads significantly exacerbated ToR congestion.

Storage ToR hotspots cause high-level applications pain. There are many applications that are built on top of storage services. Therefore, hotspots impacting storage services ultimately translate into impact on users. For example, YouTube relies on storage services (e.g., Bigtable) to store, retrieve and display watch history. In one incident, a ToR hotspot led to increased packet losses and slowed reads from an SSD-backed Bigtable partition. As a result, Bigtable replication latency of user data doubled, resulting in stale or missing watch histories for some users, impacting user satisfaction.

4 Impact of Hotspots on Storage Systems

In this section, we present measurements of the impact of hotspots on the performance of our distributed storage systems. Because ToR hotspots occur predominantly on racks with storage elements, they impact the performance of access to distributed storage. In our infrastructure, applications access distributed storage via queries to a distributed database (*QuerySys*), or by performing reads/writes on a distributed file system (Colossus). In this section, we study how ToR utilization affects the latency of *QuerySys* query completion and Colossus file reads and writes.

4.1 Approach and Methodology

At a high level, to understand the impact of ToR utilization on storage access, we associate the latency of a *storage operation* (a file system read/write, or a database query) with the ToR utilization observed during the operation.

Latency. We use extensive logs to measure the latency of each storage operation, relying on production workloads for measurements of file system accesses and common benchmarks for database queries. These systems are instrumented to break down the time spent on the server versus on the network. The total operation time is measured from initiation to response, including network latency, which encompasses the time for underlying Remote Procedure Calls (RPCs) to traverse the network (including ToR switches).

Utilization. Each file system read or write, and each query, produces a *latency sample*. To understand how these latency measurements depend on ToR utilization, we associate each latency sample with a ToR utilization value. We do this as follows. A file system read/write, or a database query involves many ToR uplinks and downlinks, since these operations may involve multiple servers. We associate each latency sample with the maximum utilization of all ToR links involved in a storage operation, since that lies on the critical path of

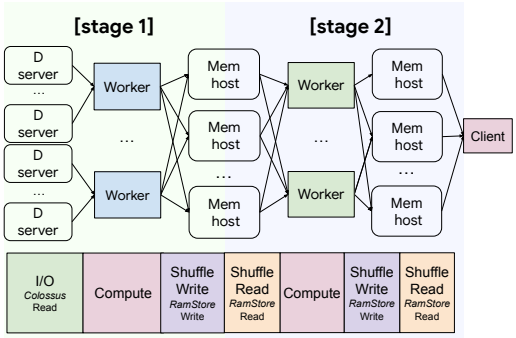


Figure 10: The data flow for an aggregation query in *QuerySys*.

an operation's completion. We obtain network utilization from a network telemetry system which reports the average throughput of each ToR during each 30 second window. For each latency sample, we use the average utilization during the time window containing the operation.

Metrics. To understand and quantify how storage access latency is affected by utilization, we divide utilization values into buckets in increments of 5%. For each bucket, we compute the *p95* latency of the samples within that bucket. We define the p95 *latency inflation* at a given utilization bucket as the ratio of the p95 latency in that bucket to the p95 latency of the lowest utilization bucket for which we have data. Latency inflation characterizes how storage access latency is affected by increasing ToR utilization.

We use a tail measure (p95) because tail performance is critical for many of our applications. We have also measured median latency inflation (defined similarly). This shows qualitatively similar results, so we omit this for brevity.

To understand how ToR utilization inflates storage access latency, we define *Load-tolerance*: the utilization at which p95 (or median) latency inflation is τ. In this paper, we consider two values of τ, 1.5 and 2. At $\tau = 2$, the storage access latency is double that at the lowest utilization which we consider unacceptable. For simplicity, we currently define unacceptable latency based on this fixed threshold. In future work (§6), we plan to define it based on application latency SLOs, which directly characterize the sensitivity of storage access to utilization.

4.2 Database Queries

A significant component of the offered load in our datacenters is traffic from queries of a distributed database, *QuerySys*. Query execution consists of a series of stages, and each stage runs multiple *workers* in parallel. Each worker (running on compute racks) processes a *partition* by reading data initially from a distributed file system, Colossus (or, in later stages, from an in-memory file system *RamStore*), performing some computation, and writing data to *RamStore*. Figure 10 depicts this for an *aggregation* query which produces statistics across one or more columns in a large relational table.

As discussed in §3, imbalanced infrastructure upgrades or network-intensive workloads can result in a *network/compute imbalance* that can increase ToR utilization and, in turn, affect query completion times. When infrastructure used for databases is upgraded such that compute on a rack increases (*e.g.,* because servers are upgraded) without a corresponding increase in ToR capacity, ToR utilization can increase. Database workload network-intensiveness can grow in several ways: *e.g.,* the rate of queries can increase, queries can become more complex, and queries can process larger tables; any of these factors can require more workers. More workers concurrently accessing distributed storage from a rack can increase ToR utilization.

In this section, we seek to quantify how the end-to-end query completion time, as well as the latency of each read or write operation, is affected by ToR utilization. When a query is issued, its completion time is affected by the concurrent workload (other queries and other application traffic) using the network. Some queries may traverse highly utilized ToRs. This gives us an opportunity to study how database queries are affected by ToR utilization.

Methodology. While most database queries execute in stages (*e.g.,* Figure 10), they differ widely in terms of the number of stages they use, the number of workers they employ, and the tables they access. We cannot thus assess ToR utilization impact on a single query, because each query may be affected to different extents at different utilization levels.

We use seven *benchmark* queries that our *QuerySys* team runs periodically to assess performance. These benchmarks include canonical queries (aggregations, materializations, etc.) that reflect real-world usage patterns, as well as industry standard benchmarks (*e.g.,* TPC-H and TPC-DS [46]), and thus capture the shape and semantics of most common queries against *QuerySys*. The breakdown of latency at each worker and each stage is logged for these benchmarks. This enables us to understand and quantify the extent to which network, compute, and storage affect end-to-end performance.

The rest of this section presents the impact of ToR utilization on these benchmarks.

Aggregation is the most common query in *QuerySys*: computing aggregates (*e.g.,* sum, count, average) from one or more columns in a large relational table. To scale to large tables, *QuerySys* employs multiple workers each of which processes distinct partitions of the table. This query executes in two stages (Figure 10). Both stages involve networked storage access and computation.

In this query, networked storage (Colossus) access account for approximately 30% of the query's total time. The time difference between Colossus reads and the query total time is primarily computation time, representing roughly 50% of the query's total time.

Figure 11 shows the p95 latency inflation and the varying *Load-tolerance* of these operations for $\tau = 1.5$: The *Load-tolerance* of the end-to-end query latency can be quite high (85%). For this query, the component which accounts for half the overall query latency, the computation time, is largely

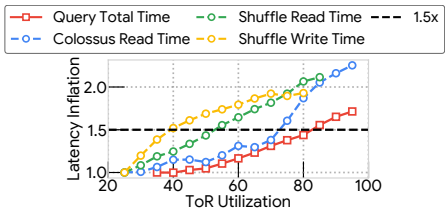
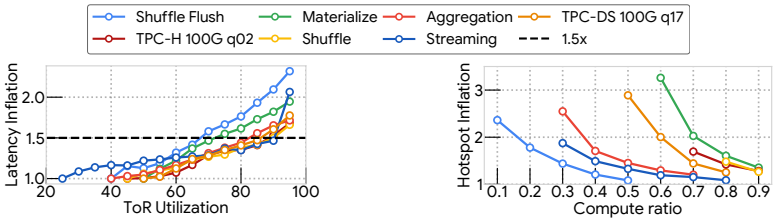


Figure 11: The query latency inflation vs. ToR utilization for the aggregation query.



(a) The query latency inflation vs. ToR utilization. **(b)** The CPU ratio of a query vs. the latency inflation at utilization 75%.

Figure 12: The query latency inflation of different *QuerySys* benchmarks.

independent of network hotspots, resulting in a higher *Load-tolerance*. The other components of the storage stack have a lower and widely varying *Load-tolerance*: shuffle writes at 40%, shuffle reads at 52%, and Colossus reads at 72%. Shuffle read/write have lower *Load-tolerance* because they don't involve compute and storage operation on the *RamStore* hosts. Colossus read is slightly higher because it involves storage but minimal compute.

Figure 12a shows p95 latency inflation and Table 1 shows *Load-tolerance* of the other queries in the benchmark. **Shuffle flush** is most sensitive to ToR utilization, primarily because it triggers an intensive data-flush from *RamStore* memory hosts to Colossus when the host runs out of memory to complete the write. This operation increases the share of network transfer time in the total query time, resulting in the lowest 1.5× *Load-tolerance* among all benchmarks (70%). **Materialize** writes results through Colossus; its *Load-tolerance* of 75% is similar to that of shuffle flush and Colossus reads during aggregation. The remaining benchmark queries are less network-intensive, so have a *Load-tolerance* of 90–95%.

The impact of computation. Queries access storage, but also process the retrieved data (*e.g.,* aggregate the data). This processing time affects the tolerance of a query to hotspots.

To illustrate this, we defined *Hotspot-inflation*: the latency inflation at the hotspot threshold (75% utilization, §3). We explored *Hotspot-inflation* for different queries as a function of *query compute ratio* — the ratio of the sum of the processing time for the 10 slowest workers[2] within each stage by the sum of their worker time (aggregated across all stages).

Figure 12b depicts the relationship between these two quantities. To understand this figure, it helps to understand that different samples from a single benchmark query may exhibit different compute ratios (because storage operations and networking take different amounts of time). The figure groups samples by compute ratio, and plots the *Hotspot-inflation* for each group. Thus, for example, samples of the shuffle flush benchmark which have a compute ratio of 0.1 have a *Hotspot-inflation* of about 2.25. The figure demonstrates a clear negative dependence between compute ratio and *Hotspot-inflation*. This explains why higher compute ratios result in higher *Load-tolerance*s. In other words, the more compute-bound a query is, the less sensitive it is to network hotspots.

[2]Currently, our telemetry system tracks processing times only for the 10 slowest workers.

	Load-tolerance (1.5×)	*Load-tolerance* (2×)	*Hotspot-inflation* (75%)
Shuffle Flush	70	90	1.67
Materialize	75	–	1.55
Aggregation	85	–	1.38
TPC-DS 100G q17	90	–	1.35
TPC-H 100G q02	90	–	1.34
Shuffle	95	–	1.29
Streaming	95	95	1.37

Table 1: *Load-tolerance* and *Hotspot-inflation* of different *QuerySys* benchmarks.

Different queries have different *Hotspot-inflation* at the same compute ratio because they may, for example, use different storage operations. For example, when the CPU ratio is 0.8, *materialize* prober has more *Hotspot-inflation* than *streaming* prober because the former incurs Colossus writes while the latter incurs Colossus reads. For Colossus, writes are more sensitive to ToR hotspots than reads (§4.3).

Summary. In Table 1, we compare the *Load-tolerance* and *Hotspot-inflation* across queries. Most queries also have high *Load-tolerance*; many did not reach 2× inflation. For these, their high *Load-tolerance* comes from the significant time spent on compute — ranging from 10% to 90% (Figure 12b).

This discussion implies that for some *QuerySys* benchmarks substantial tail latency reductions can be had by reducing the occurrence of ToR hotspots. §5 discusses hotspot-aware placement mechanisms we use to achieve that.

4.3 File Access

In a distributed file system like Colossus, files are split into chunks, with each chunk replicated across multiple servers. Consider a rack with servers that host chunks. If these servers are upgraded to add more storage I/O capacity (e.g.,new storage hardware that increases the storage read/write rate), but the ToR's capacity is not correspondingly upgraded, a network/storage imbalance can result. This imbalance can result in a ToR hotspot, which can, in turn, affect the latency of reads and writes to the file system. In this section, we (a) study how Colossus read/write latency is impacted by ToR hotspots and by ToR utilization, and (b) quantify the prevalence of network/storage imbalance across our fleet.

Methodology. We collect the total latency of datacenter HDD chunk read and write requests on racks with HDD stor-

age (§2.1) with chunk sizes between 100KB and 1MB from the Dapper tracing system [40] from production workloads over one day.

HDD reads. Figure 13a shows the p95 latency *inflation* (ratio against latency at lowest utilization, §4.1) for network and total latency. Both network and total latency increases with ToR utilization. At the 75% hotspot threshold, network latency is inflated by 4×, but Colossus HDD reads have a *Hotspot-inflation* (the inflation at the hotspot-threshold of 75%, §4.1) of only 1.5×. This is because only the network portion of the Colossus read time is inflated. In all but the last utilization bucket, network accounts for less than 20% of the total HDD read latency; in the last utilization bucket, network doesn't contribute more than 40%. *Load-tolerance* (§4.1) is another way to depict the impact of ToR utilization on storage access latency. For Colossus reads, the 2× *Load-tolerance* is 95%; this means that read latency is less than 2× for all ToR utilizations lower than 95%.

HDD writes. Colossus HDD writes show a qualitatively different behavior than HDD reads because Colossus employs write-back caching on *D* servers (§2.2): writes are cached in battery protected server-side memory and later flushed to persistent storage. Thus Colossus HDD write latency is dominated by network latency, and hence is impacted significantly by increases in network latency as a result of high ToR utilization. Network contributes to 30–40% for low utilization buckets and more than 50% for higher utilization buckets (>75%). By contrast, though Colossus caches reads, read cache hit rates are low, because our workloads mostly access large files sequentially, resulting in little read locality. Hence similar behavior is not observed on reads.

Writes are very sensitive to ToR utilization. As Figure 13b shows, their 2× *Load-tolerance* is a mere 50%, so, even in a lightly loaded network, write latency can be twice the latency when compared to an unloaded network. Their *Hotspot-inflation* is around 4; writes to a rack with a ToR hotspot can be 4 times slower than writes to a rack with an unloaded ToR.

While HDD write operations are impacted more than reads due to the amount of storage access, the network portion of the latency is similarly affected in both operations. The network latency components include round-trip delay (RTT), serialization delays, and congestion control shaping traffic to fill the ToR link capacity. Measurements of RTT and packet losses (not shown here) remain consistently low across utilizations as expected from our deployments of congestion control and load balancing techniques [22, 25]. The shape of the network latency curve reflects the workload's inherent burstiness relative to the available capacity.

The impact of imbalanced infrastructure. Having characterized the impact of network utilization on reads and writes, we now show how network/storage imbalance impacts file reads and writes in Figure 14. To derive this figure, we classified each rack in a datacenter into one of three categories based on its level of resource balance: *High-Uplink* if its installed ToR uplink capacity exceeds provisioning guideline relative to installed storage, *Medium-Uplink* if the capacity matches the guideline, and *Low-Uplink* if the capacity is lower than the guideline. Figure 14 shows the fraction of storage reads and writes to each type of rack across different ToR utilizations, across all of our datacenters. The figure dramatically illustrates the impact of network/storage imbalance. Storage requests that experience low ToR utilization access *High-Uplink* ToRs (in green). Conversely, storage requests that experience extremely high ToR utilization almost exclusively access *Low-Uplink* ToRs (in red).

5 Hotspot-Aware Placement

As discussed in §3, ToR hotspots occur as a result of an interplay between network supply and demand for compute and storage resources. As such they cannot be resolved by *in-network* mechanisms *e.g.,* in switches or host networking stacks. Therefore, we integrate *hotspot awareness* into our cluster scheduler and distributed file system. We've specifically developed two hotspot-aware placement strategies:

1. *ToR-utilization aware task placement and migration* (UTP) takes ToR utilization into account when placing tasks (such as query workers §4) on servers, or migrating tasks away from them, thereby improving the balance between network demand-supply for compute resources.
2. *ToR-capacity aware chunk placement* (CCP) takes the imbalance between ToR capacity and rack storage capacity into account when making storage chunk placement decisions. Thereby improving the balance between network demand-supply for storage resources.

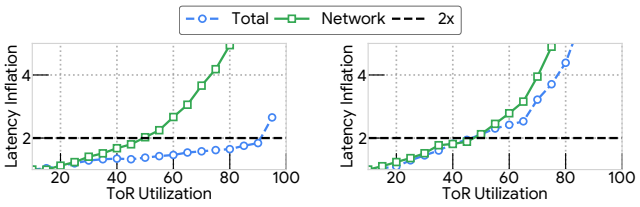
As we describe below, the design of these hotspot-aware placement strategies is informed by the measurements presented in §3 and §4. Moreover, neither of these placement decisions is trivial, since each needs to balance complex, competing objectives. At the same time, avoiding hotspots is important, since doing so can potentially improve network utilization and applications' perceived latency.

5.1 Task Placement and Migration

We modified the Borg cluster scheduler to bias task placement away from, and reactively migrate tasks from, ToR hotspots. Before describing this technique, abbreviated UTP, we briefly describe Borg; see [9, 45] for details.

Background. Borg selects the server on which to place each task (e.g., worker) in a job. It utilizes historical measurements to estimate task resource requirements, and uses this to select suitable servers. When an estimate is inaccurate or workload changes, Borg can reactively migrate affected tasks to improve job performance, but task migration must be careful to avoid impacting availability SLOs offered to service owners.

In practice, a cluster scheduler must (a) scale to clusters with a large number of servers and (b) satisfy many other, often competing, objectives. It must: balance load between servers, prevent over-subscription to resources such as CPU



(a) HDD read. **(b)** HDD write.

Figure 13: The total latency and the network latency inflation (*i.e.,* p95 latency values normalized against the latency at the lowest utilization).

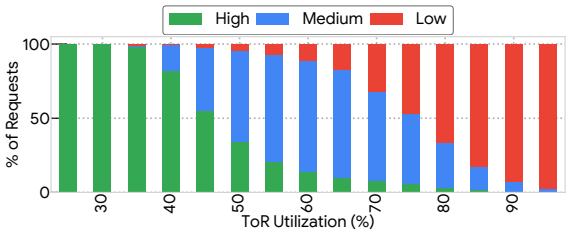


Figure 14: The distribution of chunk read and write request by uplink categories (*High-*, *Medium-* and *Low-Uplink*) for ToRs at a given level of utilization.

or memory, over-commit resources to take advantage of statistical multiplexing, spread tasks of a job across racks to minimize the likelihood of correlated failure, enable memory access locality, and pack jobs into as few servers as possible.

Borg addresses the first challenge by choosing a random sample of placement candidates from amongst servers in the cluster, rather than considering all servers. It addresses the second challenge by assigning a vector of scores to each server based on multiple objectives. Borg prioritizes server selection objectives, with lower-ranked objectives taking precedence. For instance, if load balancing is deemed more critical than bin packing, Borg may choose a server that optimizes load distribution even if it leads to suboptimal bin packing efficiency.

Requirements. Before we designed and implemented UTP, Borg made *bandwidth-independent task placement decisions* — its decisions did not take network utilization into account. Its primary task — to find a set of servers to satisfy application requirements and potentially competing infrastructure constraints — was complex enough that no attempt was made to add network as a resource to Borg earlier, since that would require imposing constraints on group of servers beneath each ToR rather than individual servers. Given that hotspots can degrade performance, and that Borg is a mature infrastructure, we imposed two requirements on UTP: (a) it should be designed as a minimal change to Borg (i.e., not requiring network as a full-fledged new resource type), and (b) it should not adversely impact any existing cluster objective.

Design. To make minimal changes to the scheduler, ideally we should minimally modify an existing objective so that it does not regress any other existing objective. This is hard to do for ToR utilization. Borg can easily determine if placing a task on a server would over-run the server's CPU resources; this depends only on the resources currently committed by the server to other tasks. It is harder to determine how placing a task on a server would impact the server's ToR utilization, since that depends on *other* servers on the rack (that may or may not be in the randomly chosen candidate set) and how their workload changes. Reasoning about this impact correctly would require Borg to consider a large number of servers, which can impact scaling.

Proactive UTP placement. Fortunately, UTP required a relatively simple change to Borg for the following reason: at any given instant, there are relatively few ToR hotspots in each cluster, since the average ToR utilization in our datacenters is low. Given this, UTP tweaks a low-priority objective in Borg, that of ensuring load-balance across servers: it prefers servers that would balance ToR-utilization across the cluster more evenly. Specifically, UTP achieves this by assigning a score to each candidate machine that reflects the ToR utilization for that machine should the task get placed on that machine. In computing this score, we take into account instantaneous ToR utilization as well as peak task demand.

This methodology works well even on a random sample of servers; given the low average ToR utilization, there are likely to be many servers in the sample with low ToR utilization, so in most of the cases Borg can satisfy other cluster-level objectives while finding a suitable candidate server which also improves better balancing of ToR utilization.

Reactive UTP migration. Even with ToR-utilization aware task placement (UTP), ToR hotspots can develop. This is because the placement decision is based on historical estimates, which can be incorrect as the usage pattern of a workload changes, or may not be available for new workloads. To address this, UTP migrates tasks away from ToR hotspots. Borg already supports the infrastructure to orchestrate this for other dimensions (e.g., cpu, memory bandwidth, etc) and we extended it toward network. To this end, UTP must make two decisions: (a) how to determine if a ToR is a hotspot, (b) which task(s) to migrate and when.

UTP employs a 75% utilization threshold to identify hotspots. This aligns with the load tolerance of common workloads, including those discussed in §4.2 and Colossus reads up to 1MB. It also adheres to the fleet-wide requirement of reserving 25% capacity for in-place upgrades and expansions. It then greedily finds the latency-tolerant tasks with the largest network bandwidth usage on a rack to migrate. This ensures the fewest possible migrations and avoids migrating latency-sensitive tasks when possible. Finally, UTP migrates tasks only if availability budgets would permit.

Results. Figure 15 shows the efficacy of UTP in addressing ToR hotspots after its fleetwide rollout. Specifically, it shows that the number of hot ToRs was reduced by 90% after deploying these features in the scheduler. The remaining hot

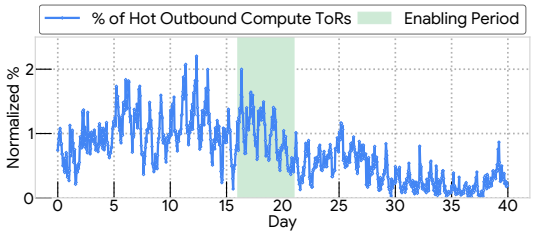


Figure 15: The fleet-wide percentage of compute ToRs hot in the outbound direction before and after deploying UTP, normalized by the average value before the deployment.

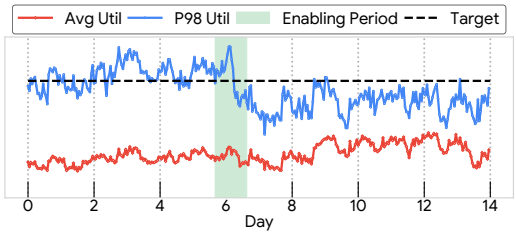


Figure 16: The change of the p98 outbound utilization of compute ToRs as well as the average cell level utilization, defined as the total egress throughput from compute ToRs divided by their total uplink capacity, after UTP was enabled in a pilot cluster.

ToRs are likely due to cases where higher-priority objectives in task placement would prefer a hot ToR rather than cold ToRs. As expected, given the low average ToR utilization in our datacenters, such cases are not common. Future work can consider ways to improve scoring and the relative order of objectives to further reduce the occurrence of hot ToRs.

Figure 16 shows a time series of the evolution of the p98 outbound utilization of *Compute* ToRs during the pilot of UTP in one of our clusters before the fleetwide rollout. It also depicts a timeseries of the average cell level utilization in the cluster (§5.2) during that same window. UTP reduces the tail of the ToR utilization to below the hotspot threshold (75%) after the deployment of UTP. This benefits comes despite a higher average utilization due to an unrelated workload change after UTP was enabled.

In theory, UTP could have been purely reactive: it could have omitted ToR-aware initial placement, and merely invoked task migration when a hotspot was detected. Figure 17 shows that a purely reactive design can increase task migrations by nearly 2×, relative to our design. Moreover, with a purely reactive approach, network-intensive jobs are 7× more likely to be scheduled on hot ToRs (Figure 18).

UTP enables significant reductions (up to 13%) in p95 latency across many of our *QuerySys* benchmarks (Figure 19). The shuffle flush and materialize benefit the most from UTP because they are the most sensitive to ToR utilization with the lowest *Load-tolerance* (Figure 12a). On the other hand, the improvement on the TPC-H benchmark is not significant, which is expected because the workload is not sensitive to ToR utilization.

As discussed in section §3.4, ToR hotspots were a recurring cause of service incidents. Enabling UTP resulted in a 70% reduction in the monthly occurrence of such incidents. The remaining incidents post-UTP were primarily attributed to either resolved bugs within UTP or occurred in specific clusters where UTP had not yet been deployed.

Metric	Change
Percentage of hot ToRs	−44.6%
p98th utilization	−18.5%
Average utilization	10.9%

Table 2: The change of the median value of the metrics reported in Figure 15 and Figure 16 in a 5-day window before and after UTP was enabled. We also used the Student's t-test on the metrics samples in the two periods to confirm that the change was statistically significant.

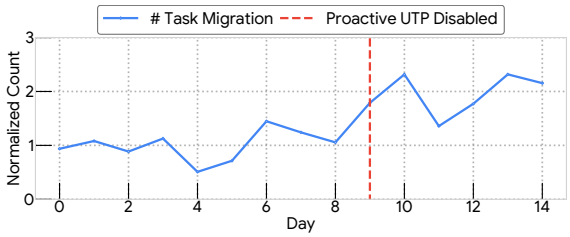


Figure 17: The number of task migration events in a day due to UTP before and after the disabling of the proactive UTP placement, normalized by the average value before.

Lastly, as discussed earlier, a key constraint that we needed to satisfy in the design of UTP was to not adversely impact any existing cluster objective. We didn't observe any regression in key Borg objective after the rollout of UTP. Corresponding graphs are omitted for brevity.

5.2 Storage Chunk Placement

ToR-capacity aware chunk placement (CCP) addresses imbalanced infrastructure upgrades by steering Colossus chunk placement away from racks whose storage capacity has grown out of balance with provisioned ToR bandwidth.

Background. Distributed file systems use heuristics to determine where to place new file chunks. These heuristics are designed to achieve several, often competing, objectives to determine the most appropriate storage servers at which to replicate these chunks. These objectives are similar to that of task placement: spreading for reliability, load balancing, access locality, and so on.

Design. As with UTP, CCP was designed to ensure minimal changes to the existing Colossus distributed file system. These changes were similar in principle to those for UTP, so we omit the details for brevity but describe the key idea underlying the approach.

In §4.3, we showed that storage requests to *High-Uplink* ToRs experience low ToR utilization. Conversely, storage requests that experience extremely high ToR utilization almost exclusively access *Low-Uplink* ToRs. Based on this observation, CCP prioritizes High-Uplink ToRs over Medium-Uplink and Low-Uplink ToRs for new chunk placement.

Results. We now discuss latency reduction resulting from

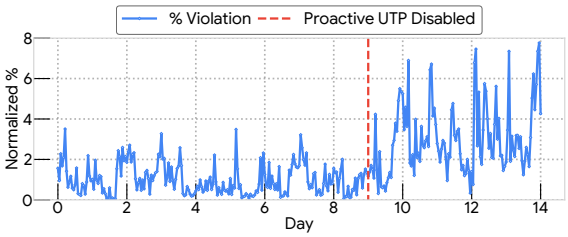


Figure 18: The percent of job scheduling events where a network-intensive job is scheduled on a hot ToR before and after the disabling of the proactive UTP placement, normalized by the average value before.

Metric	Change
Number of task migration events	−41.0%
Percentage of scheduling violations	−74.1%

Table 3: The change of the median value of various metrics reported in Figure 17 and Figure 18 in a 5-day window with ToR utilization-aware placement enabled against in a 5-day window without the feature. We also used the Student's t-test on the metrics samples in the two periods to confirm that the change was statistically significant.

this approach, using measurements from a 15-day pilot in one of our clusters. Figure 20 shows the relative reduction in Colossus p95 total storage access and network latency in the pilot cluster. As CCP biases storage accesses towards high-capacity ToRs, p95 network latency reduces by 50–80% and p95 total latency reduces by 30–60%.

6 Future Directions

Automation techniques for hotspot mitigation. Hotspot mitigation in our storage systems required significant manual intervention, leveraging service owner knowledge of application metrics, dependencies, and request prioritization within the application. To scale mitigation to more applications, automation techniques must be developed. Future work can explore declarative methods for service owners to define application characteristics, dependencies, and constraints, enabling automated backend systems to identify and implement mitigation opportunities. Alternatively, more robust profiling techniques can extract network communication requirements from live deployments and map them to a range of system deployment conditions.

Service Level Objectives (SLOs). Our current placement strategy, which is hotspot-aware but best-effort, cannot guarantee performance under every network and workload condition. To ensure consistent performance, we must define SLOs and implement mechanisms to guarantee them. This necessitates substantial scheduler enhancements, moving beyond simple load spreading to include strict network constraint enforcement, thereby mitigating SLO risks.

Network fault tolerance for ML. The distinct characteristics of Machine Learning (ML) workloads, such as single points of failure and sensitivity to worker performance, necessitate novel network fault tolerance strategies. Placement and rescheduling techniques discussed in this paper can be

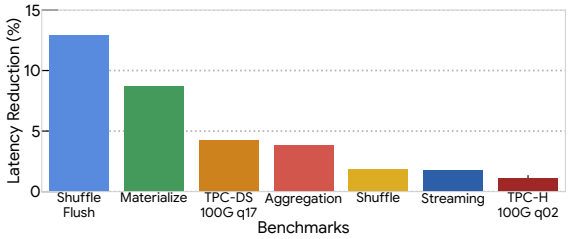


Figure 19: The relative improvement of the p95 query latency for each of the seven *QuerySys* benchmarks studied in Section 4.2 in a 5-day window before and after UTP was enabled in the pilot cluster.

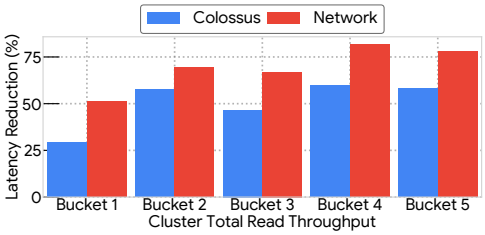


Figure 20: The reduction of the p95 Colossus and network latency of HDD reads in a pilot cluster with CCP enabled, compared with those at the time when the cluster is in the same bucket of total read throughput but with CCP disabled.

adapted for ML workloads to minimize the impact of faulty network components, such as NICs and ToRs. This is particularly crucial for jobs that rely on synchronous communication between numerous accelerators.

Network-awareness across layers. While we tackled persistent network hotspots with infrastructure-level network awareness, shorter-lived hotspots (seconds to few minutes) can make use of application-level load balancing. Application control further allows for customized hotspot handling based on request sensitivity and required quality of service. Future investigations can focus on determining optimal hotspot resolution timescales for various mitigation strategies (proactive/reactive) and layers (infrastructure/application), and developing coordinated decision-making frameworks.

Connecting placement and provisioning. Future work can explore the interplay between hotspot-aware placement and resource (re-)provisioning. While hotspot-aware placement can extend the lifespan of network infrastructure and allow for more conservative uplink:downlink oversubscription to deliver network efficiency, capacity planners need tools to determine when and where to upgrade network, storage, and compute resources to address capacity mismatches. A key area for future research is determining the optimal utilization target for provisioning and identifying utilization values to inform changes in job and data placement, enabling safely operating network at higher utilization while meeting application SLOs.

7 Related Work

Reducing network congestion. To reduce datacenter congestion, the literature explored various solutions, including congestion control [3, 4, 19, 25, 26, 28, 31, 51], network load balancing [1, 2, 8, 21, 22], and traffic engineering [7, 35]. These solutions mitigate packet drops and queuing delays by distributing traffic over time (congestion control) or across different paths (load balancing and traffic engineering), but they are bounded by bandwidth limits in ToR hotspots in which all the uplinks of a ToR are persistently hot. By contrast, hotspot-aware scheduling makes more bandwidth available, by modifying the traffic matrix.

Datacenter scheduling. Datacenter job scheduling can influence traffic patterns, but it traditionally prioritizes local resources like compute, memory and storage, assuming the network isn't a bottleneck. Various research has explored incorporating network attributes like latency, bandwidth, and reliability into storage systems to make them network-aware [6, 15, 42, 43]. Several storage systems attempt to mitigate network congestion. For instance, Sinbad [12] dynamically adjusts replica placement based on real-time link utilization to address imbalances. However, it is restricted to replicated file encodings and requires live network data, unlike our ToR-capacity aware chunk placement (CCP), which is encoding-agnostic and relies on simpler, static ToR uplink information. NetHint [11] proposes an interactive cloud tenant-provider mechanism to optimize application performance by identifying network hotspots. Yet, this approach necessitates application-level network awareness and is confined to single-tenant environments, contrasting with our infrastructure-level, multi-tenant hotspot management. Furthermore, systems like Diktyo [37] and Kubernetes [13] optimize service placement for latency by considering network bandwidth and topology, primarily focusing on microservice environments.

High-Performance Computing (HPC) and Machine Learning (ML) have long incorporated network awareness into job scheduling, as exemplified by CASSINI [34] and Corral [20]. This paper demonstrates that network awareness is equally critical for traditional compute and storage services, especially in mitigating application tail latency.

Network capacity planning. Recent research has explored leveraging application-specific information to enhance bandwidth allocation: Saba [23], for example, uses ahead-of-time profiling to distribute network bandwidth based on how sensitive applications are to changes in bandwidth. Jupiter's networks [41] employ both traffic engineering and topology engineering, which operate at different timescales. Traffic engineering swiftly adapts to changes in topology and traffic, whereas topology engineering is a slower, planned process for implementing new network structures. However, capacity planning alone cannot eliminate network hotspots due to the fundamental causes discussed earlier (§3.3). Our work focuses on analyzing how sensitive applications are to network hotspots, and how network-aware placement can reduce the occurrence and duration of these hotspots.

Characterizing application performance. Prior research has explored the impact of various factors on application performance. Ousterhout *et al.* [30] proposed a methodology for quantifying performance bottlenecks and used it to analyze the Spark framework's performance, concluding that network optimizations can only reduce job completion time by a median of at most 2%. Their conclusions are not in contradiction to this paper because (1) the network utilization in their datacenters is low and below 50%, so they do not observe the effect of hotspots, and (2) workloads in our datacenters are able to fully parallelize the execution, which magnifies the impact of the worst straggler. Seemakhupt *et al.* [38] analyzed latency bottlenecks in Remote Procedure Calls (RPCs), identifying network performance issues using passively-sampled RPCs [33] at Google. Zambre *et al.* [48] broke down the time involved in small message communication into CPU, I/O, and network usage. More recently, Kong *et al.* [24] modeled how NIC resource usage affects RDMA performance. Our work complements these studies by focusing on how ToR utilization influences performance, ranging from basic storage operations to complex application operations in *QuerySys*.

8 Conclusion

This paper demonstrates the need for hotspot-aware scheduling to mitigate the impact of network hotspots on application performance in datacenters. Traditional solutions like congestion control and traffic engineering prove insufficient; instead, directly managing workload placement is essential for reining in elevated tail latency. Our work reveals two surprising findings: first, even disk- or compute-bound application operations are vulnerable to elevated network utilization levels, with ToR hotspots capable of doubling their latency. Second, adding simple hotspot awareness to existing compute and storage schedulers dramatically reduces application tail latency and eliminates most ToR hotspots. This approach, while simpler than incorporating network as a fully schedulable resource, yields significant gains.

These results highlight the need for datacenter schedulers to comprehensively consider network demand and supply. Future research shall investigate holistic scheduling frameworks accounting for network bandwidth, topology, and compute and storage resources to pave the way for balanced datacenter environments with predictable application performance.

Acknowledgement

This work wouldn't have been possible without the contributions of many individuals and teams across Google, particularly within the Networking Infrastructure, Compute, and Storage organizations. We are especially grateful to to Arjun Singh, David Wetherall, Ian Whalley, Milo Martin, Thomas Wenisch, Larry Greenfield, Gaurav Dhiman, Flo-

rentina Popovici and the anonymous NSDI reviewers for their invaluable feedback and to our shepherd Ravi Soundararajan for guiding us through the revision process. This research was in part supported by NSF CNS NeTS 2211383.

References

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 503–514, New York, NY, USA, 2014. Association for Computing Machinery.

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

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery.

[5] Amazon. Amazon redshift. https://aws.amazon.com/redshift/, 2024.

[6] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. Ca-nfs: A congestion-aware network file system. *ACM Trans. Storage*, 5(4), dec 2009.

[7] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, New York, NY, USA, 2011. Association for Computing Machinery.

[8] Olivier Bonaventure, Christoph Paasch, Gregory Detal, et al. Use cases and operational experience with multipath tcp. *RFC 8041*, 2017.

[9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. Flumejava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.

[11] Jingrong Chen, Hong Zhang, Wei Zhang, Liang Luo, Jeffrey Chase, Ion Stoica, and Danyang Zhuo. NetHint: White-Box networking for Multi-Tenant data centers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1327–1343, Renton, WA, April 2022. USENIX Association.

[12] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 231–242, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Kubernetes community. Network-aware scheduling. https://scheduler-plugins.sigs.k8s.io/docs/kep/260-network-aware-scheduling/readme/, 2023.

[14] Marek Denis, Yuanjun Yao, Ashley Hatch, Qin Zhang, Chiun Lin Lim, Shuqiang Zhang, Kyle Sugrue, Henry Kwok, Mikel Jimenez Fernandez, Petr Lapukhov, Sandeep Hebbani, Gaya Nagarajan, Omar Baldonado, Lixin Gao, and Ying Zhang. Ebb: Reliable and evolvable express backbone network in meta. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 346–359, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Amir Epstein, Elliot K Kolodner, and Dmitry Sotnikov. Network aware reliability analysis for distributed storage systems. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 249–258. IEEE, 2016.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[17] Google. A peek behind Colossus, Google's File System | Google Cloud Blog. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2021.

[18] Google. Bigtable overview. https://cloud.google.com/bigtable/docs/overview, 2023.

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 407–420, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of ACM CoNEXT*, pages 149–160, 2014.

[22] Abdul Kabbani, David J. Wetherall, Gautam Kumar, Junhua Yan, Kira Yin, Masoud Moshref, Mubashir Adnan Qureshi, Qiaobin Fu, Van Jacobson, and Yuchung Cheng. PLB: Congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, New York, NY, USA, 2022. Association for Computing Machinery.

[23] M.R. Siavash Katebzadeh, Paolo Costa, and Boris Grot. Saba: Rethinking datacenter network allocation from application's perspective. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 623–638, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 31–48, Boston, MA, April 2023. USENIX Association.

[25] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Mike Ryan, David J. Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[26] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Microsoft. Azure synapse analytics. https://azure.microsoft.com/en-us/products/synapse-analytics, 2024.

[28] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[29] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[30] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, Oakland, CA, May 2015. USENIX Association.

[31] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. *SIGCOMM Comput. Commun. Rev.*, 44(4):307–318, aug 2014.

[32] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming Google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 66–85, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Mubashir Adnan Qureshi, Junhua Yan, Yuchung Cheng, Soheil Hassas Yeganeh, Yousuk Seung, Neal Cardwell, Willem De Bruijn, Van Jacobson, Jasleen Kaur, David Wetherall, and Amin Vahdat. Fathom: Understanding datacenter application network performance. In *Proceedings of the ACM SIGCOMM 2023 Conference*,

ACM SIGCOMM '23, page 394–405, New York, NY, USA, 2023. Association for Computing Machinery.

[34] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, Santa Clara, CA, April 2024. USENIX Association.

[35] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, aug 2015.

[36] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. F1 query: Declarative querying at scale. *Proc. VLDB Endow.*, 11(12):1835–1848, aug 2018.

[37] José Santos, Chen Wang, Tim Wauters, and Filip De Turck. Diktyo: Network-aware scheduling in container-based clouds. *IEEE Transactions on Network and Service Management*, 20(4):4461–4477, 2023.

[38] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 498–514, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Harsha Sharma, Parth Thakkar, Sagar Bharadwaj, Ranjita Bhagwan, Venkata N. Padmanabhan, Yogesh Bansal, Vijay Kumar, and Kathleen Voelbel. Optimizing network provisioning through cooperation. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1179–1194, Renton, WA, April 2022. USENIX Association.

[40] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

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

[42] Márton Sipos, Josh Gahm, Narayan Venkat, and Dave Oran. Network-aware feasible repairs for erasure-coded storage. *IEEE/ACM Transactions on Networking*, 26(3):1404–1417, 2018.

[43] Moritz Steiner, Bob Gaglianello Gaglianello, Vijay Gurbani, Volker Hilt, W.D. Roome, Michael Scharf, and Thomas Voith. Network-aware service placement in a distributed cloud environment. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 73–74, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Tino Tereshko and Jordan Tigani. Bigquery under the hood: Google's serverless cloud data warehouse | google cloud blog, Jan 2016.

[45] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 30:1–30:14, 2020.

[46] TPC. Transaction Processing Performance Council (TPC). https://www.tpc.org, 2024.

[47] Yiting Xia, Ying Zhang, Zhizhen Zhong, Guanqing Yan, Chiun Lin Lim, Satyajeet Singh Ahuja, Soshant Bali, Alexander Nikolaidis, Kimia Ghobadi, and Manya Ghobadi. A social network under social distancing: Risk-Driven backbone management during COVID-19 and beyond. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 217–231. USENIX Association, April 2021.

[48] Rohit Zambre, Megan Grodowitz, Aparna Chandramowlishwaran, and Pavel Shamis. Breaking band: A breakdown of high-performance communication. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[49] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: Admission control for performance-critical RPCs in datacenters. In *Proceedings of the*

ACM SIGCOMM 2022 Conference, SIGCOMM '22, page 1–18, New York, NY, USA, 2022. Association for Computing Machinery.

[50] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 221–234, Boston, MA, February 2019. USENIX Association.

[51] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

Appendix

A *QuerySys* Dataflow

Aggregation queries involve two stages (Figure 10), communicating via a shuffle primitive provided by *RamStore*. In the first stage, workers read data from *D* servers using the Colossus read API, perform parallel computations, and write intermediate results to *RamStore* memory hosts. Second, workers read these results, perform further computations, and write final results to memory hosts.

This process involves many-to-many communication between workers, *D* servers, and memory hosts, making it difficult to correlate ToR utilization with operation latency (§4.2). To simplify this analysis, we focus on the maximum ToR utilization observed across all ToRs[3] involved in each worker's operation (Table 4). This approach allows us to capture the impact of the worst hotspot on the operation's performance.

B Impact of Chunk Sizes

Figure 21 shows how chunk sizes impact the *Load-tolerance*. We look at 3 different operation size buckets: *10KB – 100KB*, *100KB – 1MB*, and *1MB–10MB*, and measure the 2× *Load-tolerance* for HDD reads and writes. A larger *Op size* decreases *Load-tolerance*, *i.e.,* making it more sensitive to the utilization. Thus, for example, reads of small chunks of 10-100 KB can tolerate fully loaded ToRs with a latency inflation of less than 2. For these small chunks, read access time is dominated by disk read latency (disk seeking), and so should, in general, offer worse performance than larger reads. In contrast, large reads of a MB or more incur a latency inflation of 2 even with a 50% ToR utilization, because they are bottlenecked by the network. HDD write operations have a low *Load-tolerance* even for small chunks, since writes do not wait for disk accesses, and hence are dominated by network latency at any size.

[3] Except for Colossus read operations we only consider the ToR utilization of workers but not *D* servers due to telemetry limitations.

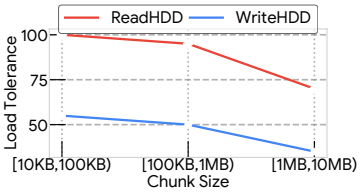


Figure 21: The impact of chunk sizes on the *Load-tolerance*.

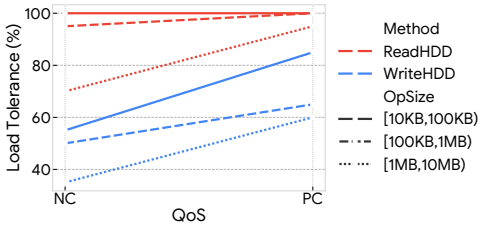
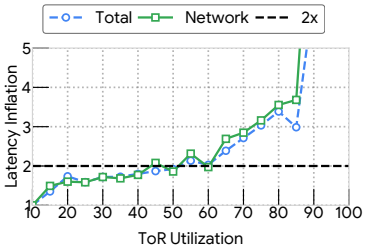
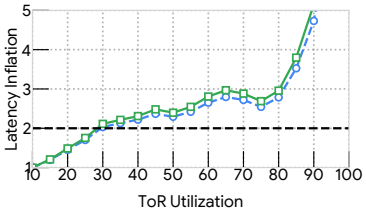


Figure 22: In all Op sizes, moving from NC QoS to PC QoS increases the *Load-tolerance*.



(a) SSD read.



(b) *RamStore* read.

Figure 23: The latency inflation of SSD and *RamStore* reads as a function of ToR utilization.

C Impact of QoS

Typical cluster applications in public clouds, including storage, classify their traffic into three priority classes: performance-critical (*PC*) traffic is associated with user-facing applications, non-critical (*NC*) traffic such as bulk storage operations generally cares about sustained rate, and best-effort (*BE*) traffic such as background analytics and backup traffic has the lowest priority. *PC*, *NC*, and *BE* traffic are mapped to high, medium, and low QoS classes respectively [49].

We measure network utilization every 30 seconds at each ToR for each quality of service (QoS) level. In our results reported below, the utilization always refers to the weighted-effective utilization of the given QoS class. The weighted-effective utilization is defined by the utilization of each

Operation	Description	Associated Network Utilization
Colossus Read	Read data from Colossus to worker	worker downlink
Shuffle Write	Write output from a worker to *RamStore*	**MAX**(worker uplink, *RamStore* memhost downlink)
Shuffle Read	Read the result generated by the previous stage from *RamStore* to worker	**MAX**(worker downlink, *RamStore* memhost uplink)

Table 4: The time of individual *QuerySys* worker's operation and the associated network utilization.

QoS and the weight ratio between QoSes. In practice, a flow of higher QoS can still be impacted by traffics of a lower QoS due to the weighted queues at the network devices. The weighted-effective utilization is capturing this subtlety with a weighted sum of all QoSes traffic occurs in the given ToR for each interval. Lower QoS contributions have smaller weight (< 1) and higher QoS traffic will have a larger weight (> 1).

Figure 22 shows that both HDD read and write has higher *Load-tolerance* when move from *Non-critical* (NC) to *Performance-critical* (PC) QoS. This is because traffic of the higher QoS observe less queuing time than that of the lower QoS. This less queuing delay will behave as a higher *Load-tolerance*.

D SSD and *RamStore*

SSD reads are much faster than HDD reads, and thus are more sensitive to network utilization. As shown in Figure 23a, SSD reads are more sensitive to high ToR utilization than HDDs. The *Hotspot-inflation* for SSD read is about $2.5\times$, and the $2\times$ *Load-tolerance* is about 55%. Thus, even when ToR utilization is not high enough to be classified as a hotspot, SSD read latency can be more than double the latency on an unloaded network.

RamStore reads have lower latency than Colossus since it stores the data in RAM. For this reason, *RamStore* is more sensitive to higher utilization and hotspots than any of the storage subsystems discussed so far. As shown in Figure 23b, *RamStore*'s $2\times$ *Load-tolerance* is 30%, and its *Hotspot-inflation* is 3.

Enhancing Network Failure Mitigation with Performance-Aware Ranking

Pooria Namyar[‡], Arvin Ghavidel[‡], Daniel Crankshaw[†*], Daniel S. Berger[†],
Kevin Hsieh[†], Srikanth Kandula[†], Ramesh Govindan[‡], Behnaz Arzani[†]
[‡]*University of Southern California,* [†]*Microsoft*

Abstract– Cloud providers install mitigations to reduce the impact of network failures within their datacenters. Existing network mitigation systems rely on simple local criteria or global proxy metrics to determine the best action. In this paper, we show that we can support a broader range of actions and select more effective mitigations by directly optimizing end-to-end flow-level metrics and analyzing actions holistically. To achieve this, we develop novel techniques to quickly estimate the impact of different mitigations and rank them with high fidelity. Our results on incidents from a large cloud provider show orders of magnitude improvements in flow completion time and throughput. We also show our approach scales to large datacenters.

1 Introduction

Datacenter networks often incur a variety of (concurrent) failures ranging from failed or lossy links to localized, persistent congestion. Repairing these failures takes time [1]. For example, operators require days to replace optical links and hours to fix hardware-induced packet corruptions [11, 21, 63, 71]. As a result, cloud providers install mitigations to reduce the impact of failures while they work on repairs. In this paper, we focus on *network-level* mitigations[1] such as disabling links or switches and re-routing traffic. These are effective because of the path and resource diversity in datacenter networks.

Given the economic importance of cloud services and the increasing likelihood of failures at scale, it is essential to find and implement effective mitigations quickly. Today, cloud providers are increasingly using automation to mitigate each incident. For example, Azure uses automation for nearly 80% of its incidents. An auto-mitigation system allows an operator to pre-define a limited set of potential mitigations for each type of failure. The system then chooses the "best" mitigation for each individual incident. At its core, an auto-mitigation system ***ranks mitigations*** based on one or more criteria, and it must do so ***quickly*** to be effective. At Azure, mitigations must be in place within 5 minutes [21] of failure localization.

Azure uses *local* criteria to assess mitigations.[2] For example, disabling a link is acceptable if it leaves sufficient functional uplinks at the corresponding switch. The state of the art either uses *global proxy* metrics, such as the residual path diversity from the top-of-rack (ToR) switches to the spine of the datacenter [71], or *global non-end-to-end* measures like packet loss and network utilization [63]. However, these methods can negatively impact customers by suggesting inadequate mitigations (§2).

Approach	Metric	E	G	U	B	S	P
NetPilot	Util/Drop	×	✓	×	✓	✓	×
CorrOpt	#Paths	✓	✓	×	×	✓	×
Operator	#Uplinks	×	×	×	✓	✓	×
SWARM	FCT/Tput	✓	✓	✓	✓	✓	✓

TABLE 1: SWARM is the only method that mitigates failures based on **E**nd-to-end **G**lobal **P**erformance metrics, considers **U**ncertainty in future networking behaviors, supports a **B**road range of actions and failure, and **S**cales to large datacenters. (**E**: End-to-End, **G**: Global, **U**: Uncertainty, **B**: Broadly applicable, **S**: Scalable, **P**: based on Performance)

In this paper, we explore a new mitigation ranking criterion (Table 1): the impact on the *end-to-end* connection-level performance (CLP) metrics, throughput and flow completion time. We can quantify the *global* impact using distributional measures of these quantities (averages and percentiles) across all connections in the datacenter. Failures adversely impact these global end-to-end metrics, and an ideal mitigation is the one that minimizes the impact. For instance, if our goal is to optimize 1st percentile (1p) throughput, the best mitigation is the one with the least impact on 1p throughput (§3).

From a cloud operator's perspective, this criterion reflects the network performance that customers experience and is preferable to local or non-end-to-end metrics that may not correlate with customer-visible network performance. However, at the scale of modern datacenters, the feasibility of quickly ranking mitigations based on global end-to-end CLP measures is unclear.

We introduce SWARM, a service for operators and auto-mitigation systems that quickly ranks mitigations while scaling to large clusters. SWARM leverages the insight that ranking mitigations only require an estimate of CLP distributions to produce an effective *ordering*. Its *CLP estimator* models traffic, routing, and transport behavior in sufficient detail to ensure ranking fidelity and, at the same time, produces results in just a matter of minutes.

To approximate CLP, SWARM must estimate *per-flow* performance not just for the current network state (topology and

*The author contributed to this work while at Microsoft.

[1] We use network-level mitigations and mitigations interchangeably. We briefly discuss application-level mitigations in §3.4.

[2] Its auto-mitigation system uses local criteria to determine whether taking a fixed action is better than taking no action.

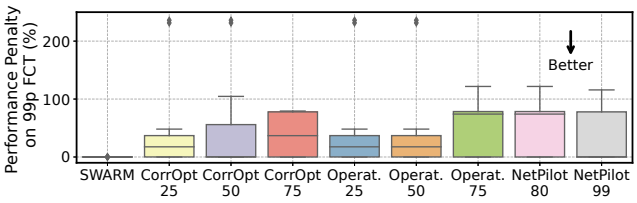


FIGURE 1: Our solution (SWARM) takes orders of magnitude better decisions by ranking mitigations using approximates of flow-level metrics and by accounting for uncertainties. The results are from scenario 1 in Mininet, see §4.2 (Operat. = Operator).

routing) but for potential (unknown) future network states that may manifest while mitigation is in place. Flow performance can depend on other concurrent flows as well as flow arrivals and departures. The central challenge in SWARM is producing an estimate quickly while accounting for all these factors.

SWARM overcomes these challenges as follows (§3):

- It takes flow arrivals, sizes, and communication probabilities as input distributions. It then samples a set of flow-level traffic traces to ensure statistical significance. For each demand, it generates routing samples to capture uncertainty in flows' paths. SWARM estimates CLP for each sample and then combines these estimates to create a composite distribution that succinctly captures traffic and routing variability. It uses this distribution to rank mitigations.
- SWARM estimates CLP separately for long and short flows. Estimating CLP for short flows is easier since they experience less time-varying network behavior. SWARM takes care in modeling long flows along two dimensions: (1) whether their throughput is limited by loss or contention, and (2) how this limitation changes over time as flows arrive/depart and network bottlenecks shift.
- It uses a suite of aggressive scaling methods, which include pipelining, parallelism, topology downscaling, and careful data structure design to estimate CLP quickly.

Using CLP estimates allows SWARM to explicitly account for failure characteristics (*e.g.,* packet drop rate), reason about a broader range of mitigations (taking no action or bringing back a previously disabled link), and model failures (*e.g.,* packet drops below the ToR) that previous methods [63, 71] cannot (see Table 1).

In summary, we make the following contributions:

- We propose CLP-aware failure mitigation, which finds the mitigation with the least impact on network performance. This is a significant departure from state-of-the-art.
- We identify sufficient approximations that allow us to build a robust and scalable CLP estimator that helps rank mitigations effectively (Fig. 1) and support a broader range of failures and mitigations compared to prior work (Table 1).
- We show SWARM is fast at scale and useful. For common failure scenarios (Scenarios 1 and 2 in §4), it picks either the best mitigation or one that is at most 9% worse than the best mitigation. It also outperforms the state-of-the-art by orders of magnitude. In more complicated failure cases which many prior work [63, 71] do not support, SWARM picks an action that is only 29% worse than the best mitigation.

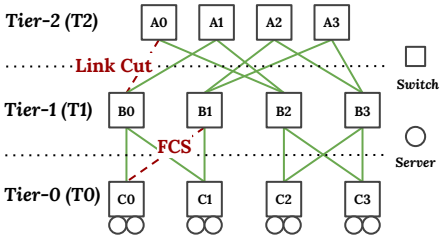


FIGURE 2: An example of two consecutive failures. First, the link between C0 and B1 experiences frame check sequence (FCS) errors. After mitigating, but before fixing this failure, a fiber cut between A0 and B0 causes congestion-induced packet drops.

2 CLP-Aware Mitigation

We use simplified versions of real-world incidents at Azure (Fig. 2) to explain the limitations of state-of-art auto-mitigation techniques and to illustrate the benefit of CLP-based impact assessments.

Failure scenario. Multiple link failures are common in cloud providers [7, 31, 55]. We emulate this in the Clos topology in Fig. 2. First, frame check sequence (FCS) errors [71] appear on a link. Operators mitigate this failure, but before they can physically replace the link, a fiber cut on another link (LINK CUT) causes congestion and packet loss.

We emulate (in Mininet [37], details in §4) a sequence of flow arrivals and successively apply each mitigation (or a combination of them) for FCS and LINK CUT. We use HIGH FCS and LOW FCS to denote the drop rate of $\sim 5\%$ and 0.005% respectively. For this example, our goal is to maximize the 1st percentile (1p) throughput [3]. We show how using Azure's troubleshooting guide, CorrOpt [71], and NetPilot [63] lead to substantial and unnecessary performance degradation.

Troubleshooting guides in Azure disable any failed link (with drop rate $\geq 10^{-6}$) if at least half of the switch uplinks are healthy. This mitigation for FCS achieves a 1p throughput of 3.6 Mbps and is optimal when the drop rates are high (HIGH FCS). However, it is conservative and static, and ignores the failure pattern, link location, and traffic demand. For example, leaving the lossy link in place in LOW FCS and taking no action has a higher 1p throughput (14.2 Mbps), while disabling the link causes congestion and impacts tail performance. For LINK CUT, Azure's guidelines do not do anything in the face of congestion, which results in a 1p throughput of 2.7 Mbps. In this case, the 1p throughput is higher if we adjust WCMP [70] weights to reduce traffic on congested links (3.2 Mbps).

CorrOpt [71] disables the link in FCS if there is sufficient path diversity. This is sub-optimal for the same reason dis-

[3] SWARM can optimize quantiles of both throughput and FCT.

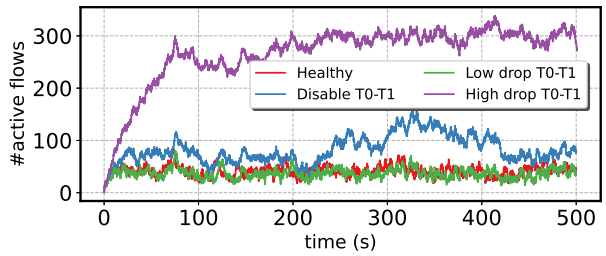


FIGURE 3: Failures and mitigations can increase flow durations, resulting in more active flows. (Fig. 2 topology in in Mininet)

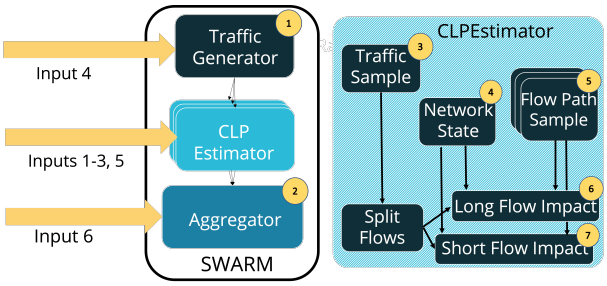


FIGURE 4: SWARM Design.

cussed above: depending on the failure properties (*e.g.,* drop rate and location), taking no action may lead to a higher 1p throughput. CorrOpt focuses on FCS errors and does not consider congestion induced by capacity drops, like LINK CUT.

NetPilot [63] always disables the faulty link to optimize one of its health metrics (loss rate). This may not be the best option. In a LOW FCS scenario, disabling the link drops the 1p throughput from 15 Mbps to 3.6 Mbps. After the LINK CUT, NetPilot disables the congested link or switch to avoid additional drops. This exacerbates the problem and reduces the 1p throughput to 3.17 Mbps. A better strategy is to undo the previous mitigation and re-enable the LOW FCS link when LINK CUT occurs, which results in 1p throughput of 14.2 Mbps.

Takeaways. While this is a simplified example, operators can negatively impact customers in practice and cause extended outages if they fail to find an effective mitigation [2]. Rules with static thresholds in troubleshooting guides cannot capture correct mitigations because CLP impact depends on traffic demands. Path diversity measures (as in CorrOpt) cannot capture customer impact since they do not account for the failure characteristics. Non-end-to-end metrics like packet loss or utilization (as in NetPilot) often suggest disabling links, which discounts better mitigations. In contrast, SWARM ranks mitigations based on impact on end-to-end global measures.

3 SWARM Design

Network operators seek to improve flow completion time (FCT) and throughput in their datacenters [5, 6, 41, 44, 61]. These CLP metrics evaluate the performance properties of importance for short and long flows. Moreover, operators express their objectives as distributional statistics over the entire datacenter (*e.g.,* tail or average performance). In this section, we describe how SWARM ranks mitigations in terms of their impact on CLP metrics, throughput and FCT.

3.1 Challenges and Insights

Challenges. To optimize CLP objectives, SWARM must quickly estimate the distributions with sufficient accuracy to ensure effective mitigation ranking. This is hard:

Traffic characterization. SWARM requires information about traffic demands to estimate CLP distributions for a given failure and mitigation. Instantaneous flow or ToR-level traffic matrices (TMs) can provide this information. However, fine-grained flow-level TMs are impractical to capture at datacenter scales and are sensitive to failures and mitigations. For example, packet drops often extend the flow durations, resulting in 3 - 4× more concurrently active flows (Fig. 3). ToR-to-ToR TMs, as in NetPilot [63], are too ambiguous since they aggregate flows with different characteristics (*e.g.,* capacity drops impact long flows more than short flows).

Routing determines the contention level at each link, which impacts throughput and FCTs. It depends on ECMP hash functions, as well as any existing failures in the network. The ECMP hash functions can change when links fail or switches reboot [56]. SWARM must consider these factors.

Transport behavior. SWARM needs to model transport behavior since CLP measures depend on congestion control algorithms (*e.g.,* BBR vs. Cubic), their parameters (*e.g.,* initial window size), and their reaction to failures (*e.g.,* packet drops). However, it is hard to model transport behavior while maintaining scalability. Accurate simulations over-index on specific protocols and are slow [29]. Faster approximate simulators do not account for lossy links [67, 68] or require a prohibitive amount of compute [67]. Existing formal models have limitations (see §6). For instance, fluid models [36] capture steady-state for long flows, but datacenter flows are often short and do not reach a steady state [44].

Temporal and spatial dependencies. CLP measures depend on the time-varying number of flows competing for bandwidth at a link. They also depend on where and for how long these flows experience congestion. For example, a flow bottlenecked at a link does not need its full fair share at other links. These bottlenecks may shift frequently due to flow arrivals and departures, and SWARM must capture them.

Approach. At the core of SWARM (Fig. 4) is a *CLPEstimator* that estimates the *distribution* of throughput and flow completion time (FCT) for a given network, failure pattern, and mitigation set.

SWARM avoids the limitations of fine-grained flow-level TMs and uses an *approximate* TM distribution (§3.2). It generates multiple (①) TM samples (③) from three inputs that cloud providers like Azure already collect: the flow arrival rates, the flow size distributions [44], and the probability of server-to-server communication [9].

Failure	Mitigation	Works that consider these failures/actions
Packet drop above the ToR	Take down the switch or link	NetPilot, CorrOpt, Operators
	Bringing back less faulty links to add capacity	×
	Changing WCMP weights	×
	Do not apply any mitigation	×
Packet drop at ToR	Disable the ToR	Operators
	Move traffic *e.g.,* by changing VM placement	×
	Do not apply any mitigation	×
Congestion above the ToR	Disable the link.	NetPilot, Operators
	Disable the device	NetPilot, Operators
	Bring back less faulty links to add capacity	×
	Change WCMP weights	×
	Do not apply any mitigation	×

TABLE 2: List of failures and mitigations in SWARM. Disabling a device is a common mitigation for congestion [63], see §E.

SWARM addresses routing uncertainty by evaluating CLPs on enough flow path samples (5) to reach a target statistical confidence (§3.3). It generates these samples based on the network state (4).

The trickiest challenge is to model the impact of losses and dependencies between concurrent flows on throughput and FCT. To this end, SWARM uses three techniques.

Epoch-based flow rate estimator. SWARM uses a fast, scalable epoch-based flow rate estimator to handle temporal bandwidth changes and flow dependencies. It divides time into multiple epochs, recomputes CLPs in each epoch, and *combines* the results to find an overall estimate.

Traffic Classification. SWARM needs to quickly find effective mitigations at scale rather than accurately estimating the flow performance, which can be slow [29]. Therefore, SWARM divides traffic into long (6) and short flows (7) based on their sizes [5, 44]. In each epoch, it estimates throughput and flow completion time separately. This approach results in significantly better mitigations (§4) for three reasons:

First, failures affect short and long flow differently. Longer flows are exposed to network variations, while shorter flows only experience a snapshot of the network and have more predictable FCTs.

Second, congestion control algorithms [12, 28] typically have a start-up phase to find available bandwidth. Short flows with few packets may finish during this phase. Therefore, they experience queueing delays caused by switch buffer occupancies rather than bandwidth limits [5, 44]. Accounting for these factors improves the effectiveness of our approach.

Third, previous studies [5, 6, 44] have shown that datacenter traffic is a mixture of short and long flows. Short flows are delay-sensitive, while long flows are throughput-sensitive. SWARM reports these separately and allows operators to adjust their mitigations based on their requirements. For example, they can prioritize short flows if their workload mainly consists of latency-sensitive short flows. Modeling these two classes separately also helps with scalability. Future work can explore modeling more than two classes of flows, such as those that are not short enough to ignore startup behavior.

Transport protocol abstraction. SWARM uses an approximate model of transport protocols that is effective and scalable: (1) it assumes long flows are TCP-friendly [4, 20], which means each long flow grabs a fair share of the bottleneck bandwidth in the absence of failures. (2) under failures and packet drops, SWARM determines if long flows are *capacity-* or *loss-limited*. For capacity-limited flows, it computes their fair share of bandwidth. For loss-limited flows, it estimates the bandwidth at which the control loop converges under loss. SWARM extends existing max-min fair algorithms [34, 45] to detect and estimate both simultaneously. (3) SWARM assumes short flows do not reach steady-state and are impacted by packet drops or queueing delays rather than the bandwidth limits.

SWARM aggregates distributions (2) from each traffic and routing sample. Operators use these to rank mitigations and set priorities based on one or more distributional metrics (*e.g.,* prioritize average throughput over FCT).

3.2 SWARM: Inputs and Outputs

Inputs. Operators or auto-mitigation tools can invoke SWARM with the following inputs:

1. Datacenter topology.
2. List of ongoing mitigations (if any).
3. Failure pattern (*e.g., estimated* loss rate) and location.
4. Data center traffic details (*e.g.,* TMs distributions).
5. Candidate mitigations to evaluate.
6. A *comparator* that ranks mitigations by CLP estimates.

Inputs 1-2-3. Cloud providers use monitoring systems [13, 40] and automated watchdogs [21] to detect incidents and use different techniques [8, 63, 66] to localize the failure. They then create incident reports [1, 2, 21] that contain details of the incident. On-call engineers or automation systems use these reports to install mitigations that are active until the operator finds the root cause and repairs the failure. These reports contain the information SWARM needs, such as the failure

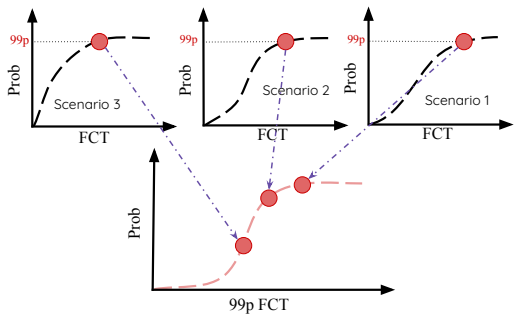


FIGURE 5: The composite distribution of 99p FCT obtained from the FCT distributions of different traffic and routing samples.

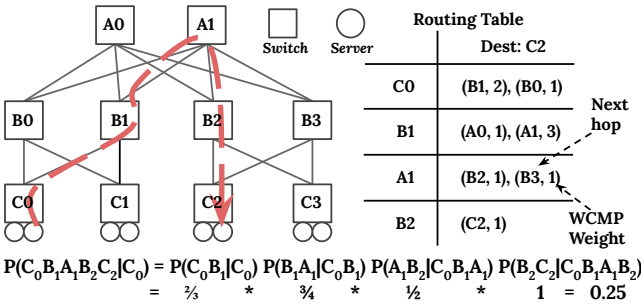


FIGURE 6: The probability of a flow taking a particular path.

location or pattern. The failure characterization (*e.g.,* packet drop rate) is sometimes imperfect, and operators may not be able to accurately localize the failure. SWARM can tolerate errors in packet drop rate (§4), and operators can incorporate the probability of different locations (§5) and iteratively refine mitigations to deal with imperfect localization.

Input 4. SWARM requires simple characterizations of datacenter traffic: the flow arrival distribution, the server-to-server communication probability, and the flow size distributions. From these, it extracts a set of flow-level demand matrices (§3.3). These probabilistic characterizations of the inputs allows SWARM to be robust to traffic variability and ensure a desired level of statistical confidence in its estimation.

Input 5. SWARM requires a mapping from the failure types to a list of mitigations or a combination of mitigations. Cloud providers [33, 57] already document possible mitigations for each type of incident in their troubleshooting guidelines and can use these documents to create the mapping. Certain actions may require additional information, such as VM placements or WCMP weights. We assume operators use existing techniques [26, 60] to find values for these inputs. Table 2 shows a sample of failures and associated mitigations.

Input 6. Operators can customize the *comparator*. We currently support two types of comparators, and we can easily extend to others (see §4). The *priority* comparator considers throughput-based and FCT-based metrics in a pre-specified priority order. The *linear* comparator is a linear combination of two or more of these metrics, where the operator specifies the weights. This flexibility enables operators to adjust their mitigation strategy based on their workloads. For example, if delay-sensitive short flows [5, 44] are the dominant workload, they can choose to prioritize the impact on short flows.

Outputs. SWARM outputs the mitigation (or mitigation combination) with minimal impact as ranked by the comparator.

3.3 SWARM: Internals

The CLPEstimator (Fig. 4) takes a demand matrix $\mathcal{T}$ and a mitigation $\mathcal{M}$ as inputs and estimates: (a) the distribution of average throughput across all the long flows in $\mathcal{T}$ and (b) the distribution of FCT across all the short flows in $\mathcal{T}$. SWARM can compute average throughput from FCT and vice versa using FCT $= \frac{\text{flow size}}{\text{throughput}}$ if necessary. SWARM calls the CLPEstimator for each candidate mitigation and ranks these mitigations based on the estimates (see Alg. A.1).

How SWARM uses CLPEstimator. SWARM samples K different demand matrices (1 in Fig. 4) and invokes the CLPEstimator to evaluate a given mitigation on each of them. The CLPEstimator then internally generates N different *routing samples* (5), where each sample describes the path for every flow in the demand matrix. We choose K and N to ensure a desired confidence level (see below).

Depending on the comparator, SWARM estimates the *distribution* of the *percentiles* of throughput and FCT across the traffic and routing samples (Fig. 5). For instance, if the comparator uses 99p FCT, SWARM would extract the 99p from FCT distributions of the $N \times K$ samples and form a *composite distribution* of the 99p FCTs. The variance of this distribution captures the uncertainty in our estimates, which we can reduce by increasing the number of samples [14] (Fig. A.4).

SWARM uses the composite distribution (2) to compare and rank mitigations. This approach allows it to handle uncertainty explicitly and provide robust mitigation rankings.

Network state representation. SWARM models the network state (4) using a graph $G = (\mathcal{V}, \mathcal{E})$, where each edge e has a capacity and a drop rate (0% = healthy and 100% = down), each node v has a drop rate and a routing table, and each server s maps to a switch. Before each invocation, SWARM *updates* this state to reflect the mitigation (line 2 in Alg. A.1). It uses data structures to ensure this step is efficient (§3.4).

Modeling traffic variability. The demand matrix $\mathcal{T}$ includes the arrival times, flow sizes, and their corresponding source and destinations. To create $\mathcal{T}$, SWARM uses input 4 (1): for each flow, it randomly samples the arrival time from the flow arrival distribution, the source-destination pair from the server-to-server communication probabilities, and the flow size from the size distribution. SWARM then invokes CLPEstimator with K demand matrix samples. It uses the Dvoretzky–Kiefer–Wolfowitz (DKW) inequality [18] to determine K based on a given confidence level α. DKW provides confidence for the difference between an empirically sampled distribution and the ground truth distribution.

Modeling routing uncertainty. The CLPEstimator handles

routing uncertainty by generating N different flow path samples (5), each representing a different routing of the flows (§3.1). First, it uses the DKW inequality [18] to determine the N to achieve a confidence level of α. Then, it samples from the distribution of possible paths between source-destination pairs. It computes the probability of a specific path for a given source-destination pair based on routing tables and associated WCMP weights at each node (Fig. 6).

Modeling the throughput of long flows. SWARM models long and short flows separately (Line 6 and Line 7 in Alg. A.1). Long flows typically reach their steady state, and their throughput depends on network variations and packet drops.

Varying network conditions. SWARM approximates network variations and the arrival and departure of flows that compete on a link by dividing time into discrete epochs (Alg. 1). It assumes stable conditions within each epoch (no flow arrival/departure) but allows for variations across them. At the beginning of each epoch, SWARM adds the newly arrived flows to the set of active flows (Line 6). Then, it computes their bandwidth share (Line 7). At the end of each epoch, SWARM updates the number of transmitted bytes for each flow, removes the completed flows, and records their overall throughput estimates (Line 8-Line 16).

Computing bandwidth share. SWARM finds the rate of each flow within an epoch in two steps (Line 7 in Alg. 1):

1. It computes the loss-limited throughput of each flow, as described below.
2. It executes a demand-aware extension of the water-filling approach [34], which uses the loss-limited throughput estimates as an upper bound on the flow throughput and provably converges in O(number of edges). We have developed this extension (see §A.2).

Modeling loss-limited throughputs. It is possible to compute the throughput under packet drop analytically, but these models are specific to certain congestion control protocols and cannot easily extend to other variants (*e.g.,* [5]). SWARM overcomes this limitation by using an empirically-driven distribution of the loss-limited throughputs. To find this distribution, SWARM measures the average throughput of a long flow under different network conditions (*e.g.,* drop rate, latency) through experiments in a small testbed. In each experiment, it ensures that link capacities are high enough so that they never become bottlenecks, and the drop rate is the only limiting factor. SWARM repeats the experiment multiple times for each network condition to create a robust distribution [18]. Depending on the uncertainty in transport protocols running in the data center, we can also change the mix of transport protocols we use. SWARM then uses this distribution to sample the drop-limited throughputs. See §B for details.

Modeling the FCT of short flows. Prior work [42] develops an analytical model for the average FCT across flows. We are unaware of any models that can estimate the distribution. In developing SWARM, we observe that short flow are more predictable since they do not stay in the network long enough to be affected by network variations. Thus, a simple empirical model is sufficient to estimate their FCT distribution.

Algorithm 1: Impact on Long Flows.

Input: G. current network state.
Input: $\mathcal{T}$ = {<source, destination, size, start time>}.
Input: $\mathcal{R}$. the sampled routing of each flow.
Input: $\mathcal{I}$. measurement interval.
Input: ζ. epoch size.
Output: β_l. distribution of throughput of long flows

```
1  K ← {}
2  time ← 0
3  β_l ← []
4  while ∃ f ∈ T : f.start ≥ time or K ≠ ∅ do
5      time ← time + ζ
6      K.add({f ∈ T : time − ζ ≤ f.start < time})
7      θ_f ← compute_throughput(G, K, R)
8      for flow f in K do
9          f.sent ← min(f.sent + ζθ_f, f.size)
10         if f.sent = f.size then
11             K.remove(f)
12             if f ∈ I then
13                 β_l.append(f.size / f.dur)
14             end
15         end
16     end
17 end
18 return β_l
```

SWARM finds a short flow's FCT distribution (6) by estimating (a) its RTT and (b) the number of RTTs to deliver its demand. A short flow's FCT is equal to the average duration of RTTs multiplied by the required number of RTTs. We derive the distribution of each metric separately and then combine them to compute the FCT distribution.

We compute the distribution of the number of RTTs to deliver a flow's demand by conducting offline experiments in a small testbed. We repeat the experiments for different configurations (flow size, slow start threshold, initial congestion window) and network settings (drop rates, RTT) and store the results in a table that maps the setting to the measured distribution.

Next, we estimate the RTT, which is equal to the sum of the propagation delay (a constant determined by the flow's path) and the queueing delay along the flow's path. To estimate queueing delay, we collect data from sending small flows on links with different utilization and active flow counts (§B).

SWARM combines these distributions to derive the FCT distribution for short flows. It multiples the number of RTTs by the sum of the propagation delay and queueing delay.

3.4 Expressivity, Scaling, and Robustness

Expressivity. SWARM supports any failure or mitigation as long as we can model it as changes to the network state or the traffic. SWARM does not need the root cause of a failure or details of a mitigation but only needs to understand their observable impact (*e.g.,* packet drop, port down). This flexibility allows SWARM to support a wide range of failures and mitigations (Table 2). Existing production and state-of-the-art systems [63, 71] do not support many of these failures, while cloud providers [7, 24, 31, 55, 63, 71]) commonly observe them.

Robustness. SWARM's network-level mitigations should ideally mask network failures. However, this masking is not always perfect, and a cloud service might react to a mitigation (or the failure in general) by changing the traffic demand (*e.g.,* using retries). For this reason, SWARM does not model a fixed traffic demand. Instead, it draws multiple traffic samples from the historical distribution of flow arrivals, sizes, and communication probabilities to ensure statistical significance (§3.3). When operators do not have such statistics (*e.g.,* after a previously unseen failure or a data center expansion), SWARM uses a distribution that captures maximum uncertainty [51].

SWARM carefully models uncertainty, but its CLP estimates may not align with what operators observe after installing the mitigation. This can happen if an uncommon failure manifests itself that was not captured during the sampling process or if the traffic changes and the historical distribution is not representative anymore. In such instances, the auto-mitigation system that uses SWARM must update its inputs and invoke SWARM again to revise the mitigation. In other words, mitigation does not have to be a single-shot process and can be adjusted over time, especially since failure diagnosis might take hours to days [21, 63].

Scaling. SWARM must find a mitigation action quickly, even on large clusters. We scale it using the following techniques:

An ultra-fast max-min fair computation algorithm. We use an approximate computation of network-wide max-min fair share rates [45], which provides significant speedup over the state-of-art methods [34] without affecting quality.

Efficient network state and traffic update. The traffic trace is independent of the network state. This means SWARM can compute the traffic samples offline. However, when SWARM invokes CLPEstimator, it must update network state to reflect each mitigation and then re-compute routing samples. To scale, SWARM separates the topology representation from the traffic representation. It models the former as a graph (§3.3) and sorts the latter into a list of tuples (source and destination server, flow size, and flow start time). This design enables, for example, disabling a link by changing the drop rate in G to 100% or updating the traffic if a mitigation such as VM migration modifies a flow.

Parallelism and pipelining. SWARM (a) evaluates demand and routing samples in parallel, and (b) parallelizes and pipelines routing sample generation with epoch execution.

Reducing the number of epochs. After these steps, the bottleneck is the number of epochs in Alg. 1. We use two techniques. Firstly, we initialize on an already warmed-up network instead of starting from an empty network. This eliminates the need for a set of epochs at the beginning to mimic the cold-start effect. Secondly, the residual impact of flows that compete within an epoch diminishes over time. Therefore, epochs with large time differences present independent network snapshots. This observation allows us to compute their CLPs independently in parallel and combine the results to form an overall distribution.

Traffic downscaling. Following POP [47], SWARM downscales the demand matrix with minimal impact on throughput. It splits a network with link capacity c into k sub-networks with link capacity $\frac{c}{k}$ and divides traffic randomly across these sub-networks. POP [47] recommends choosing a value for k that is much smaller than the number of flows, which allows each partition to capture the network contention. This approach works with any flow arrival distribution. We use Poisson distributions [23, 35, 41], where assigning flows randomly is the same as downscaling the arrival rate based on the Poisson splitting property.

4 Evaluation

Our prototype of SWARM has 1500 lines of Python. We evaluate it on three categories of incidents and show it outperforms existing methods. We also show it scales and finds the best mitigation for a 16K-server topology within 5 minutes.

4.1 Methodology

Metric. We evaluate each approach by computing the *Performance Penalty (%)*, which is the relative difference between the CLP metrics that result from the best possible mitigation and the one each technique suggests. This metric captures the unnecessary performance degradation caused by a technique choosing sub-optimal mitigation. Often, the difference between the best mitigation and the "runner-up" is insignificant, resulting in a small penalty. In contrast, the difference between the best mitigation and the one the baselines choose can be as high as 200%.

Experimentation setup. We evaluate SWARM using Mininet [37], NS3 [29], and a physical testbed. Here is the summary of our setup (see §C for more details):

Traffic characterization. We use Azure production logs to derive the flow inter-arrival time, a commonly used distribution from DCTCP [5] for our flow size distribution, and [38] for server-to-server communication probability. We ensure that the trace duration is long enough to capture all flow sizes and that we do not capture an empty network's effect. Finally, we run on 30 different traces to ensure robustness.

Emulation Setup. We mainly use Mininet for our evaluation since it leverages a real TCP/IP stack from the Linux

kernel. We extend it to improve the fidelity of our results (monitoring systems, queueing disciplines). We report results on 57 scenarios (over 4000 hours of experiment) across 3 types of common incidents in cloud providers [7, 24, 55, 63, 71] (Table 2). We describe these scenarios in §4.2 and Table A.1. We use the Clos topology from Figure 2.

We seek to emulate 1500 flows arriving per second per server, which results in 12,000 flows every second. These are on links with 40 Gbps bandwidth and 50 μs propagation delay. However, running Mininet on a VM with 64 cores and 256 GB cannot emulate this demand. Instead, we use [48, 50] to downscale the traffic and link capacities by 120×(see §C.4). We run each emulation for 500 s and measure the performance for flows that start within [50, 150) s to avoid capturing effects from an empty network. We report results for both Cubic [28] and BBR [12] (and DCTCP [5] in our simulation).

Simulation Setup. We use a Clos topology with 128 servers, 32 ToRs, 32 T1s, and 16 T2s, all connected by 20 Gbps, 100 μs links. We use DCTCP [5] as the congestion control algorithm to show generality. Each of our traffic traces is 10 s long, and we measure the flows that start within [0.5, 1) s[4]. We use DCTCP [5] and FbHadoop [54] flow size distributions.

Testbed Setup. We use a different variant of Clos (see §C.3) with 32 servers, six TORs, four T1s, and two T2 switches, all connected by 10 Gbps 200 μs links. All switches are Arista 7050QX-32. We introduce random packet loss in the testbed using a user-defined access control list (ACL) to match bits on the IP ID field in packet headers and directly modify the Broadcom firmware to drop packets the ACL matches. Thus, packet loss rates in the testbed are powers of two based on the number of bits the ACL matches on. We evaluate the impact of the failure and each mitigating action on the testbed with a traffic load of 3000 flows per second. Each trace is 30 seconds long, and we measure across flows that start within [2, 5) s.

SWARM *Parameters.* We use 32 different random traffic traces and 1000 routing samples based on §3.3. For our baseline comparisons (with Mininet), each trace is 200 s long. We compute the performance over all the flows that start within [50, 150) s. Each epoch is 200 ms. We also consider any flow with a size ≤150 KB short.

Baselines. We compare SWARM to:

NetPilot [63] iterates through each possible mitigation, computes the maximum link utilization, and picks the action that minimizes utilization. NetPilot does not model link utilization on faulty links, so it always disables corrupted links. We report these results as NetPilot-Orig. We also extend NetPilot to mitigate only if the resulting maximum link utilization is below a threshold. We report these results as NetPilot-80 for an 80% and NetPilot-99 for a 99% utilization threshold.

[4]We adjust the duration based on the flow arrival rate and link bandwidths. When these values are smaller as in our Mininet experiments, it will take longer for a certain number of flows to arrive, and each flow remains in the network for a longer period. Thus, we need to cut a longer duration at the beginning to ensure we are not capturing the effect of the empty network.

CorrOpt [71] only considers link corruption failures. It disables a link if the number of remaining paths to the spine after taking the action is above a threshold. We consider three thresholds: CorrOpt-25, CorrOpt-50, and CorrOpt-75, which use a threshold of 25%, 50%, and 75%, respectively.

Operator playbooks. When an FCS error occurs above the ToR where there is path redundancy, the Azure playbook will disable the affected link if the number of remaining uplinks at the switch is above a certain threshold. We consider three thresholds: Operator-25 uses a 25% threshold, Operator-50 uses 50%, and Operator-75 uses 75%. When there is packet loss of more than 10^{-3} at or below the ToR, the playbook will drain the affected nodes, which is expensive and risks VM reboots or interrupts. Otherwise, it would take no action.

Some baselines cause a partitioned network under certain scenarios in part due to the smaller scale in our Mininet evaluations. Unless noted otherwise, we only report cases where all baselines keep the network connected for a fair comparison.

Comparators. Most experiments use two priority comparators (§3.2) (§D.4 shows SWARM achieves low penalty across two other comparators including a linear one as well):

PriorityFCT minimizes the 99p FCT. It uses two tiebreakers, 1p throughput followed by average throughput.

PriorityAvgT maximizes the average throughput first, using two tiebreakers, 99p FCT, followed by 1p throughput.

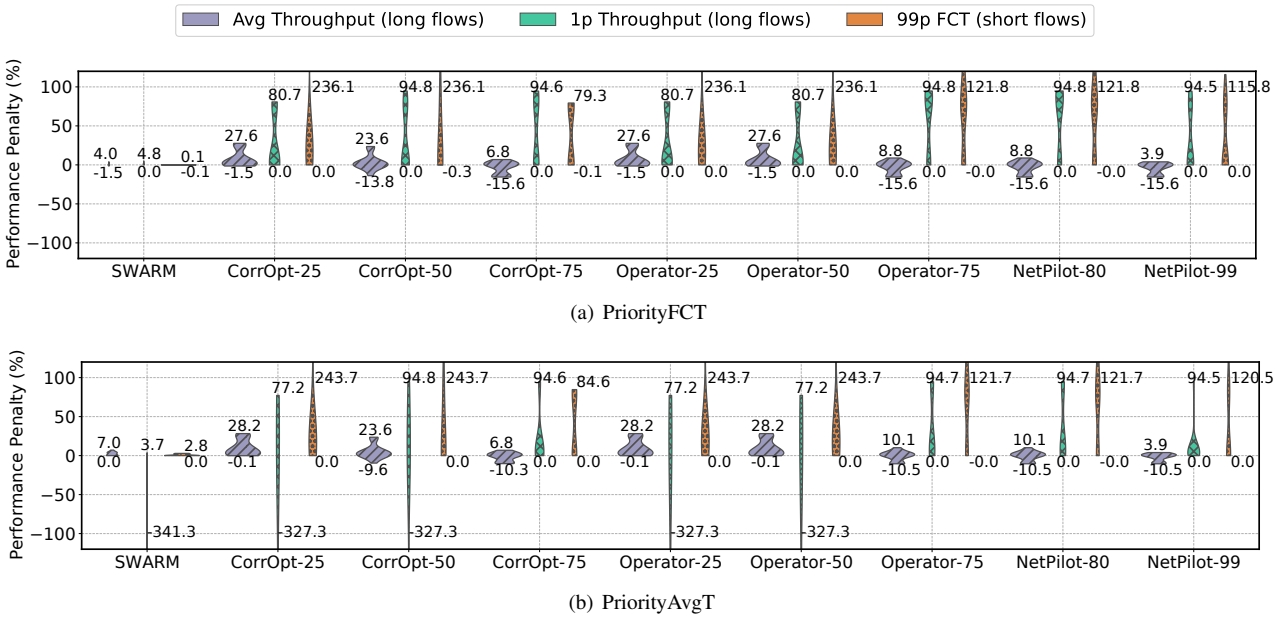
Two mitigations are tied on a particular metric if they are within 10% of each other on that metric.

4.2 Baseline Comparisons

We evaluate SWARM over three different failure scenarios, which are common in production incidents at Azure.

Scenario 1: Link-level packet corruption with network redundancy. In this scenario, we evaluate different combinations of two links consecutively experiencing FCS errors with a drop rate of $\sim 5\%$ (*high*) or $\sim 0.005\%$ (*low*). This is the most common failure pattern at Azure, and both drop rates are detected and reported as incidents. In this case, the viable mitigations are doing nothing, disabling the link, undoing past mitigations, changing WCMP weights, or any feasible combination of these. All baselines support this scenario because the link failure is above the ToR, and there is path redundancy.

In Fig. 7, we compare the performance penalty of SWARM to the baselines for both comparators. We do not compare to NetPilot-orig in this scenario since it partitions the network in 16 out of 32 failure pairs, and the results on the remaining failures are not statistically significant. We use violin plots to show the performance penalties' distribution across all incidents for each candidate approach and each metric. A tall violin plot indicates that performance penalties span a wide range, while a short and wide plot indicates that penalties are clustered within a small set of values. Even though the baselines do not explicitly use comparators, the best mitigation depends on the comparator, which causes the penalty to change for the baselines across different comparators.



(a) PriorityFCT

(b) PriorityAvgT

FIGURE 7: Comparison of SWARM and other baselines under Scenario 1 in Mininet. SWARM achieves 793× lower performance penalty on 99p FCT in the worst case compared to the next best baseline on PriorityFCT. SWARM is the only technique that achieves near-optimal performance across all three metrics and performs equally well across both comparators.

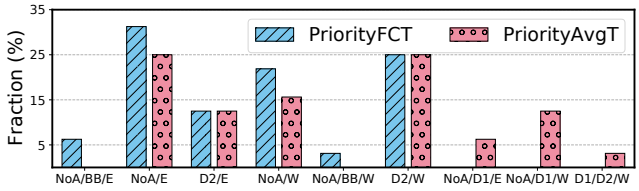


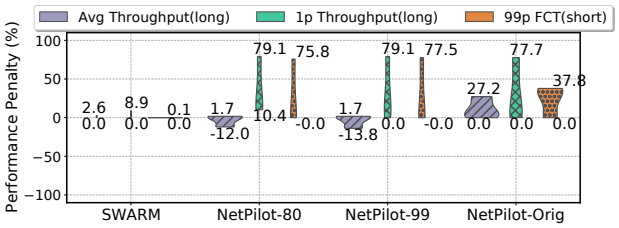
FIGURE 8: SWARM's actions in Scenario 1 (see §4). It chooses from nine action combinations and decides to take no action in more than 25% of the time. (NoA: No Action on link 2, D2: Disable link 2, BB: Bring Back link 1, D1: Disable link 1, W: WCMP Routing, E: ECMP Routing)

SWARM outperforms all baselines with a performance penalty consistently close to zero. When using the PriorityFCT comparator, SWARM has a maximum FCT performance penalty of 0.1%, compared to 79.3% penalty of the closest baseline, CorrOpt-75 (Fig. 7(a)). For the Priority-AvgT comparator, SWARM's maximum penalty on average throughput is similar to several of the baselines, such as NetPilot-99 and CorrOpt-75. However, SWARM reduces the performance penalty across *all three* CLP metrics while the baselines suffer from high performance penalties across at least one (Fig. 7(b)). SWARM's superior performance is due to its ability to analyze a broader set of mitigations and to choose one that reduces performance impact on all CLP metrics (see §F).

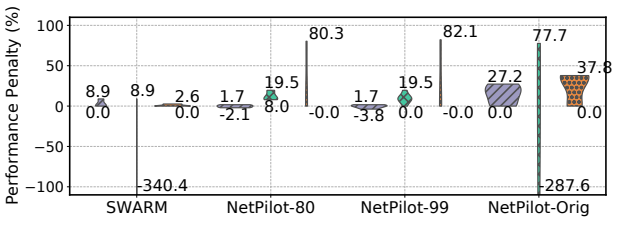
The negative penalty in some cases is because of the inherent trade-off between different metrics. For instance, the primary objective in Fig. 7(a) is to minimize short flows' 99p FCT. The optimal mitigation is the one that has the best FCT, even if it worsens other metrics. For example, CorrOpt-75 selects actions that increase the 99p short flow's FCT (highest priority) by up to 80% compared to the optimal action but achieve better average throughput (lower priority).

We show the diversity of SWARM's proposed mitigation under each comparator in Fig. 8. We focus on the mitigation for the second failure since the action space is larger and includes options such as bringing back a previously disabled link. We find SWARM chooses from nine different possible mitigations and decides to take no action in more than 25% of the cases. In two scenarios under PriorityFCT, it not only takes no action on the second failure but also *reinstates* the faulty link it previously disabled from the first failure (action NoA/BB/E). SWARM considers the failure locations, their intensity, and the traffic demands and decides to preserve those links rather than eliminate their capacity and cause congestion. Under PriorityAvgT, SWARM chooses combinations of mitigations such as disabling both links and adjusting WCMP weights.

Scenario 2: Congestion on a link. Operators shut down several faulty links (prior failures), causing the network to become over-subscribed. Meanwhile, an aggregation-core layer starts operating at half capacity (due to fiber cuts). CorrOpt and operator playbooks do not support this as they ignore traffic dynamics. NetPilot can reason about congestion but assumes the rest of the network is under-utilized. We evaluate SWARM and NetPilot under two failure patterns: (i) where the network is under-utilized and (ii) where a second link drops packets and reduces network capacity (Table A.1).

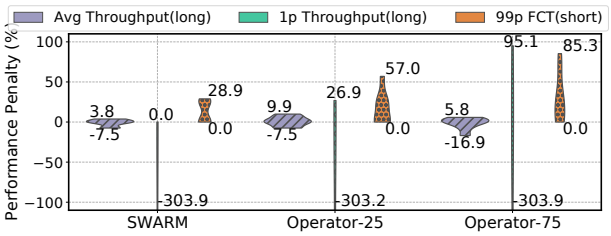


(a) PriorityFCT

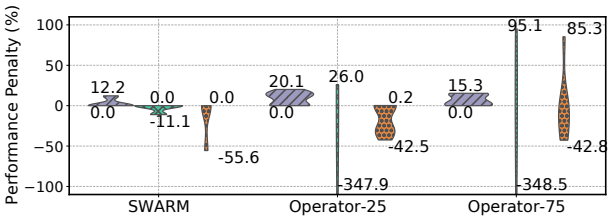


(b) PriorityAvgT

FIGURE 9: Comparison of SWARM and NetPilot variants under Scenario 2 in Mininet. Under the PriorityFCT comparator, SWARM always chooses a mitigation with near-optimal FCT performance ($\leq 0.1\%$), while the next best approach chooses mitigations up to 37.8% worse than optimal. SWARM is also the only approach with low performance penalty across all three metrics.



(a) Priority FCT



(b) Priority AvgT

FIGURE 10: Comparison of SWARM and operator playbooks on Scenario 3 in Mininet. Under PriorityFCT, SWARM has 2× lower 99p FCT penalty in the worst-case compared to operator playbooks. It also protects other non-priority metrics while optimizing for the priority metric (it outperforms baselines in both average throughput and 99p FCT under PriorityAvgT).

Fig. 9 compares the performance penalty of SWARM to the NetPilot variants for both comparators across both failure patterns. SWARM achieves consistently low penalty for its target CLP metric: 99p FCT in Fig. 9(a) and average throughput in Fig. 9(b). Since NetPilot assumes the rest of the network is under-utilized, it aggressively disables links and causes a large performance impact. Under PriorityFCT, SWARM chooses a mitigation with near-optimal performance on FCT, while the next best approach suffers an FCT penalty of 38%. Under the PriorityAvgT comparator, the NetPilot-80/99 variants result in lower impact on average throughput, but at the cost of increased performance penalty in at least one other metric (*e.g.*, NetPilot-80 achieves 7.2% less penalty in terms of average throughput at the cost of 80% penalty on 99p FCT) while SWARM is the only technique that performs well across both comparators and all three metrics.

Scenario 3: Packet corruption at ToR. In this scenario, we consider (Table A.1) failure patterns in which a ToR drops packets at either *high* 5% or *low* 0.005% rates. We also consider cases where both the ToR and a core link drop packets. CorrOpt and NetPilot do not support this failure as they can only account for scenarios where the network has redundant paths. The operator playbook makes a local decision on whether to mitigate the failure based on the severity of the packet loss, without considering the overall network condition. When the second failure occurs above the ToR, it reduces the capacity in the network core, causing the operator's approach to incur higher performance penalties.

Both playbook-based approaches suffer from performance penalties at least 2× higher than SWARM under PriorityFCT (Fig. 10). SWARM has a worst-case FCT penalty of 28.9% while the best operator approach has a worst-case FCT penalty of 57%. SWARM is again the only approach that achieves low penalty across all three metrics for both comparators.

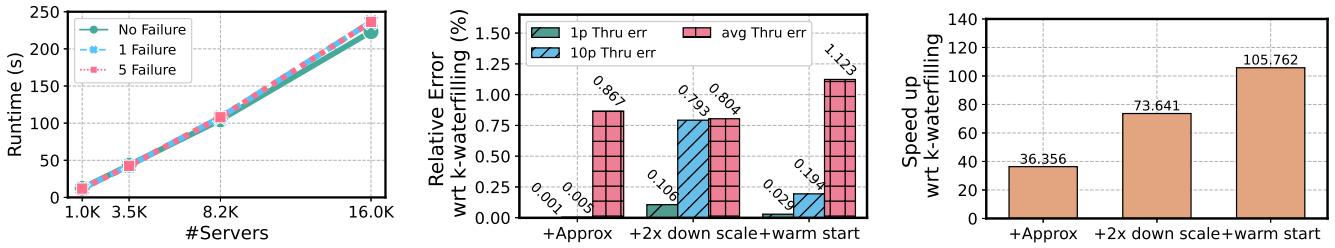
4.3 Other Results

Scalability. In Fig. 11(a), we show the time SWARM needs to find the best mitigation. SWARM's runtime scales linearly with the number of servers. Even on large-scale Clos topologies with 16K servers, SWARM needs less than 5 minutes, a fraction of the mitigation times operators report today [1, 21].

SWARM employs several approximation techniques to scale §3.4. In Fig. 11, we quantify the error and speed-up of these methods compared to a version of SWARM without these approximations. Each technique in §3.4 contributes significantly to speedup: (a) the max-min fair algorithm improves run-time by 36.3× and only introduces $\leq 0.9\%$ error; (b) downscaling traffic by 2× does not introduce additional error but produces 73.6× speedup! (c) adding warm-start and reducing the number of epochs results in 105.7× speedup and $\leq 1.2\%$ error. Future work can further speed this up by increasing the scale factor or reducing the number of epochs.

Sensitivity analysis. We evaluate SWARM's sensitivity to various inputs, including the drop rate estimates and the flow-arrival rates (see §D.1 for details).

As the inputs to the system vary, there are a few inflection points where SWARM is sensitive. These points are where SWARM can make mistakes and pick sub-optimal mitigations



(a) Runtime for different topology sizes (b) Estimation Error of different scalability choices (c) Speed up of different scalability choices

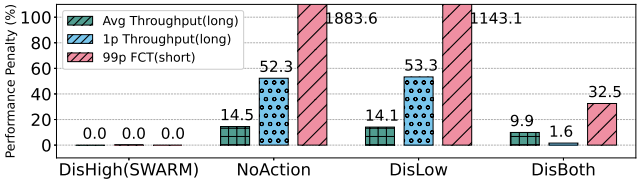
FIGURE 11: Scalability. (a) SWARM scales almost linearly with the number of servers. (b) Error introduced by each of our scaling techniques (approx refers to the ultra-fast max-min fair algorithm). (c) Speed-up from each of our scaling components. We compare with a version of SWARM that does not use the methods from §3.4, instead relies on extended 1-waterfilling [34] to compute max-min fair rates.

if the inputs are noisy. However, we find the difference between the impact of the mitigations is small around these inputs. Outside of these areas, the choice of better mitigation is clear. Thus, SWARM can tolerate large errors in the input distribution: the difference between mitigations is either large enough around that point so the choice of mitigation is clear or it is small enough where a mistake is not too costly.

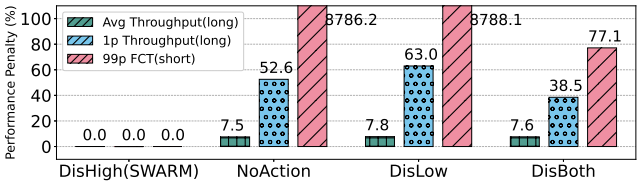
SWARM can also pick the best mitigation under different congestion control protocols (see Fig. A.3). For this, we compare two protocols with different behaviors under loss: (1) Cubic [28], which drastically reduces its sending rate under packet loss, and (2) BBR [12], which does not. SWARM picks the best mitigation irrespective of which protocol we use. However. its approximations of the 1p throughput distribution are more accurate when the mix of protocols is known (it can explicitly account for their differences in handling loss).

Simulation validation. To show SWARM's effectiveness at larger scales with realistic link speeds and latencies, we conducted a simulation using NS3 [29] on a 128-server topology with 20 Gbps 100 μs links. NS3 alone takes over a day to complete one simulation run (one sample). We invested significant effort in parallelizing NS3 using MPI and reduced the time to run one sample to 6 hours. On this topology, we induce a failure in which two links drop packets (one ToR-T1 at 0.005% and one T1-T2 at 0.5%). This scenario shows the complex effect of different packet drops at different levels of datacenters and the trade-off between causing congestion by disabling the links versus incurring packet drops by taking no action. Apart from the DCTCP distribution, we also simulated FbHadoop [54] that has more short flows. These experiments required over 2100 hours to complete.

Fig. 12 shows the performance penalties of different actions. SWARM is able to identify if the congestion introduced by disabling the link would impact the flows more than the packet drop or vice-versa. In contrast, prior work (NetPilot, CorrOpt, and Operator) ignores the impact of traffic and failure characteristics on the mitigation. SWARM finds the best mitigation (only disabling the high drop rate link). In contrast,



(a) DCTCP Traffic Distribution



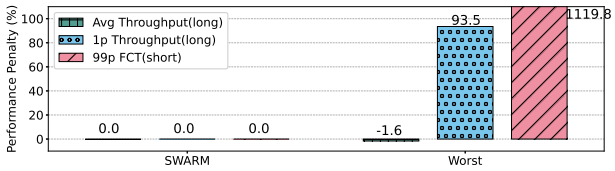
(b) FbHadoop Traffic Distribution

FIGURE 12: NS3 evaluation with different flow size distributions. (Dis=Disable, High=High drop link, Low=Low drop link)

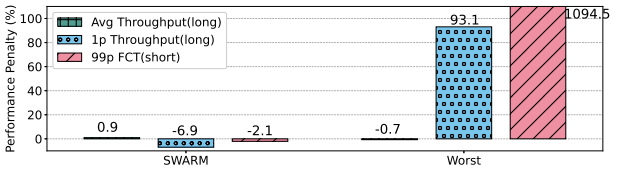
the baselines either disable both links or keep both links and incur 32% – 78% penalty on 99p FCT. Note that the performance penalty of not taking any action is the same as only disabling the link with a low packet drop since they both keep the link with a high drop rate in the network. This link ends up dropping many packets and determines the tail FCT.

Testbed validation. To demonstrate SWARM makes high-quality decisions even on a physical testbed, we induce a failure pattern from Scenario 1 in which a ToR – T1 link randomly drops 6.25% ($\frac{1}{16}$) of the packets, while a link from a different T1 to a T2 also drops packets at 0.39% ($\frac{1}{256}$).

SWARM picks an optimal mitigation for Priority-FCT and a mitigation with less than 1% penalty for PriorityAvgT (Fig. 13). In contrast, the FCT performance penalty for choosing the worst action under PriorityFCT is over 1000%. While the average throughput performance penalty is low across all mitigations in this incident, SWARM picks an action with a low penalty across all three metrics under the PriorityAvgT comparator. It avoids the 93% penalty in 1p throughput and



(a) Priority FCT



(b) Priority AvgT

FIGURE 13: Physical testbed validation with two links dropping packets. We consider four mitigation strategies (disable or take no action). Optimal mitigation has zero penalty.

1095% penalty in 99p FCT for the worst action.

Impact of the comparator. We also observe that SWARM adjusts its decision based on the comparator. Across all the single link failures from Scenario 1 (Table A.1), SWARM takes no action more often under the PriorityAvgT comparator than the PriorityFCT. This is because average throughput is sensitive to the remaining capacity in the network, so SWARM is more likely to keep the lossy link up.

Justifying design choices. We also conducted several experiments to justify the special treatment of loss-limited flows, estimating distributions of FCT and throughput, using multiple epochs to capture flow dynamics, using distributional measures to determine the best mitigation, and the importance of accounting for queueing delay (see §D.3).

5 Discussion and Limitations

SWARM is a first step towards CLP-aware incident mitigation. It has limitations that future work can address.

Support for loss-less transport. Cloud providers increasingly use RDMA [22, 58] for internal traffic. We can modify SWARM's CLPEstimator to (a) detect and account for pauses and (b) model loss-recovery approaches when losses happen.

Support for other routing protocols. SWARM applies to any datacenter that relies on ECMP or WCMP for routing. These are commonly used in Clos topologies. Our experiments show the effectiveness of SWARM on different variants of Clos. Future work can extend this to topologies that use traffic engineering (*e.g.,* direct-connect [49, 61])

Mitigations with transient effects. Some actions introduce transient risk. For example, a switch may drop packets during reboot, causing a non-steady-state impact on long flows.

Approximate failure localization. SWARM waits for operators or automation to localize the failure. It can instead use a spatial failure distribution which is available much sooner. This lowers the mean time to repair. SWARM relies on the correct input of the *type* of failure.

Impact on wide area network traffic. WAN traffic is a small fraction of the datacenter traffic [10, 19], so we ignore its impact. When WAN traffic increases, we would need to account for larger RTT.

Other extensions. Future work can extend SWARM to other metrics such as jitter and failures of software components such as load balancers. It can also account for estimated repair time in ranking mitigations, which can be challenging as incidents with vastly different repair times often have similar symptoms [25]. Future work can also explore the benefits of modeling more than two classes (short and long) of flows.

6 Related Work

Impact or risk estimation in networking. No prior work considers CLP-aware failure mitigation but several estimate the impact or future risk of management operations. Janus [4] focuses on risk estimation for datacenter management operations. RSS [64] estimates the risk for backbone management. TEAVAR [11] and [17, 62] route traffic in production wide area networks to minimize long-term impact. [43] models the risk of demand uncertainty and its impact on revenue in a WAN. Other approaches minimize the impact of failures without explicitly accounting for risk [39, 69].

NetPilot [63] and CorrOpt [71] are closest to SWARM. These can fail to produce mitigations with minimal CLP impact (see §2) because they (a) ignore the failure pattern, (b) do not account for traffic changes, (c) use proxy metrics (*e.g.,* maximum utilization) that only loosely correlate with CLPs, and (d) estimate impact on a healthy network, which limits the set of mitigations they support.

Computing fair share. Prior work assumes flows follow max-min fairness, which matches the objective of TCP [20], and either employ optimization [16, 30, 45, 46, 59], use simulations [68], or iterative algorithms [15, 32, 34, 45, 52, 53] to find the fair share. These approaches are limited; they assume flow rates are only limited by the network capacity and ignore packet drops.

7 Conclusion

Failure mitigation is crucial in running datacenters at scale, but operators often find it hard to choose the right action. SWARM ranks mitigations based on their impact on well-known CLP metrics. Our evaluation shows approximating these mertics is possible at scale and results in mitigations with orders of magnitude lower performance penalty.

Acknowledgments. We thank our shepherd, Sergey Gorinsky, and the anonymous reviewers for their insightful comments. We also thank Ricardo Bianchini, Ranveer Chandra, Jitu Padhye, Solal Pirelli, Rachee Singh and Lihua Yuan for their helpful feedback. This material is based upon work supported in part by the U.S. National Science Foundation under grant No. CNS-1901523.

References

[1] Google cloud incident summary. https://status.cloud.google.com/summary.

[2] Rca - network latency issue – west europe (tracking id 8klc-1t8). https://status.azure.com/en-us/status/history/.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[4] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based planning of network changes in evolving data centers. In *ACM SOSP*, 2019.

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

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[7] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 419–435, 2018.

[8] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 440–453, 2016.

[9] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack Stokes, Geoff Outhred, and Lechao Diwu. Privateeye: Scalable and privacy-preserving compromise detection in the cloud. In *NSDI 20*, 2020.

[10] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[11] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. Teavar: Striking the right utilization-availability balance in wan traffic engineering. In *ACM SIGCOMM*, 2019.

[12] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[13] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1989.

[14] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1–3):1–294, jan 2012.

[15] Emilie Danna, Avinatan Hassidim, Haim Kaplan, Alok Kumar, Yishay Mansour, Danny Raz, and Michal Segalov. Upward max-min fairness. *J. ACM*, 64(1), mar 2017.

[16] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*, pages 846–854, 2012.

[17] Ferhat Dikbiyik, Massimo Tornatore, and Biswanath Mukherjee. Minimizing the risk from disaster failures in optical backbone networks. *Journal of Lightwave Technology*, 32(18):3175–3183, 2014.

[18] A. Dvoretzky, J. Kiefer, and J. Wolfowitz. Asymptotic Minimax Character of the Sample Distribution Function and of the Classical Multinomial Estimator. *The Annals of Mathematical Statistics*, 27(3):642 – 669, 1956.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 51–66, 2018.

[20] S. Ben Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical bandwidth sharing: A study of congestion at flow level. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, page 111–122, New York, NY, USA, 2001. Association for Computing Machinery.

[21] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee, David Maltz, et al. Scouts: Improving the diagnosis process

through domain-customized incident routing. In *ACM SIGCOMM*, 2020.

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

[23] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 216–229, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, aug 2011.

[25] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *ACM SIGCOMM*, 2009.

[28] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[29] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527, 2008.

[30] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[31] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[32] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[33] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1410–1420, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. A distributed algorithm to calculate max-min fair rates without per-flow state. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–42, 2019.

[35] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 281–294, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Frank Kelly. Fairness and stability of end-to-end congestion control. *European journal of control*, 9(2-3):159–176, 2003.

[37] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[38] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming

Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 527–538, 2014.

[40] Chris Lonvick. The bsd syslog protocol. Technical report, 2001.

[41] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1–18, Santa Clara, CA, February 2020. USENIX Association.

[42] Marco Mellia and Hui Zhang. Tcp model for short lived flows. *IEEE communications letters*, 6(2):85–87, 2002.

[43] D. Mitra and Qiong Wang. Stochastic traffic engineering for demand uncertainty and risk-aware network revenue management. *IEEE/ACM Transactions on Networking*, 13(2):221–233, 2005.

[44] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[45] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. Solving Max-Min Fair Resource Allocations Quickly on Large Graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[46] Pooria Namyar, Sucha Supittayapornpong, Mingyang Zhang, Minlan Yu, and Ramesh Govindan. A throughput-centric view of the performance of datacenter topologies. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 349–369, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. SOSP '21, page 521–537, New York, NY, USA, 2021. Association for Computing Machinery.

[48] R. Pan, Balaji Prabhakar, K. Psounis, and D. Wischik. Shrink: A method for scaleable performance prediction and efficient network simulation. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, volume 3, pages 1943–1953 vol.3, 2003.

[49] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *ACM SIGCOMM*, 2022.

[50] Konstantinos Psounis, Rong Pan, Balaji Prabhakar, and Damon Wischik. The scaling hypothesis: Simplifying the prediction of network performance using scaled-down simulations. *SIGCOMM Comput. Commun. Rev.*, 33(1):35–40, jan 2003.

[51] Derek W Robinson. Entropy and uncertainty. *Entropy*, 10(4):493–506, 2008.

[52] Jordi Ros-Giralt, Noah Amsel, Sruthi Yellamraju, James Ezick, Richard Lethin, Yuang Jiang, Aosong Feng, Leandros Tassiulas, Zhenguo Wu, Min Yee Teh, and Keren Bergman. Designing data center networks using bottleneck structures. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 319–348, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Jordi Ros-Giralt and Wei Kang Tsai. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2, pages 717–726. IEEE, 2001.

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

[55] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 595–612, 2017.

[56] Sergey Sarykalin, Gaia Serraino, and Stan Uryasev. Value-at-risk vs. conditional value-at-risk in risk management and optimization. In *State-of-the-art decision-making tools in the information-intensive age*, pages 270–294. Informs, 2008.

[57] Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula, Arjun Radhakrishna, and Anurag Gupta. Autotsg: learning and synthesis for incident troubleshooting. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1477–1488, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.

[59] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, San Jose, CA, April 2012. USENIX Association.

[60] Sucha Supittayapornpong, Pooria Namyar, Mingyang Zhang, Minlan Yu, and Ramesh Govindan. Optimal oblivious routing for structured networks. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1988–1997, 2022.

[61] Min Yee Teh, Shizhen Zhao, Peirui Cao, and Keren Bergman. Couder: robust topology engineering for optical circuit switched data center networks. *arXiv preprint arXiv:2010.00090*, 2020.

[62] Bruno Vidalenc, Laurent Ciavaglia, Ludovic Noirie, and Eric Renault. Dynamic risk-aware routing for ospf networks. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 226–234, 2013.

[63] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM*, 2012.

[64] Yiting Xia, Ying Zhang, Zhizhen Zhong, Guanqing Yan, Chiun Lin Lim, Satyajeet Singh Ahuja, Soshant Bali, Alexander Nikolaidis, Kimia Ghobadi, and Manya Ghobadi. A social network under social distancing: Risk-Driven backbone management during COVID-19 and beyond. In *NSDI 21*.

[65] Nofel Yaseen, John Sonchack, and Vincent Liu. tpprof: A network traffic pattern profiler. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1015–1030, Santa Clara, CA, February 2020. USENIX Association.

[66] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for azure. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 519–532, USA, 2018. USENIX Association.

[67] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. Mimicnet: fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 287–304, 2021.

[68] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. Scalable tail latency estimation for data center networks. *arXiv preprint arXiv:2205.01234*, 2022.

[69] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. Arrow: restoration-aware traffic engineering. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 560–579, 2021.

[70] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

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

A Algorithm Details

A.1 CLPEstimator Function

Alg. A.1 shows the CLPEstimator in detail. It first uses the DKW inequality [18] to compute the necessary number of samples to achieve a certain confidence level. It then adjusts the topology and the traffic based on the mitigation $\mathcal{M}$. Finally, it divides the resulting traffic into short and long flows, and computes their CLPs separately on multiple routing samples (§3.3).

Algorithm A.1: CLPEstimator Function.

Input: $\mathcal{T}$. traffic matrix (src, dest, start time, size).
Input: G. network state (location, type, failures).
Input: $\mathcal{M}$. mitigation.
Input: α. confidence threshold
Output: $\beta_l = \{\beta_l^{\mathcal{R}}\}$. $\beta_l^{\mathcal{R}}$ is the distribution of impact over all the long flows for routing sample $\mathcal{R}$.
Output: $\beta_s = \{\beta_s^{\mathcal{R}}\}$. $\beta_s^{\mathcal{R}}$ is the distribution of impact over all the short flows for routing sample $\mathcal{R}$.

1 $N \leftarrow$ num_samples(α)
2 $G_a, \mathcal{T}_a \leftarrow$ apply_mitigation($G, \mathcal{M}, \mathcal{T}$)
3 $\mathcal{T}_s, \mathcal{T}_l \leftarrow$ split_traffic($\mathcal{T}_a$)
4 **for** $n \in 1 \ldots N$ **do**
5 $\mathcal{R}_n \leftarrow$ get_routing_sample(G_a)
6 $\beta_l^{\mathcal{R}_n} \leftarrow$ impact_long_flows($G_a, \mathcal{T}_l, \mathcal{R}_n$)
7 $\beta_s^{\mathcal{R}_n} \leftarrow$ impact_short_flows($G_a, \mathcal{T}_s, \mathcal{R}_n$)
8 **end**
9 **return** β_l, β_s

A.2 Demand-Aware Max-Min Fair

CLPEstimator computes the impact of a given mitigation $\mathcal{M}$ on the throughput of long flows in two steps. (1) it estimates the drop-limited throughput for each flow (Line 1 in Alg. A.2), assuming there is no congestion and the packet drop enforces the maximum possible rate. (2) CLPEstimator computes the max-min fair rate of each flow and enforces these drop-limited throughputs as an upper limit on the rate (demand) of each flow (Line 2). This ensures that a flow does not receive more than its drop-limited throughput unless it is limited by its fair share when congestion is more severe than the drop rate.

Algorithms to compute max-min fairness [34] assume demands are unbounded and are only limited by the network capacities. In SWARM, we develop a demand-aware extension of these algorithms that can enforce limits on each flow's rate (Alg. A.3). To achieve this, we augment the topology and add one virtual edge per flow with capacity set at the drop-limited rate. Then, we use existing algorithms to compute the network-wide max-min fair rates on this augmented topology. We can use the same method to enforce congestion control rate limits in the first few epochs of each flow in cases where the protocol's congestion window limits the flow's rate.

We can use any of the variants of the k-waterfilling algorithm [34] in the network-wide-max-min-fair function in Line 6 (but some can result in O(|E| + |F|) number of iterations where |E| is the number of edges and |F| the number of flows). Instead, we use a faster variant of max-min fairness [45] that speeds up k-waterfilling by 30× with minor degradation in the quality of the estimated rates (Fig. 11(c), Fig. 11(b)).

Algorithm A.2: Compute throughput.

Input: G. current network state.
Input: $\mathcal{T}_{1\times n}$. source destination pairs (long flows).
Input: $\mathcal{R}_{1\times n}$. the (sampled) routing for each flow.
Output: F. max-min fair rate

1 $\theta \leftarrow$ get_drop_limited_throughput($G, \mathcal{T}, \mathcal{R}$)
2 $F \leftarrow$ demand_aware_max_min_fair($G, \mathcal{T}, \theta, \mathcal{R}$)
3 **return** F

Algorithm A.3: Demand-Aware Max-Min Fair.

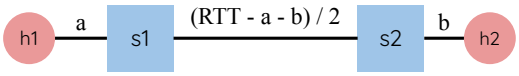
Input: G. current network state.
Input: $\mathcal{T}_{1\times n}$. source destination pairs (long flows).
Input: $\mathcal{R}_{1\times n}$. the (sampled) routing for each flow.
Input: $\theta_{1\times n}$. the drop-limited rate for each flow.
Output: $\hat{\theta}_{1\times n}$. max-min fair rates after applying congestion.

1 **for** $\forall f \in \mathcal{T}$ **do**
  /* add one virtual link per flow */
2 $e \leftarrow G$.add_virtual_edge()
  /* set capacity to drop-limited rate */
3 e.capacity $= \theta_f$
  /* add to the route of flow f */
4 $\mathcal{R}_f$.add(e)
5 **end**
6 $\hat{\theta} \leftarrow$ network_wide_max_min_fairness($G, \mathcal{T}, \mathcal{R}$)
7 **return** $\hat{\theta}$

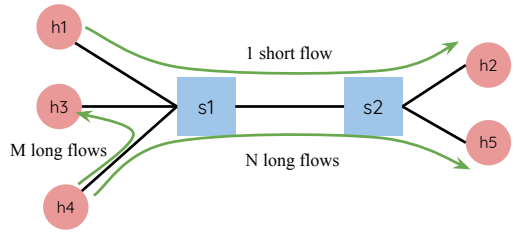
B Details of Offline Measurements

We measure three empirically driven distributions offline and use them to estimate CLPs in SWARM: (1) the upper bound on the throughput of long flows in a lossy network, (2) the necessary number of RTTs to deliver short flows, and (3) the queueing delay. In this section, we describe how we gathered these distributions. We also show a sample in Fig. A.8.

Throughput of long flows in a lossy network. We use Topol-



(a) Topology 1



(b) Topology 2

FIGURE A.1: Topologies for offline measurements.

ogy 1 in Fig. A.1 where h1 sends a long flow f to h2. We use iperf3 to measure throughput under different network conditions (*e.g.,* by inducing packet drops in s1 – s2 or changing the RTT by adjusting the delay on s1 – s2). We ensure that link capacities are sufficiently high so that the drop rate always dictates the rate. We also repeat each experiment multiple times to create a statistically significant distribution [18].

Number of RTTs for short flows. We send a short flow from h1 to h2 in Topology 1 (Fig. A.1) under different settings (*e.g.,* flow size, slow start threshold, initial congestion window) and network conditions (*e.g.,* packet drop rate, RTT). We repeat each experiment multiple times, measure the FCT of the flow in each experiment, and construct an FCT distribution. We ignore the queueing delay since we only have one flow in this experiment and divide the FCT distribution by the two-way propagation delay to derive the necessary number of RTTs.

Queueing delay for short flows. Queueing delay is harder to measure as it depends on network load and the number of competing flows. We use Topology 2 in Fig. A.1 to estimate the queueing delay distribution under various network conditions. We route M long flows from h4 to h3 and N additional long flows from h4 to h5. After these long flows reach the steady state, we route a flow small enough to finish within an RTT from h1 to h2 and measure its FCT.

In this experiment, we can control the number of flows that compete on the s1 – s2 link using N and the utilization of the s1 – s2 link using M and N. The latter is because the h4 – s1 is the main bottleneck in the network and determines the rate of the N long flows that go through s1 – s2. We vary both M and N to model different network conditions and measure the FCT of the short flow. We subtract the two-way propagation delay from the FCT to derive the queueing delay. Once again, we repeat these experiments to ensure statistical significance in the resulting distributions.

C Additional Experiment Details

In this section, we describe the details of how we evaluate SWARM using emulation, simulation, and testbed experiments on a set of representative and common incidents.

C.1 Traffic Traces

We evaluate our approach using many traffic traces to ensure our results are reliable. Each trace includes several network flows, each of which has 4 pieces of information: source, destination, size, and start time. We ensure these traces are long enough to capture the impact of mitigation on a wide range of flows with various sizes. To prevent capturing the impact of an empty network, we discard the performance of flows that begin in an initial window of the trace. Note that the duration of this window and the trace itself depend on the network bandwidth and delay in each setup.

Flow sizes. We follow recent works [38, 65] and sample flow sizes from a well-known and widely used distribution from DCTCP [5].

Flow start time. We generate flow start times using a Poisson distribution [35,38,41,61] with inter-arrival time derived from Azure to ensure the load on the network is reasonable (unless otherwise specified).

Source and destination. We follow [38] in picking the source and destinations.

C.2 Failures Types

We evaluate our approach on different types of failures (see Table 2). These failures (*e.g.,* packet drop on multiple links or capacity drop due to fiber cut) are common in Azure and other cloud providers, as several studies confirm [7, 24, 31, 55, 63, 71].

We use Mininet (a network emulator) as our primary evaluation framework for the reasons outlined below. We provide specific details on the failure cases in Table A.1. These 57 cases represent a wide range of potential issues that could arise in a datacenter. For instance, we evaluate two scenarios per packet drop rate for single-link failure incidents, which cover all possible single-link failures that can occur in a data center. This is because (a) the target link is either between ToR and an aggregation or between an aggregation and a core switch [3], and (b) Clos is a symmetric topology. Therefore, all ToR-aggregation and core-aggregation links are equivalent [4]. This means it is enough to only look at two cases, which are representative of a wide range of possible incidents.

The same logic applies to other cases. For example, one of our two-link failure scenarios considers the case where two ToR-aggregation links in the same pod drop packets. This scenario represents any two ToR-aggregation links that may fail in any of the pods over the entire datacenter.

C.3 Emulation, Simulation, and Testbed

Emulation Setup (Mininet). Mininet uses a real, production-grade TCP/IP stack from the Linux kernel. This allows us to evaluate our methods on production-grade implementations of congestion control algorithms such as BBR [12] or Cubic [28]. However, scaling Mininet is challenging [52]. We use the Clos

		Details	#scenarios
Scen. 1	1 link failure	One T0 – T1 and one T1 –T2, each with two levels of packet drop rate ∼5% and 0.005%	4
	2 link failures	Four combination of pair of links (two T0 – T1s in the same cluster connected to the same T0, two T0 – T1s in the same cluster connected to different T0s & T1s, one T0 – T1 & one T1 – T2 connected to different T1s, and two T1 – T2s connected to different T1s & T2s), each with all combinations of the two packet drop levels and all the possible ordering of failures.	32
Scen. 2	no other link failure	one T1 – T2 with capacity reduced to half	1
	1 other link failure	one T1 – T2 with capacity reduced to half, another T0 –T1 link with 3 levels of failure (two packet drop levels + completely down), and all possible ordering of failures.	6
Scen. 3	no link failure	One T0 with two levels of packet drop rate ∼5% and 0.005%	2
	1 link failure	One T0 and One T0 – T1 in the same cluster connected to a different T0, with all combinations of packet drop rates (ToR with two levels of packet drop rate and link with three levels of failure (two packet drop rates + completely down)), and all the possible ordering of failures.	12
		total number of evaluated scenarios	57

TABLE A.1: Mininet Experiment Details.

topology from Fig. 2 with 8 servers, 4 ToRs, 4 aggregation switches (T1), and 4 spine switches (T2). Even with this setup, we required a server with 64 cores and 256 GB memory to achieve the desired load level. We include results from *4000 hours experiments (over 5 months)* to cover all 57 scenarios, all combinations of possible mitigations, and running each multiple times.

Emulating a network with gigabit-level bandwidth (40 Gbps) and microsecond-level propagation delay (50 μs) is impossible in Mininet. Therefore, we downscale the traffic and the link properties following [48, 50]. This involves (1) reducing the link capacities and increasing the link delay to ensure the network bandwidth-delay product remains the same and (2) increasing the inter-arrival time between flows by the same factor (in our case, 120×). Our testbed and simulation experiments compensate for this.

We tested our approach with both Cubic [28] and BBR [12] congestion control algorithms in Mininet (and DCTCP [5] in our NS3 simulations). Our findings suggest that the choice of mitigation depends less on the congestion control, and a common abstraction that ensures these flows receive max-min fair rates is enough (TCP objective [20]).

For our evaluation, we use traffic traces that last 500 seconds (around 9 minutes) to capture enough samples from different flow sizes. We only measure the impact on flows that start within [50, 150) seconds to avoid including the start-up phase and to ensure all the captured flows finish before the end of the traffic trace. We repeat each experiment for 30 different traffic traces (§C.1).

Simulation Setup (NS3). We use a Clos topology with 16 cores (T2), 32 aggregations (T1), 32 ToRs, and 128 servers. Each link has 20 Gbps bandwidth and 100 μs propagation delay. We use DCTCP [5] as the congestion control algorithm.

We consider an example where two links drop packets (one at a high rate ∼0.5% and the other one at a low rate ∼0.005%). This scenario captures the impact of both the drop rates and the network load on the decision. We also find the packet drop rate impacts the scalability of the simulation. Thus, we had to slightly reduce it compared to our Mininet experiments. Even with this drop rate, it takes NS3 one day to compute the performance of one mitigation on a single traffic trace. We use MPI to speed up NS3 and reduce the runtime.

Each of our traffic traces is 10 s long, and we measure the impact on flows that start within [0.5, 1) s. We repeat each experiment for 30 different traffic traces.

Testbed Setup. Our testbed has 32 servers connected using a Clos topology with six ToRs, four aggregation switches (T1), and two core switches (T2). Each link in our testbed has 10 Gbps bandwidth and 200 μs propagation delay. The topology is a Clos [3] where all T1 and T2 switches are connected to each other (different from Mininet and NS3 simulations).

Due to hardware limitations, we can only inject packet drops at the power two. We consider an example where we have two links dropping packets (one at a high rate $\frac{1}{16}$ and the other one at a low rate $\frac{1}{256}$).

C.4 Other Details

SWARM. We use 32 traffic traces and 1000 routing samples in SWARM (unless mentioned otherwise). For our baseline comparisons in Mininet, we set the duration of each traffic trace to 200 s, and as in Mininet, we compute the CLP metrics over all the flows that start within [50, 150) s. We also use 200 ms as the epoch size for Alg. 1. Note that an ideal epoch size should be in the order of the flow inter-arrival time (1 ms in Mininet). However, we find that SWARM can still find good mitigations even when we use a much larger epoch size.

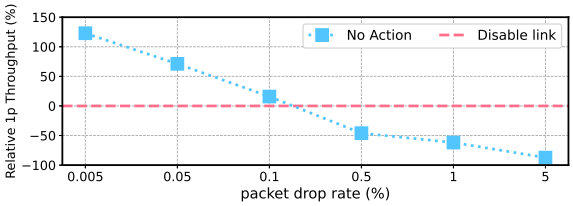
Routing. We assume either ECMP or WCMP routing [27, 70], which are typical in data centers.

D Extended Evaluation

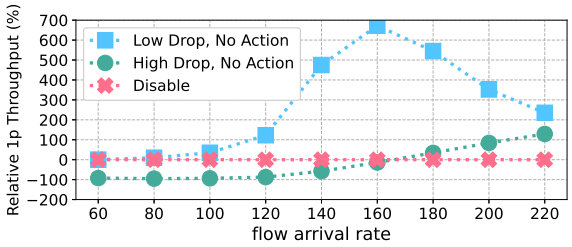
In this section, we present an extended evaluation of SWARM.

D.1 Sensitivity to Inputs

Packet drop rates. We compare the relative difference between the 1p throughput of taking no action and disabling a



(a) Sensitivity to packet drop rate



(b) Sensitivity to flow arrival rate changes

FIGURE A.2: Evaluating sensitivity of SWARM to errors in the input packet drop rate and flow arrival rate using Mininet. All experiments are in a scenario where a T0 – T1 drop packets.

link in a failure scenario where a T0 – T1 link drops packets at different rates (Fig. A.2). We find the choice of the right decision to be bi-modal with a wide room for error. It is better to take no action for all the drop rates below $\sim 0.1\%$ while the best action is to disable the link beyond that point. Also, the penalty close to this transition point (0.1% drop rate) is rather small. In other words, the error in the input packet loss rate has to be an order of magnitude for SWARM to make the wrong decision, an unlikely possibility in today's clouds.

Flow arrival rates. We investigate how SWARM's decisions change under two failure severities and different flow arrival rates (Fig. A.2). We observe that the gap between the two decisions is significant outside of a few inflection points, which means SWARM will pick the best mitigation in most cases.

For instance, in the high drop rate scenario, disabling the link is better than taking no action for arrival rates ranging from 60 to 160 flows per second (fps). Taking no action is better once we pass 160 fps as bringing down the link causes network congestion. This means SWARM has a wide margin of error. At small and large arrival rates, the difference between the two actions is consistently large, which means SWARM would pick the right action. At medium arrival rates, the difference is very small, so whichever action SWARM picks results in almost the same impact.

D.2 Sensitivity to Congestion Control

We conduct a limited experiment to evaluate whether SWARM is resilient to the choice of congestion control. We consider a scenario where a T0 – T1 and a T1 – T2 are dropping packets at low and high drop rates, respectively. We use two example congestion control protocols (CUBIC [28] and BBR [12]), which behave differently under loss. Cubic significantly reduces its rate under loss while BBR does not.

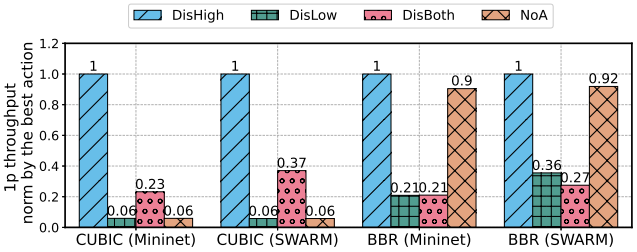
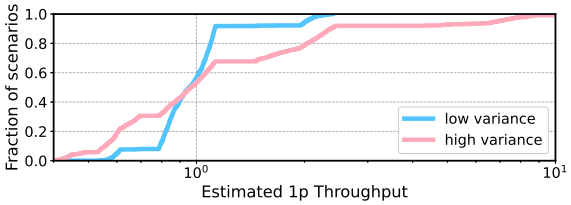


FIGURE A.3: Evaluation of SWARM on multiple congestion controls in Mininet. In general, SWARM is able to correctly order the mitigation actions and approximately capture the relative difference between different mitigation actions on both congestion controls (BBR [12] and Cubic [28]).



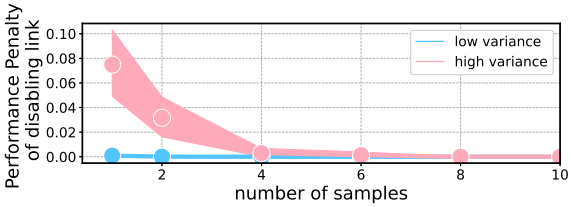


FIGURE A.4: Variance in the input flow arrival rate and its impact on the performance penalty of mitigation actions in Mininet.

We compare SWARM's estimated 1p throughput with Mininet under four mitigations (Fig. A.3). We find (1) SWARM can estimate throughput adequately enough to pick the best mitigation and (2) the choice of the best action remains independent of the congestion control protocol. SWARM's estimates are more accurate when it operates on the same congestion control protocol. This means it is more robust when operators correctly input the likelihood of different protocols being used inside their datacenters.

D.3 Assumptions and Design Choices

We also validate our assumptions and design choices.

Capturing the impact of drop-limited flows. We account for drop-limited flows by computing an upper bound on their rate and then enforce these bounds during max-min fair rate computations. We evaluate this assumption in an experiment where a single link delivers a varying number of flows and drops packets at different rates (Fig. A.5). We observe our observation is valid: each flow takes the minimum of its fair share rate and its drop-limited throughput.

We next evaluate our design choices for estimating the

distribution of the flow completion time and throughput (Fig. A.5). Specifically, we show the quality of SWARM's approximations compared to Mininet measurements. We use a scenario where two links drop packets at different rates, and the mitigation is to disable the link with the higher drop rate.

Single Epoch vs. Multiple Epochs. In SWARM, we use epochs to capture the shift in network bottlenecks as flows arrive or depart over time. We observe that not capturing these dynamics leads to more than 50% estimation error on average (this is equivalent to running SWARM with only one epoch). Instead, SWARM uses multiple epochs to model flow arrival/departure and avoids this error.

Distributions vs deterministic estimates. In SWARM, we pick the best mitigation based on CLP distributions. This method increases confidence in the final decision and has less error compared to relying on a single CLP sample. SWARM can efficiently compute these distributions in parallel.

Queueing Delay. We show the importance of accounting for queuing delay using an example scenario where the link C0 – B0 in the Fig. 2 topology drops packets at a high drop rate and the operator prioritizes 99p FCT of short flows. In this case, the best action is to disable the link. After we apply the mitigation, the link C0 – B1 starts to drop packets at a high drop rate. Disabling C0 – B1 is not feasible anymore as it partitions the network, so we have to either take no action or bring back C0 – B0. If we ignore the queuing delay, both of these options have the same approximate 99p FCT. However, bringing back C0 – B0 is a better mitigation since it increases the path diversity and reduces the queuing delay (Table A.5).

D.4 Results for Other Comparators

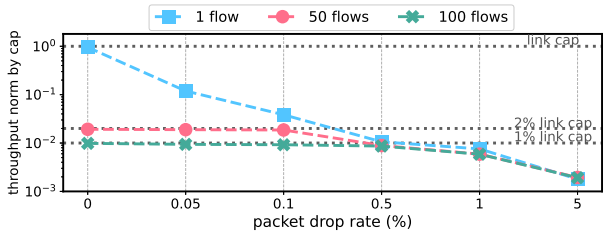
Priority1pT. This comparator minimizes the 1p (1st percentile) throughput. It uses two tiebreakers in the following order: average throughput and 99p FCT.

Linear combination. This comparator minimizes a weighted combination of the three CLP metrics (99p FCT, 1p throughput, and average throughput);

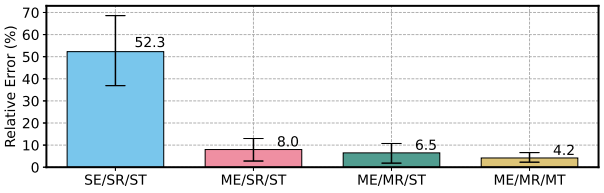
$$w_0 \frac{\text{99p FCT}}{\text{99p FCT}_\text{h}} + w_1 \frac{\text{1p Thru}_\text{h}}{\text{1p Thru}} + w_2 \frac{\text{avg Thru}_\text{h}}{\text{avg Thru}}$$

where w_i is the assigned weight and Metric_h is the metric measured in a healthy network. Note that we prefer lower FCT but higher throughput. Therefore, we use their inverses in our definition of the linear comparator. SWARM admits any combination of weights, but we evaluate for the case where all weights are set to 1. This is different from all the other comparators as it does not set any preference for any metric.

Summary of the results. Fig. A.6 and Fig. A.7 compare the performance penalty of SWARM against other baselines across the three types of failures for Priority1pT and Linear combination comparator. In summary, SWARM's performance penalty is low across all the scenarios and metrics.



(a) Drop-limited vs Capacity-limited



(b) Impact of different design choices on throughput estimation (E=Epoch, R=Routing Sample, T=Traffic Sample, S=Single, M=Multiple). For example, SE/SR/ST refers to using a single epoch, with a single routing sample, and a single traffic sample. We run each method with 10 different seeds.

Approach	Best Action	FCT Penalty
Ignore Queueing	Disable C0 – B0	48 %
Model Queueing	Bring back C0 – B0	0 %

(c) How queueing delay influences the mitigation choice.

FIGURE A.5: Validationg assumptions and design choices using Mininet. (a) shows flows are capacity- or loss-limited. A flow is loss-limited when its rate drops below its fair share of the link capacity (marked as dashed lines) (b) shows the impact of our design choices and the relative error with respect to Mininet. (c) the importance of accounting for queuing delay.

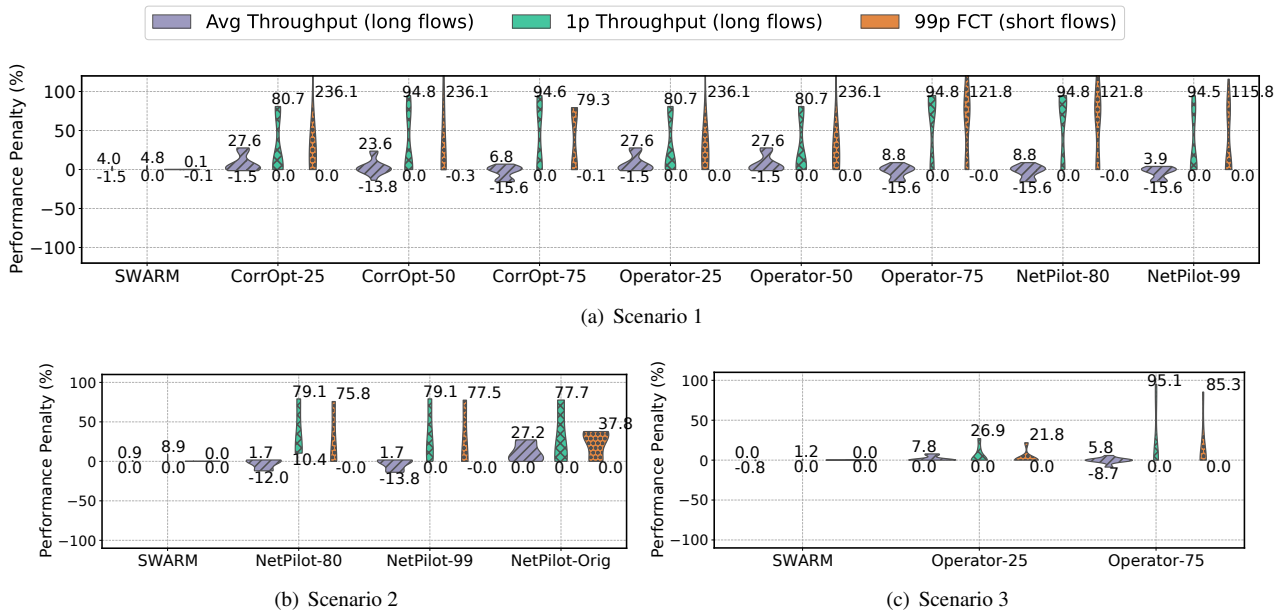
E Disabling the Congested Device

The routing protocols such as ECMP ignore the asymmetry in the datacenters, which can cause congestion. We can mitigate this by disabling the congested link or device so the routing can utilize other paths. For instance, each logical link between two switches [63] consists of multiple physical links. A control protocol multiplexes the packets over these physical links by hashing the packet's header. When a cut happens in any of these physical links, the link capacity decreases and can cause congestion on the remaining physical links. To mitigate this, we can disable the logical link to enable ECMP/WCMP to use other paths with healthy links.

F SWARM's Benefits

The reason behind SWARM's benefit depends on the scenario. In some incidents, SWARM chooses the actions that are already supported but ignored by prior work. In some other cases, the benefit is due to the larger action space of SWARM. We show this using two example failures on the topology in Fig. 2.

Scenario 1 (same action space). In this case, a link between a tier-0 and tier-1 switch start dropping packets (*e.g.,* C0 – B1). Existing threshold-based methods [63, 71] will take no



(a) Scenario 1

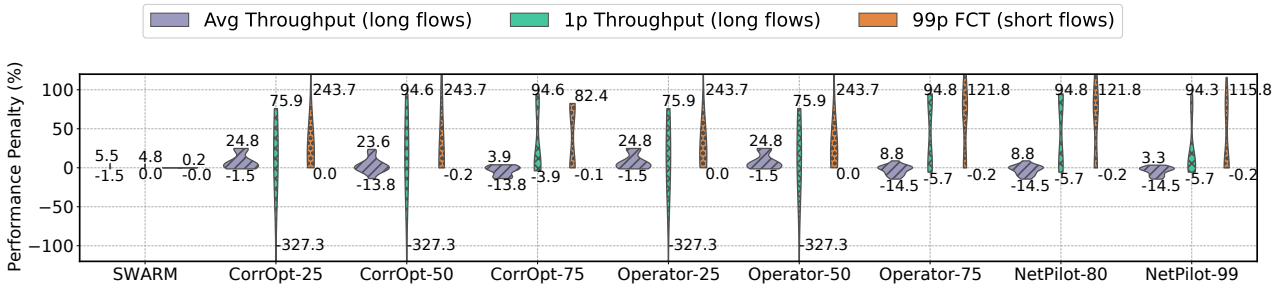
(b) Scenario 2

(c) Scenario 3

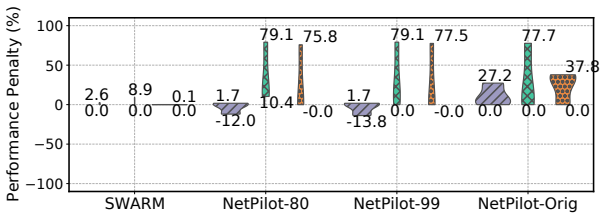
FIGURE A.6: SWARM outperforms other baselines under Priority1pT comparator in Mininet. It is the only method with a low penalty across all the metrics and all the scenarios.

action if they use a large threshold. This decision process completely ignores the impact of the packet drop rate. If the packet drop rate is high, we should disable the link. If these threshold-based methods use a smaller threshold, they end up always disabling the link. This decision completely ignores the impact of traffic load and the packet drop rate. Therefore, it can cause severe congestion. SWARM takes all these factors into account and finds much more effective mitigations (even though the action space is the same).

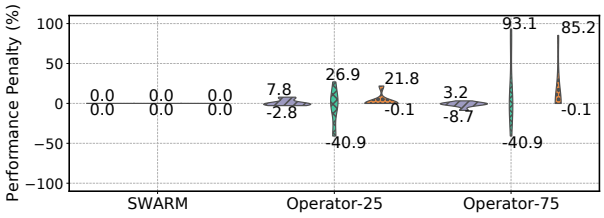
Scenario 2 (larger action space). Imagine a sequence of failures. First, a link between a tier-0 and tier-1 switch starts dropping packets at a moderate rate. SWARM would decide to disable the link. Later, another link between the same tier-0 switch and another tier-1 switch starts dropping packets at a much higher rate. In this case, SWARM disables the new link and re-enables the old link (undoing its previous action). This is because disabling both links would reduce the capacity of the network substantially, causing severe congestion. SWARM can reason about these cases, which enables exploring a broader set of actions, such as bringing back less faulty links.



(a) Scenario 1



(b) Scenario 2



(c) Scenario 3

FIGURE A.7: SWARM outperforms other baselines under Linear combination comparator in Mininet. SWARM consistently achieves low penalty (always $\leq 8.9\%$) across all the metrics and all the scenarios.

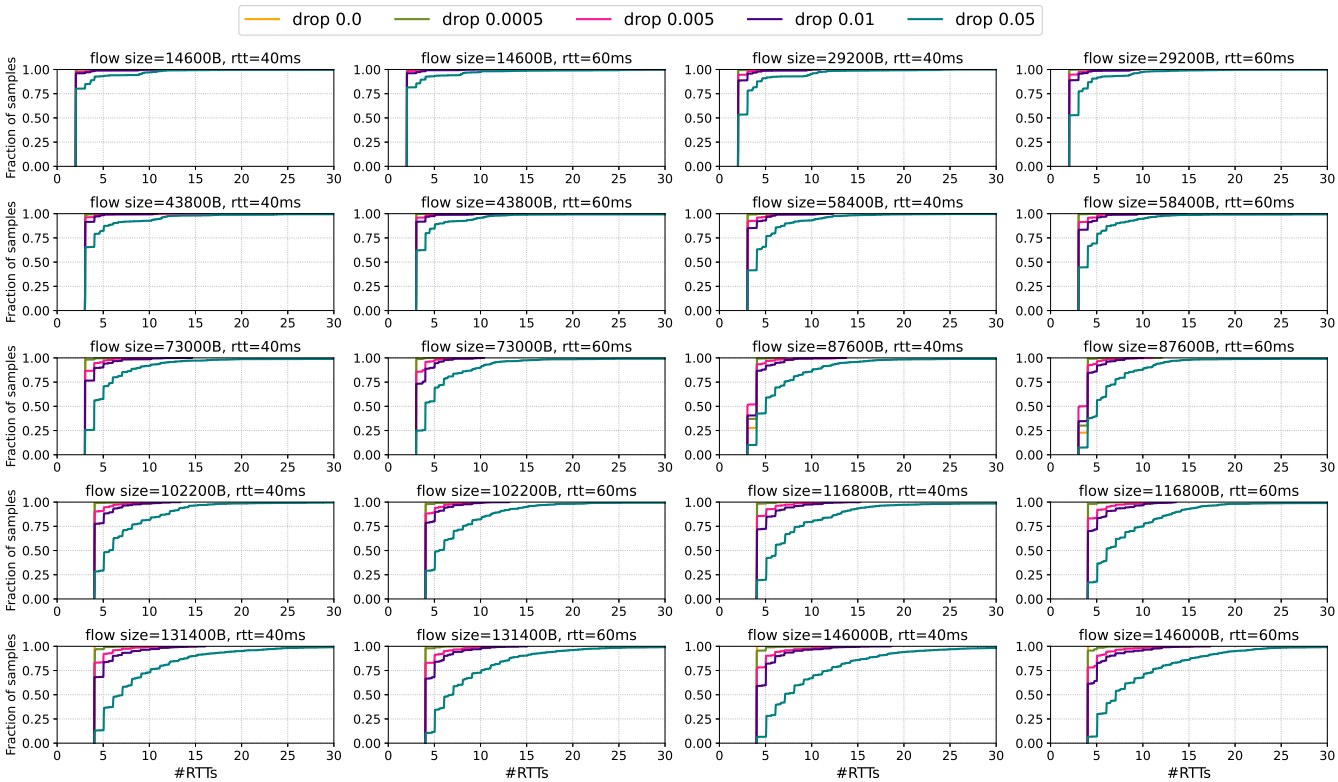


FIGURE A.8: An example of distribution measured for short flow's FCT using Mininet.

One-Size-Fits-None: Understanding and Enhancing Slow-Fault Tolerance in Modern Distributed Systems

Ruiming Lu[1,2] Yunchi Lu[1] Yuxuan Jiang[1] Guangtao Xue[2] Peng Huang[1]

[1]*University of Michigan* [2]*Shanghai Jiao Tong University*

Abstract

Recent studies have shown that various hardware components exhibit fail-slow behavior at scale. However, the characteristics of distributed software's tolerance of such slow faults remain ill-understood. This paper presents a comprehensive study that investigates the characteristics and current practices of slow-fault tolerance in modern distributed software. We focus on the fundamentally nuanced nature of slow faults. We develop a testing pipeline to systematically introduce diverse slow faults, measure their impact under different workloads, and identify the patterns. Our study shows that even small changes can lead to dramatically different reactions. While some systems have added slow-fault handling mechanisms, they are mostly controlled by static thresholds, which can hardly accommodate the highly sensitive and dynamic characteristics. To address this gap, we design ADR, a lightweight library to use within system code and make fail-slow handling adaptive. Evaluation shows ADR significantly reduces the impact of slow faults.

1 Introduction

The reliability of distributed systems hinges on their robust fault tolerance mechanisms. While deployed distributed systems today have been designed to tolerate crashes well using techniques such as state-machine replication, recent studies [36, 50] show that, *fail-slow* behavior is prevalent in different hardware components (*e.g.*, disk, NIC, switch) of systems at scale. For example, a NIC can suddenly experience a 50% packet loss. Such behavior occurs due to firmware bugs, driver bugs, external temperature, etc., and leads to severe incidents.

These studies have motivated a few recent research solutions [49, 54, 57, 64]. For example, Copilot is a consensus protocol designed to tolerate one slow replica by introducing proactive redundancy and parallel processing [54]. Developers also become increasingly aware of this overlooked failure mode. For instance, HBase developers introduced an enhancement feature [9] to detect and mitigate slow syncs.

Despite the progress, the characteristics of slow-fault tolerance in distributed systems remains ill-understood. The most notable existing insight comes from a 2013 study [32], which shows that slow hardware can cause severe cascading failures and "limplock" of the entire cluster. This study, while illuminating, focuses on demonstrating the existence of limplock in *worst-case* scenarios. For example, it demonstrates that when the disk throughput on the HDFS master node is permanently slowed down by 1000×, HDFS experiences a cluster limplock. Moreover, the landscape of both hardware and software has evolved significantly since 2013. Hardware is now more powerful, holds larger capacities, and integrates new technologies. Software adopts asynchronous programming, event-driven designs, chaos engineering, observability tools, *etc.* These changes have potentially altered the status quo.

To address this gap, this paper presents a study that comprehensively examines how modern distributed software reacts to slow faults. Our goal is to provide insights for developers to enhance their systems' slow-fault tolerance. The key challenge developers face is that, unlike crash faults, slow faults are not binary but rather have great varieties. Even defining slow tolerance is not easy. Indeed, when a hardware component has a slow fault, the software execution inevitably gets impacted. For distributed software, it can leverage its redundancy and concurrency to minimize the overall performance degradation.

Our study thus focuses on examining the nuances of fail-slow tolerance. We develop a testing pipeline that uses fault injection to systematically explore various slow faults and measure how a system reacts to the varieties. We also examine the current practices of slowness detection and mitigation.

Our study reveals many interesting findings. For example, we find that the sensitivity to different configurations of slow faults varies significantly across the studied systems. In some systems, a 1 ms network slow fault can cause 25% performance degradation, while other systems need a 100 ms delay to show a similar level of performance degradation. In almost all studied systems, we observe the existence of a *danger zone*, where a slight increase in slow-fault severity results in a significant increase in degradation. However, the range of these danger zones is often narrow, which poses challenges for early alerting. Moreover, a milder slow fault can cause more harm than a severe one. We also find that while the latest solutions or recent versions of a mature system have added slow handling mechanisms, these mechanisms are not effectively triggered.

A unifying theme in our study's findings is that slow faults not only are non-binary but also have highly dynamic and sensitive impacts (depending on workload, severity, duration, etc.). However, developers' existing efforts only try to address the non-binary nature but miss the dynamic and sensitive characteristics. The existing slowness handling mechanisms are controlled by static, over-conservative thresholds, which are only triggered by the most severe slow faults. Even when

System	Language	Version	Benchmarks
Cassandra	Java	4.0.10	YCSB [30]
HBase	Java	2.5.6	YCSB
HDFS	Java	3.3.6	MRBench, TeraSort
etcd	Go	3.5.10	YCSB, v3 benchmark [6]
CRDB	Go	23.1.11	YCSB, sysbench [19]
Kafka	Java, Scala	3.5.0	OpenMessaging [12]

Table 1: Studied systems. *CRDB*: CockroachDB.

the thresholds are exposed as parameters, it is hard for users to tune these parameters and they cannot be changed dynamically once set. Fail-slow handling needs to be made more dynamic.

Motivated by this takeaway, we design ADR, a lightweight *adaptive* fail-slow handling library that developers can easily use as a plug-in in their *code*. ADR will replace the static logic with an adaptive one and determine the levels of mitigation actions. ADR is robust against varying slow faults and workloads by monitoring both the value and update frequency of traced variables. We implement ADR with two distributed systems. We demonstrate that by using this library, the system performance degradations under different slow faults and different workloads are significantly reduced by 65% on average with low overhead, compared to using static thresholds.

Besides code enhancement, developers start to introduce test cases when they add slowness handling mechanisms. However, they only perform unit tests that use a worst-case setting to validate the basic code correctness. The effectiveness of these mechanisms in practice remains unchecked. Thorough fault injection testing is needed to measure the realistic effectiveness. The testing pipeline we develop addresses this gap and provides a continuous measurement tool for developers to understand and improve their systems' slow-fault tolerance level.

Our testing pipeline and ADR are available at `https://github.com/OrderLab/xinda`.

2 Methodology

We examine six large-scale distributed systems (Table 1). We choose them because they are representative and widely deployed in many companies. They are developed in different programming languages and support critical services, including distributed storage, databases and streaming. We select their latest stable versions at the time of our study.

We also study three state-of-the-art *solutions* tackling slow faults. We examine their effectiveness in Section 5.

Benchmarks. Measuring slow-fault tolerance requires representative workloads. Prior work uses manually crafted microbenchmarks (e.g., creating thousands of empty files to exercise the logging protocol) [32]. Such benchmarks provide localized views of individual features in the system. However, deployed systems receive many more complex and mixed requests. Thus, we use comprehensive benchmarks to better reflect the system responses under real deployments.

Table 1 lists the benchmarks we use to measure each system. For database systems, we use YCSB [30] to generate workloads with distinct access patterns, including read-only, write-only, and mixed reads and writes. For etcd, we also use the official benchmarking tool [6] to test other operations (*e.g.*, range queries and transactions) and etcd-specific requests (*e.g.*, leases and watches). For CRDB, we also use sysbench [19], a popular benchmark for relational database systems, to test delete, insert and range query operations. For HDFS, we use MRBench and TeraSort: MRBench executes a set of MapReduce jobs while TeraSort evaluates the system's capacity to sort large volumes of data. For Kafka, we use OpenMessaging [12] to generate message streams under different configurations (*e.g.*, partitions, batch size, and linger time).

Load stress. We adjust the benchmark load intensity to ensure all nodes are highly utilized: if one node crashes, the throughput of the remaining nodes does not increase significantly. We achieve this by tuning the concurrency levels of the benchmarks (*e.g.*, thread count in YCSB).

Fault Injection. Prior studies [32, 36, 49] show that slow faults can originate from hardware, software (*e.g.*, drivers), or environment issues. Regardless of their root causes, slow faults will ultimately affect a distributed system through the underlying interfaces. Thus, instead of injecting hardware-level faults, we inject slowness directly to the filesystem and network layer to minimize influences of uncontrollable factors. We use blockade [2] to inject network faults and charybdefs [3] to inject filesystem ones (Appendix A).

Since slow faults have wide varieties, we examine them by the following fault attributes:

• *Severity.* We study three major types of slow faults: network packet loss, network delay, and filesystem delay.

For packet loss, we choose 1%, 10%, 40%, and 70%. This is evidenced by prior empirical studies in data center networks and Internet service providers (ISPs) [36,52,61]: faulty links/devices could lead to 2% to 100% packet loss in Microsoft Azure [61]; 1-50% packet loss is observed from 12 institutions including NetApp and Twitter [36]; 0.1% to 100% packet loss is used to test a gray failure detector in ISPs [52].

For network delays, we choose 100 us, 1 ms, 10 ms, 100 ms, and 1 s. This is backed by a study from Microsoft data centers [37], where the tail network latency varies from several milliseconds to hundreds or even thousands of milliseconds.

For filesystem delays, we choose 1 ms, 10 ms, 100 ms, and 1 s. This is based on an empirical study on fail-slow disks in Alibaba data centers [50], where faulty device latency ranges from hundreds of microseconds to nearly a second.

• *Location.* We inject faults to only one node per experiment. For leader-based systems, faults are injected to either the follower or the leader: master or regionserver for HBase, namenode or datanode for Hadoop, leader or follower node for etcd. Although there is a concept of *leader* in CRDB (one leaseholder per range) and Kafka (one leader per partition), such leader roles are dynamic and are changing at runtime; for Cassandra, node roles do not differ after cluster initialization (gossiping). Thus, we inject faults to only one (arbitrary) node of the CRDB, Kafka, and Cassandra clusters.

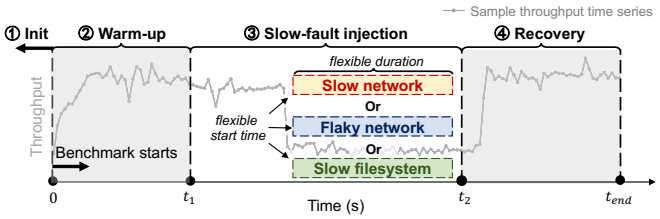


Figure 1: Our slow-fault tolerance testing pipeline.

Multiple slow faults could occur simultaneously. However, prior work suggests that fail-slow is a rare yet independent event [49, 57]. For example, Alibaba monitors 248 K SSDs for 10 months [49], only to find 304 fail-slow drives and they all come from different nodes; Nutanix monitors distributed services on 39 K nodes over 7 months [57], only to collect 232 *independent* fail-slow tickets. Therefore, in our experiments, we inject faults to one node at a time to capture the most common and realistic fault scenarios while maintaining focus on a representative fault model.

• *Duration.* We explore faults of varying durations, from transient issues lasting 30 seconds to long-lasting ones that cover the whole experiment.

2.1 Testing Pipeline

We develop a general and automated slow-fault tolerance testing pipeline that streamlines the aforementioned components: deploying a system, ensuring comparable experiment setup, running cloud benchmarks, injecting diverse slow faults, and collecting and summarizing key metrics. Figure 1 shows the diagram of our testing pipeline. It has four main stages:

① **Initialization.** Upon each test, we first bring up a cluster of the tested system. We wait until the cluster is fully initialized and ready to serve any requests. Test data in a benchmark is also loaded in this stage. We use a small cluster of 3–6 nodes for the testing. We also validate our major findings using a larger cluster size of 10 nodes and 20 nodes in Appendix B.

② **Warm-up.** We let the benchmark run for a warm-up period (30 s) before we start injecting any faults. This is to ensure that the system has reached a steady state so that the statistics we collect are representative. The benchmark client is connected to the system via a dedicated node to avoid interference with the system under test. We also make sure the client connects to all nodes of the system so that the benchmark will not stop in case of a node failure. Moreover, the performance records during the warm-up period will be excluded from our analysis.

③ **Slow-fault injection.** After the warm-up period, the pipeline is ready to inject faults. It enumerates distinct fault attributes and their combinations, injects one setting of slow fault per experiment, and records the setting for later analyses.

④ **Recovery.** The pipeline clears the injected faults, after which the system performance will gradually recover.

2.2 Data Collection

Our pipeline collects performance logs from ongoing benchmarks and system logs throughout the testing process. After each test concludes, the pipeline processes these logs to extract critical system responses, such as tail latency and throughput time series. We then utilize these responses to calculate *performance degradation*, which serves as the primary metric for quantifying the system's nuanced tolerance to slow faults.

Specifically, we first calculate the average throughput before fault injection (denoted as *avg_normal*) and during slow faults (denoted as *avg_slow*). We then derive the performance degradation as $\frac{(avg_normal - avg_slow)}{avg_normal} \times 100\%$, where a higher percentage indicates more severe degradation. Analyzing how degradation and latency vary across different fault scenarios provides insights into the system's tolerance to slow faults.

3 Nuanced Slow-Fault Tolerance

A key challenge distributed systems developers face in addressing slow faults is that they often cannot anticipate the specific slow faults their systems might encounter in the wild, particularly when the systems are deployed on hardware or workloads that differ from their testing environment. In this section, we delve into this nuanced nature by injecting diverse fault scenarios (§3.1), enforcing different hardware resource limits (§3.2), and simulating distinct workload patterns (§3.3).

3.1 How Systems React to Varying Slow Faults?

Figure 2 presents the performance degradation of the studied systems under different slow faults with varying *types*, *severities*, and *locations*. We also validated the observations and findings below using a larger cluster size (Appendix B.1).

Finding 1. *The same slow fault incurs radically different consequences across systems. For example, some systems degrade by more than 25% with a* 1 ms *network delay, while others remain robust until faced with delays 100× greater.*

All the evaluated systems use network and filesystem extensively. They all have extensive fault-tolerance mechanisms. Despite these similarities, their slow-fault tolerance still exhibits **high variability**. For example, CRDB and etcd require a delay as high as 100 ms to yield a notable degradation (>25%), while Cassandra, HBase and Kafka only need 1ms. Moreover, systems like Cassandra, HBase, and Kafka can even experience an unbearable degradation of 70% with only a 10 ms network delay, while the other systems (except for the etcd leader case) never degrade to this extent even with a 1 s delay.

The discrepancy arises from the differences in their fault tolerance mechanisms, at both the design and implementation levels. We further discuss their practices in Section 4.

Implication. Infrastructures (*e.g.*, network) and systems (*e.g.*, a database) are often monitored independently and use one-size-fits-all alerts. This can easily miss seemingly minor slow fault that turns out to be damaging for certain systems. System-specific and cross-layer slowness monitoring is needed.

Finding 2. *The relation between performance degradation and fault severity is non-monotonic. In half of the studied systems, higher packet loss leads to lower degradation.*

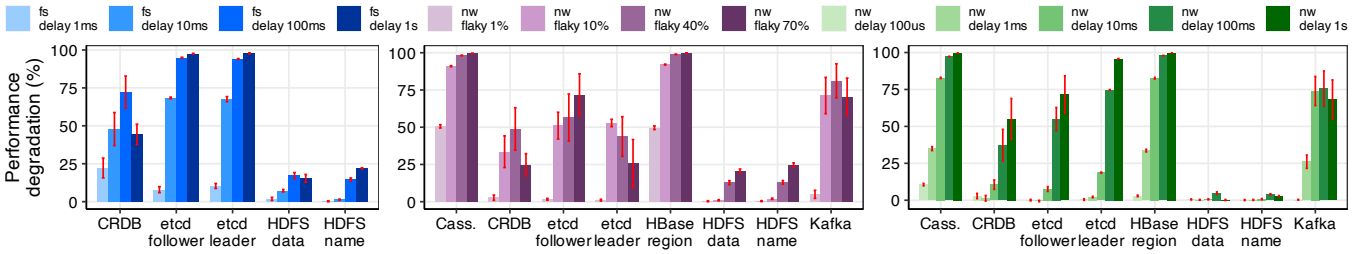


Figure 2: Performance degradation under filesystem delay (left), network packet loss (middle), and network delay (right). The darker the colors, the more severe the faults are. Each experiment was repeated 50 times for generalizability. Error bars refer to 95% confidence intervals.

In CRDB, as Figure 2 shows, the degradation only increases as the filesystem delay increases from 1 ms to 100 ms; however, with a 1 s delay, the resulting degradation (44%) is surprisingly smaller even than the case of 10 ms delay (48%). In Kafka, degradation under a 1 s delay (68%) is also smaller than that under a 10 ms delay (74%). In etcd, the performance becomes better when we inject a higher packet loss (i.e., a *negative* trend) to the leader. The underlying cause lies in whether leader re-election is triggered: a slow leader degrades performance until heartbeat loss triggers re-election (see Section 4.2.1).

Finding 3. *Even within the same system, the performance impact of different slow-fault types varies by up to 10×.*

In CRDB, the average degradation with different filesystem delays is 47%, which is 3.6× that of a delayed network (13%). In etcd, the average degradation under a delayed network is 45%, which is 1.7× than that under packet loss.

Implication. System architects need to consider the distinct impact of different slow-fault types when deciding which infrastructure component to prioritize and optimize. Developers should introduce tailored handling for sensitive fault types.

Finding 4. *Slow faults in followers can impose a more severe (1.5×) performance penalty than similar faults in leaders.*

It is commonly assumed that faults in leader nodes are more impactful than those in followers. However, our study shows a surprising twist in etcd: a slow follower incurs severe (up to 68%) and higher degradation than slow leaders. For example, under flaky networks, the degradation by slow followers (45%) is 1.5× that of slow leaders (31%). This is counter-intuitive as only one slow (even crashed) follower in a 3-node quorum-based system like etcd should at most degrade the overall performance by a third. We investigate and find that the root cause lies in the mismatched awareness of slow faults from the gRPC client and etcd endpoints (elaborated in Section 4.2.2). In contrast, a slow master in HBase does not bottleneck the system, as the YCSB benchmark only interacts with the region servers. In HDFS, both namenodes and datanodes exhibit a similar impact. Since leader roles in Kafka and CRDB are dynamic and change at runtime, the performance impact of a slow leader or follower can be hardly traced or compared.

3.2 Do Resource Limits Amplify Slow Faults?

When a distributed system gets deployed, it can run on different hardware. Does the system's slow-fault tolerance change under varying hardware resource limits? Figure 3 presents the performance degradation of an etcd cluster under four fault scenarios. Each node is subject to the same limits on CPU (1-5 cores) and memory (8-32 GB) based on hardware recommendations provided by the developers [8].

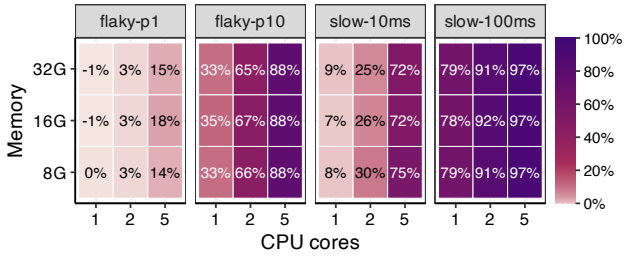


Figure 3: Degradation of etcd under different CPU and memory limits. Each cell corresponds to the average degradation of the same experiment repeated 50 times. Faults are injected to the leader.

Finding 5. *Scaling up resources improves normal performance but paradoxically exacerbates the impact of slow faults.*

For example, for etcd, under a network delay of 10 ms and a per-node memory limit of 32G, degradation increases from 7%, 26%, to 72% with a per-node CPU core limit of 1, 2, 5, respectively. This is because the baseline request latency becomes lower with more CPU cores (from 74 ms to 10 ms), thus the same delay worsens the degradation. This is also the case for packet loss, where latency increase is more pronounced with more CPU cores. However, degradation does not vary much as memory changes. This is because etcd tends to be CPU-bound under heavy load [8]. We validate these observations with a larger cluster size (Appendix B.2).

Implication. More hardware resources do not equate to better fault resilience. Developers need to test their systems' slow tolerance mechanisms under different hardware constraints. The slow-fault alerting and actions need to be resource-aware and recalibrate upon resource changes.

3.3 How Workloads Affect Slow Tolerance?

Deployed distributed systems need to handle diverse and often unpredictable workloads, which can cause developers to overlook slow-fault tolerance in a specific workload. Figure 4 shows the performance impact of network delays for Cassandra and etcd under different workload patterns (read only, balanced mix, and write only). We also validate the following observations with a larger cluster size (Appendix B.3).

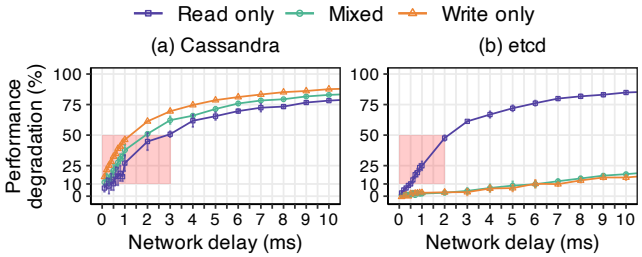


Figure 4: Performance degradation of (a) Cassandra and (b) etcd under distinct workload patterns and network delays. Each set of experiments was repeated 50 times for generalizability. The shaded area highlights the "danger zone" where performance degradation escalates dramatically. Mixed: 50% reads and 50% writes.

Finding 6. *The performance impact of slow faults is highly dependent on the workload, particularly in Raft-based systems (etcd, CRDB, and Kafka), which show up to 7.2× differences.*

Under a 10 ms network delay, the performance degradation of etcd under read-only, balanced mix, and write-only workloads is 85%, 18%, and 15% (up to 5.7× differences). For CRDB, the degradation can vary up to 7.2× differences. For Kafka, the degradation under the asynchronous [1] and synchronous [18] schemes is 20% and 3% (6.7× differences), respectively. For Cassandra and HBase, the degradation under the three workload patterns has lower variances (an average of 1.3× and 1.1× differences, respectively). Also, different systems are sensitive to different workload patterns. In Cassandra, write-only workloads yield the highest degradation, while read-only ones yield the highest degradation in etcd.

Finding 7. *Danger zones commonly exist in the studied systems, where a small increase in slow-fault severity leads to a significant increase in performance degradation.*

Danger zone. The sensitivity to workload patterns and fault severities prompts us to investigate the presence of a "*danger zone*", where degradation escalates dramatically from a moderate 10% to an unbearable 50% as the severity of the fault increases. Identifying the danger zone could potentially help system administrators to proactively monitor and mitigate slow faults, before they escalate and incur catastrophic degradation.

Figure 4 highlights the danger zone. Etcd displays a danger zone from 1 ms to 2 ms network delays only under read-only workloads. In Cassandra, however, danger zones exist under all workloads: 0.2 ms ~ 3 ms, 0.1 ms ~ 2 ms, and 0.1 ms ~ 1 ms for read-only, mixed, and write-only workloads, respectively.

We also apply the danger zone analysis to the other systems. Table 2 summarizes the results. The notion of a danger zone exists in almost all studied systems. However, in systems like etcd and CRDB with write-dominant workloads (≥ 50% write), the danger zones are not obvious. This is because slow writes can be alleviated by their Raft-based protocols. Thus, degradation does not increase drastically as the fault severity increases. Kafka, under batch workloads, does not exhibit obvious danger zones, as batched requests can alleviate the impact of slow faults. HDFS also shows no clear danger zone

	Network delay (ms)			Packet loss (%)		
	R	**M**	**W**	**R**	**M**	**W**
CS	0.2~3	0.1~2	0.1~1	0.2~2	0.2~1	0.2~0.7
HB	0.2~2	0.2~2	0.2~2	0.1~1	0.2~1	0.2~1
CR	4~5	N/A	N/A	2~5	N/A	N/A
EF	1~3	N/A	N/A	0.2~2	N/A	N/A
EL	0.5~2	N/A	N/A	0.2~2	N/A	N/A
	Async	**Sync**	**Batch**	**Async**	**Sync**	**Batch**
KF	0.7~2	0.6~6	N/A	1~5	1~5	N/A
	MRBench	**TeraSort**		**MRBench**	**TeraSort**	
HD	N/A	N/A		N/A	N/A	

Table 2: Danger zone of Cassandra (CS), HBase (HB), CRDB (CR), etcd leader (EL) and follower (EF), Kafka (KF), and HDFS (HD) under different workloads. We denote N/A for cases where the danger zone does not exist. R: read-only; M: mixed; W: write-only.

as it is optimized to support batch processing workloads.

Implication. Early alerting, based on the danger zones associated with sensitive workload patterns, is useful for effectively and proactively managing slow faults. However, the range of these danger zones is quite narrow, typically spanning only a few milliseconds of delay or less than a few percentage points of packet loss. Therefore, the utility of danger zones for early alerting may be limited without fine-grained monitoring.

3.4 Does Tuning Configurations Help?

To ensure consistency and avoid overfitting, we use default configurations of the studied systems. Does tuning a system help its slow-fault tolerance? We use HBase as an example, tuning its configurations for tolerating a 100 ms network delay.

To locate configurations related to slow-fault handling, we search the HBase codebase for keywords such as "slow" and "timeout". In total, there are 120 places matched. Most of them are `TimeoutException` and not configurable. We then manually inspect the rest of them and find 10 configurations (invoked in 43 different files) that are most related to slow faults (*e.g.*, RPC timeout, retry limit). Then, we assign different values for them, making sure that they will always be triggered under a 100 ms network delay. For example, we configure the RPC timeout to be either 10 ms, 50 ms, or 100 ms. Together, we explore 7,776 different combinations of configurations.

We then iterate through each combination and test HBase by injecting a 100 ms network delay with fixed workload settings. This already adds up to *540 machine hours*. As a result, the optimal one yields an average degradation of 27%, which is indeed much lower than that of the default configuration (98%). Finally, we test HBase with this optimal configuration under the same slow fault but different workload settings.

Finding 8. *Fine-tuning system configurations enhances tolerance to specific slow faults, but only within controlled environments. Even worse, it may harm system availability when faced with diverse, real-world workloads, leading to severe degradation or even system failures.*

Figure 5a shows the average degradation under different

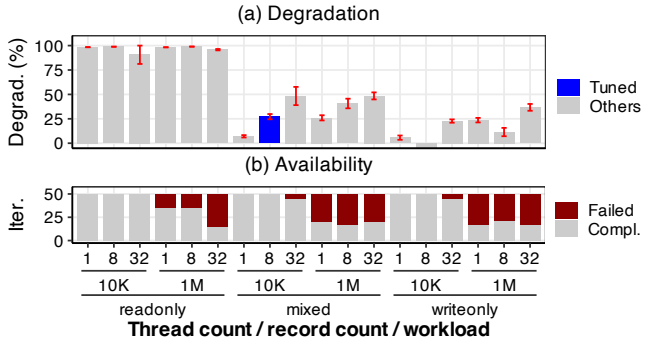


Figure 5: Degradation and availability of HBase with fine-tuned, optimal configurations to tolerate a 100 ms network delay. The x-axis corresponds to combinations of workload settings. The blue bar corresponds to the workload that HBase is fine-tuned on. The red bars indicate the number of failed runs out of 50 iterations. Error bars refer to 95% confidence intervals. *Compl.*: completed runs.

workloads. We iterate through different numbers of client threads, number of records (10K and 1M), and workload patterns. The blue bar corresponds to the workload setting that we previously fine-tuned on. The previously optimal configuration does not always perform well. For example, it causes severe degradation (>90%) under read-only workloads. Also, the degradation jumps to 48% with 32 clients, while a previously suboptimal configuration yields 34% degradation.

When analyzing the results, we find that the HBase cluster would sometimes fail and disconnect from the clients. Thus, we further check the availability of HBase, measured by the number of completed runs. As Figure 5b shows, among 18 workload settings, HBase fails at least once in 11 of them (the red bars), which are all heavier than what we previously fine-tuned on. The worst case is a read-only workload with one million records and 32 client threads: 35 out of 50 runs fail. After investigation, we find that the previously optimal configuration overfits the specific fault and workload, making HBase over-sensitive to and thus fail under heavier workloads.

Implication. Fine-tuning systems for slow faults is not only time-consuming but also does not generalize well across different workloads. Relying on static, fine-tuned configurations makes a system's slow-fault tolerance fragile.

3.5 Can Tail Latency Capture Slow Faults?

Tail latency is a widely used metric for performance monitoring and identifying potential slowdowns. Is tail latency always reliable for capturing slow faults? Figure 6a presents the tail latency of etcd, under a 10 ms network delay in the leader. Each experiment lasts for 150 s with faults extending from 30 s (20% service length) to 150 s (100% service length).

Finding 9. *Tail latency is not always reliable in capturing slow faults. The latency measurements are prone to under-representing the severity of slow faults due to the dilution by abundant measurements from the normal (fast) period.*

The measured tail latency does not exhibit significant variations as fault duration increases. For example, as the duration

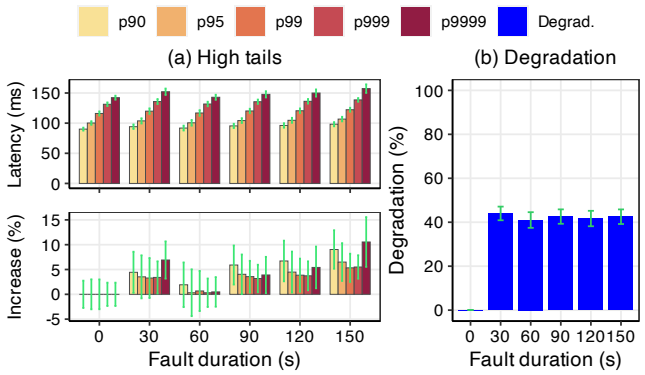


Figure 6: Tail latency and performance degradation of etcd. A 10 ms network delay is injected to the etcd leader with fault durations ranging from 30s to 150s. Each experiment lasts for 150 s.

increases from 30 s to 150 s, the increase in 99.9^{th} latency only ranges from 0.4 ms to 7.2 ms compared to the baseline, which translates to a 0.3%~5.5% relative increase. This is counter-intuitive as we are injecting a 10 ms delay but in return only observe an increase in the 99.9^{th} as minimal as 0.4 ms.

This phenomenon illustrates a limitation of relying on tail latency to detect slow faults. When a system suffers from a slow fault, much fewer requests are processed, hence fewer latency measurements are collected during the slow period. They can be outnumbered by those gathered in the normal period, leading to a misleading perception of low impact for the fault. We observe this pitfall when attempting to repeat the evaluation of MultiPaxos [44, 51] in Copilot [54] (Section 5). Even when the injected fault lasts 90 s in a 100 s experiment time, the 99^{th} latency is still not indicative, because the 90 s slow period only produces 900 latency measurements, while the 10 s normal period produces 300K measurements. Other systems exhibit similar phenomena. For example, the 99^{th} latency is only indicative when the degradation is severe ($\geq$ 83% in Cassandra, $\geq$ 68% in Kafka, $\geq$ 98% in HBase) and persistent ($\geq$ 90 s in Cassandra, $\geq$ 120 s in Kafka and HBase).

This highlights the need to use additional metrics to more comprehensively capture slow faults. For example, throughput degradation, regardless of fault durations, is more indicative of the impact of slow faults on the system (Figure 6b).

3.6 Other Findings

Besides performance degradation, we also study other aspects of slow-fault tolerance. First, we study the time to recover from slow faults (Appendix E). We find that systems recover fast (typically within 2 s) from filesystem and network delays but slower (sometimes more than 20 s) for flaky networks. Moreover, we study the residual impact of slow faults (Appendix F). We find that severe slow faults incur notable residual performance impacts on several systems. Lastly, we examine how slow-fault tolerance evolves across different versions of the same system (Appendix G). We find the improvement in slow-fault tolerance is limited over time.

System metric		Static threshold WT	WC	FT
Cassandra	Execution time of last query [16]	500 ms	-	-
CRDB	Execution time of last disk write [14]	5 s	-	20 s
CRDB	Time to flush pending logs [15]	10 s	-	20 s
etcd	`/livez` to check raft loop execution [7, 13]	5 s	3	-
HBase	Time to flush WAL to disk [9]	100 ms	100	10 s
HDFS	Time to read responses (ACK) from datanodes [17]	30 s	-	-
Kafka	Execution time of last request [10, 11]	30 s	-	2 min

Table 3: Examples of static thresholds used in all studied systems for slow detection. WT: warning threshold; WC: warning count; FT: fatal threshold.

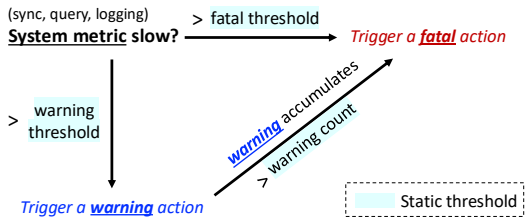


Figure 7: Summary of the common slow detection logic in all studied systems.

4 Current Practices to Handle Slowness

In this section, we study the current development and testing practices of handling slowness in the studied systems.

4.1 Slowness Detection Method

Finding 10. *All studied systems use static timeouts.*

Table 3 summarizes developers' efforts of slowness detection in the codebases of the studied systems. For example, HBase uses a few static thresholds such as `DEFAULT_SLOW_SYNC_TIME_MS=100`ms [5] to identify slow syncs. It also counts the number of slow syncs in a one-minute window: if the total number exceeds a default value of 100 [4], additional actions (log rolls) will be triggered [9]. We will further study the effectiveness of this approach in Section 4.2.3.

Another example lies in the storage engine of CRDB (*i.e.*, pebble), where disk health checks enforce a fixed `diskSlowThreshold` (set as 5 s) to detect disk slowness, and adopts `maxSyncDuration` (set as 20 s) to trigger a service crash [14]. Similar logic is also applied to detect slow logging [15].

Moreover, Cassandra uses `slow_query_log_timeout_in_ms=500`ms for slow queries [16]. HDFS uses `dfsclient SlowLogThresholdMs=30`s to trigger a slow read processor warning [17]. Kafka uses `request.timeout.ms` and `delivery.timeout.ms`, set as 30 s and 2 min respectively, to retry operations [10, 11]. etcd detects stalled writes by checking raft loop execution and exposes a `/livez` endpoint that supervisors can use to restart etcd if needed [7, 13].

Summary. Figure 7 summarizes the common logic of two-staged slow detection in the studied systems. For a monitored system metric, developers first set a warning threshold (WT) to detect slowness. If WT is exceeded, all studied systems will print warning messages. Moreover, if WT is exceeded multiple times (warning count, WC), or the metric exceeds a more severe threshold (fatal threshold, FT), escalated actions like log rolls or service kill will be triggered.

4.2 Slowness Mitigation Action

Besides detection and alert, current systems also trigger different mechanisms in reaction to slow faults. We study the following four major mechanisms: *leader re-election, client-side reconnection, log roll, and escalation to fail-stop*.

4.2.1 Leader Re-election

Systems like CRDB, etcd, and Kafka employ Raft, the leader-based consensus protocol [55]. If a leader encounters failures, either fail-stop or fail-slow, it will be unable to send out heartbeats in time. Subsequently, a healthy follower node will become the new leader and take over the workload.

Finding 11. *Leader re-election is only effective when slow faults are severe enough to impair heartbeats or lease liveness.*

In Section 3.1, we observe that, a slow etcd leader yields less degradation under intensive slow faults (70% packet loss) compared with milder ones (10% packet loss). This is also the case for CRDB. After further investigation, we discover that both systems recover from intensive faults a few seconds after injection. In etcd, the heartbeat timeout is 100 ms. Under milder slow faults, the degraded leader is still responsive to heartbeats. Thus, re-election will not be triggered, leading to continuous slowdowns. Conversely, under more severe slow faults, the leader may fail to respond. Therefore, a new leader will be elected and the system will be no longer bottlenecked by slow faults. The logic is similar in CRDB due to the lease interval (6 s). Appendix C provides more detailed illustrations.

4.2.2 Client-Side Reconnection in gRPC

In replicated systems, it is common for clients (or a load balancer) to maintain a pool of servers. When a server becomes unreachable, the client can reconnect to another server.

Finding 12. *The client side and server side can have different awareness of a slow fault, causing a single slow follower to degrade the cluster performance.*

We observed in Section 3.1 that, a slow etcd follower can also degrade system performance, even if its peer nodes are both healthy and underloaded in the meantime.

Root cause. Clients use a gRPC balancer to connect to and distribute requests across the etcd cluster. Initially, the balancer maintains a simple keepalive ping with one of the endpoints. This connection persists unless the keepalive fails. If the endpoint is a mildly slow follower, the keepalive may not break and thus the slow follower still degrades system performance. Conversely, if severe slow faults break the connection, the balancer will switch to another endpoint, and the performance is restored. Appendix D provides a more detailed illustration.

Essentially, there is a mismatch of information between the server and the client. Although the server side can detect the slow follower, the gRPC balancer lacks this insight. This calls for enhanced protocols to exchange slowness information.

4.2.3 Slowness-triggered Log Roll in HBase

HBase employs Write-Ahead Log (WAL) to record all data modifications. After a write operation is logged, HBase tries to flush WAL to disk (i.e., called *sync*). When WAL reaches a certain size or age, HBase closes the current WAL and starts a new one (i.e., called *log roll*) on a random datanode. Log roll was designed as an efficient mechanism to constrain the WAL file size and ensure fast log replay during recovery.

Interestingly, HBase developers later leverage log roll to also address slow faults. Recall that HBase will randomly select new datanodes after a log roll, thus the log rolling can help mitigate slow faults by avoiding the slow datanode. Starting from Version 2.3 (July 2020), developers add log roll into the slow-fault handling pipeline [9]. It detects slow syncs, and triggers a log roll when syncs exceed 10 s or there are more than 100 slow syncs in one minute.

Finding 13. *The conservative thresholds prevent log roll from being triggered even under severe slow faults.*

Despite the new slow-fault handling mechanism, HBase still suffers from severe performance degradation under slow faults as evidenced in Section 3.1. The empirical thresholds developers choose limit this mechanism's effectiveness.

The current threshold of slow sync, 100 ms, is excessively conservative. As evidenced by our evaluation, a mere 100 ms network delay on the HBase region server already leads to a drastic performance degradation (see Figure 2). By the time a slow sync is detected, the system has already been significantly impacted. Similarly, the condition to initiate a log roll—incurring 100 slow syncs within 60 seconds—is too strict. Under such a condition, HBase is likely to have already collapsed. To verify this, we inject a network delay of 1 s that lasts for 60 s (yielding a degradation of more than 98%). As a result, only 32 slow syncs are reported during the 60 s injection window, failing to trigger a log roll.

4.2.4 Escalation to Fail-Stop

As previously shown, allowing slow nodes to continue serving can worsen the system performance. Recent systems adopt the *fail early principle*, choosing to kill the corresponding process or node in the face of unexpectedly severe slowness [15].

Finding 14. *Recent practices adopt fail early principle to handle the severely slow processes and nodes.*

For example, in CRDB, if a sync operation exceeds `maxSyncDuration=20s`, the process will be killed. In etcd, if we inject persistent yet slight network delays, the system keeps logging warnings and error messages, and kills the slow endpoint in minutes. For HBase, if the slowness-triggered log rolls coincide with a flaky network, there are also chances these log rolls fail and escalate to crash failures of nodes.

While such actions can be beneficial, they need to be applied carefully, because they can impair the system reliability, *e.g.*, leading to cascading failures [38]. Distinguishing a truly severe slow fault from noisy ones is also challenging.

4.3 Testing

In all the studied systems, the tests for their slow-fault handling take the form of unit tests instead of integration tests. In unit tests, developers use the `sleep` function to exceed static detection thresholds, and assert-check the triggered actions. For example, in etcd, developers simulate stalled disk writes by calling `sleep("30s")` [20]; in Cassandra, developers fabricate a slow operation, by calling `Thread.sleep()` [22], then assert-check whether the operation is detected as slow, completed, or aborted [21]. Yet upon reviewing the code and documentation, we find that none of the systems employ an injection tool to test slow faults in an end-to-end manner. This raises concerns about whether preset static thresholds are appropriate and whether the mechanisms can be reliably triggered in a real deployment, particularly as different deploying environments and workloads incur varying system responses.

Finding 15. *Current testing of slow-fault handling mechanisms focuses on the functionality instead of the effectiveness and robustness of their triggering conditions in practice.*

5 State-of-the-Art Solutions

Researchers have recently proposed advanced fail-slow solutions, which have not yet been incorporated into the studied systems. Are these latest techniques effective? In this section, we evaluate three state-of-the-art solutions.

Perseus. Perseus [49] uses machine learning techniques to detect fail-slow storage devices. It relies on device-level telemetry data to train a regression model that infers the normal latency of a device, and detects a slow one if its latency is much higher. The training and detection are both offline.

We re-implement Perseus strictly following its original design. Since Perseus's core idea is to model device-level throughput with latency, we adapt it to model the end-to-end latency and throughput of the studied systems. Figure 8a shows the latency-vs.-throughput (LvT) distribution of HBase under a 50/50 read-write workload. The blue points refer to the LvT distribution of HBase without slow faults. First, we apply the same outlier detection algorithms [23, 59] to filter out biased samples. Then, we train a similar polynomial regression model and calculate the 99.9% prediction upper bound (the blue line): any LvT points above the line are considered slow. Finally, we inject different levels of network delays (the green points) and check if the upper bound identifies any slow faults or not.

This solution fails to detect slow faults in HBase even under our most severe delay of 1 s. The main reason is that the LvT patterns of distributed systems are different from those of storage devices, making the model rather biased. In the original context, slow faults in NVMe SSDs cause latency to jump from tens of microseconds to milliseconds or seconds, while throughput remains moderate. In contrast, in HBase, a mild delay like 10 ms already degrades the throughput by 83%. We evaluate other systems and test different workloads. The results all show that this approach is not robust against

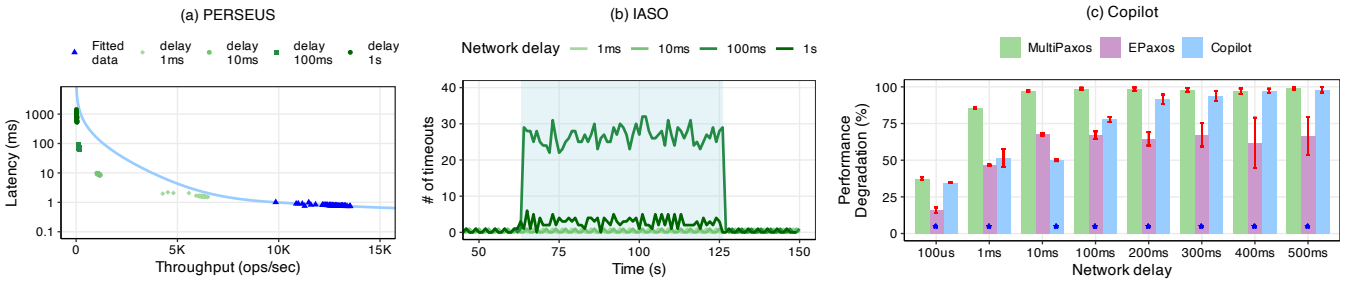


Figure 8: Evaluation of advanced fail-slow solutions. In Copilot (c), we mark the system with the smallest degradation using a blue asterisk.

slow faults in distributed systems.

IASO. IASO [57] is a peer-based, node-level fail-slow detection and mitigation framework for distributed systems. It relies on the deployed systems to emit timeout signals, converts them into a score, and compares the scores across nodes to detect slow ones (*i.e.*, nodes with higher scores than their peers).

However, since these timeouts are mostly static and over-conservative (Section 4.1), the effectiveness of IASO is limited. Figure 8b shows the number of timeouts per second as we inject different network delays to HBase (during the blue-shaded area). Clearly, HBase only emits notable timeouts when the delay reaches 100 ms. This coincides with our prior study in Table 3, where HBase has employed a 100 ms warning threshold for slow syncs. Tuning timeout thresholds may help IASO detect milder delays (e.g., ≤10 ms), yet it is time-consuming to do so and does not adapt to different workloads, as evidenced in Section 3.4.

Copilot. Copilot [54] is the first 1-slowdown-tolerant consensus protocol for quorum-based systems. It introduces two leaders that proactively add redundancy to all stages of the execution flow, so that any one slow replica (be it leader or follower) will not bottleneck the performance.

We evaluate Copilot and its two baselines (MultiPaxos [44, 51] and EPaxos [53]) using its original benchmark. Figure 8c presents their performance degradation under different network delays. Faults are injected to one of the pilots for Copilot, the leader in MultiPaxos, and one arbitrary node in EPaxos. We outline the following observations.

First, even under the mildest delay of 100 us, Copilot still incurs a degradation of 35%. Second, Copilot only yields the smallest degradation under a 10 ms delay. The root cause lies in two sets of static configurations. ❶ Copilot lets the fast pilot take over the work of the other slow pilot if the latter does not respond within 10 ms (*i.e.*, fast-takeover timeout). When the slowdowns are less than 10 ms, Copilot simply waits. This explains why Copilot is not optimal until 10 ms. ❷ When the delays are above 100 ms (*i.e.*, heartbeat sending interval), normal replicas will not receive heartbeats from the slow pilot in time. However, the slow pilot will not be marked as dead and still function until heartbeats are missing for more than 1 s (*i.e.*, heartbeat missing interval). This explains why Copilot still has a higher degradation when delays are above 100 ms.

6 An Adaptive Fail-Slow Detection Library

A consistent theme among our study's findings is that slow faults have highly diverse and dynamic characteristics. Changes (even small ones) in the workload, deploying environment, fault severity, location, *etc.*, can lead to dramatically different consequences. While developers become increasingly aware of the fail-slow problem and start to address it, the current practice of using static thresholds can hardly accommodate the characteristics of slow faults. We advocate for introducing *adaptive* designs [39] into slow-fault handling.

6.1 Limitations of Existing Solutions

As Section 5 shows, recent work does not address the dynamic nature of slow faults. Additionally, solutions such as Perseus and IASO are designed for *operators* to monitor systems as black boxes. They do not offer built-in support to help *developers* implement adaptive slow-fault handling within a system at the code level. An intrinsic adaptive design tackles the problem directly at the scene. It monitors slowness at fine granularity, such as a function call. It also invokes targeted actions to quickly mitigate slowness. However, realizing this design can require significant developer efforts. A library to reduce these efforts is desirable.

6.2 ADR: Adaptive Detection at Runtime

To address the gap, we design ADR, a lightweight adaptive fail-slow detection library for distributed systems. Developers use ADR as a plug-in when adding fail-slow handling code. ADR traces some built-in variables (*e.g.*, `syncOpLatency`), automatically adapts the associated threshold variables to decide slowness, and invokes different levels of defined actions. We aim to achieve the following goals: (1) ease of use; (2) accurate and fast detection; (3) low overhead; (4) robustness.

ADR can be integrated into a system by adding only few changes. Developers have added extensive tracing code for different operations (*e.g.*, write) and compared these tracing variables with static thresholds to detect slow operations. ADR leverages such existing code and replaces the static logic in place to make the detection adaptive.

ADR is directly motivated by the takeaways from our study. It replaces static timeouts (§4.1) with percentile-based detection to adapt to varying slow faults (§3.1), deploying environment (§3.2), workloads and danger zones (§3.3). Since percentile-based monitoring like tail latency can be diluted

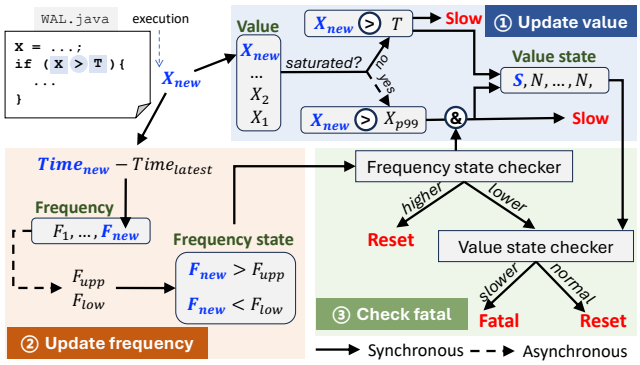


Figure 9: Overview of ADR.

by normal records (§3.5), ADR analyzes how frequently the tracing variables get updated for more reliable cross-validation. As systems employ different mitigation actions (§4.2), ADR outputs a slowness level (slow, fatal) for developers to trigger appropriate actions.

As shown in Figure 9, ADR monitors both the value and the update frequency of some tracing variables defined in the system code using sliding windows. The high-level idea is to adapt the original thresholds used in comparison based on historical values to identify slow data points, and distinguish workload changes from slow faults by checking how often the variable gets updated, *i.e.*, update frequency.

It employs two state checkers to make decisions, which either trigger internal actions (*reset windows*) or return a slowness level (slow or fatal) for the system code to act upon. If it detects a sudden increase in the update frequency, it assumes the system is under a heavier workload and resets all sliding windows to adapt. Conversely, if the update frequency drops, due to either a slow fault or a lighter workload, ADR further checks the values of the traced variable: if they show a sustained slowdown, ADR identifies a slow fault and triggers a fatal action; otherwise, it adapts to a lighter workload in the same manner. We describe each step below.

❶ Update value. When a new value of the tracing variable is updated (X_{new}), ADR stores it into a sliding window of historical values. Initially we use the original threshold T, preserving the default behavior, until the sliding window is saturated. We then calculate the 99^{th} percentile as the adaptive threshold (X_{p99}). If the new response is higher than the threshold (potentially slow), we further query the frequency state checker in Step 3. If the frequency is decreasing, we mark this response as slow and store its state in a sliding window of historical states (to check for continuous slowdown later).

❷ Update frequency. When a new value is updated, ADR also calculates the timestamp difference between the new and last value. We divide 1 s by the difference to get the update frequency of the tracing variable (unit: response/second), stored using a sliding window. Then, ADR calculates the average and standard deviation of the frequency window and derives *average* + *stdDev* (*average* − *stdDev*) as the upper (lower) bound. A frequency higher than the upper bound indicates a potential workload increase (*e.g.*, a burst of requests or heavier workload), while a frequency lower than the lower bound indicates either a decrease in workload intensity or a slow fault. We maintain another sliding window of frequency states to check for continuous frequency changes later.

❸ Check fatal. Finally, ADR checks the frequency and value state windows to distinguish workload changes from continuous slow faults. If the frequency is continuously higher, the system may experience a workload increase and ADR will reset all sliding windows and be ready to adapt new workload. If the frequency is continuously lower, ADR would cross-validate with value states: if the value state indicates a continuous slowdown, ADR would trigger a fatal action if needed; otherwise, the system may experience a workload decrease and all windows will be reset.

Adaptive threshold. ADR calculates the 99^{th} percentile as an adaptive threshold. This does not mean ADR will always treat 1% of total responses as slow. Instead, ADR cross-validates with frequency states to check if the frequency is continuously decreasing. In this case, ADR can accurately detect slow faults while not emitting false alarms due to workload variations.

Dynamic window length. The length of all sliding windows is set to be positively correlated with the frequency (i.e., 10 × current frequency). This is to make sure that the window size is always proportional and adaptive to any frequency change.

State checkers. ADR uses two state checkers to monitor continuous slowdown and workload changes. The value state checker inspects the most recent n value records, and checks if more than half of them are slow; the frequency state checker examines the most recent n frequency states (which represent heavier or lighter workloads) to see if more than half of them are persistent. We choose n to be equal to the current update frequency (*i.e.*, the number of updates in the last second). For example, when the frequency drops from 1 K records/sec to 100, the checkers will only use the most recent 100 data points instead of 1 K. This allows ADR to quickly adapt to workload changes while avoiding the dilution of slow fault records.

ADR also considers the case when slow faults occur during the data collection phase: when the value window is not populated yet to draw a X_{p99} adaptive threshold. In this case, ADR relies solely on the frequency state checker and resets all windows when the update frequency changes notably.

Example. Figure 10 shows the use case of ADR in HBase. Originally, HBase already included tracing code for the WAL sync. The latest sync time is stored in `timeInNanos` (a tracing variable). It further defines two static threshold variables, `slowSyncNs` (line 2) and `rollOnSyncNs` (line3), and uses two statements (lines 5 and 8) to check if the sync time exceeds the thresholds. Different actions would be taken accordingly, emitting a warning (line 7) or rolling WAL (line 10).

With ADR, the original parameters inside the "if" conditions are retained but wrapped up using ADR's APIs. Specifically, we call `isWarn()` in line 6 to compare the latest sync time

```
1  // Default static thresholds
2  long slowSyncNs = TimeUnit.MILLISECONDS.toNanos(100);
3  long rollOnSyncNs = TimeUnit.SECONDS.toNanos(10);
4  void postSync(final long timeInNanos, ...) {
5  -  if (timeInNanos > slowSyncNs) {
6  +  if (ADR.isWarn(timeInNanos, ">", slowSyncNs)) {
7        LOG.info("Slow sync detected.");
8  -     if (timeInNanos > rollOnSyncNs) {
9  +     if (ADR.isFatal(timeInNanos, ">", rollOnSyncNs)) {
10         requestLogRoll(SLOW_SYNC);
11       }
12   }
```

Figure 10: Applying ADR in HBase. Lines with '-' correspond to HBase's original slow-fault detection using static timeouts. Lines with '+' show how ADR is integrated into HBase.

Setup	Warning threshold	Fatal threshold
Static-1	1 ms	10 ms
Static-2	10 ms	100 ms
Static-3	100 ms	1 s

Table 4: Evaluation setups for the three static schemes.

with an adaptive threshold. Then we use `isFatal()` in line 9 to check if there is a continuous slowdown and frequency change (or if the latest value exceeds the default threshold). If so, a log roll would be triggered per developers' design.

Limitations. ADR cannot detect slow faults that appear during system start-up. Operators at the scene are typically able to monitor and handle such cases manually. Additionally, since ADR relies on update frequency to distinguish workload changes from slow faults, it may misclassify slow faults that occur precisely during workload transitions. Finally, ADR assumes developers already know where to check for slow faults. We believe a keyword-based search of potential code locations for instrumentation (§6.3) can be a useful addition.

6.3 Evaluation

Our evaluation aims to answer: (1) can ADR effectively detect and mitigate slow faults of varying severity? (2) is ADR robust to different and dynamic workloads? (3) what is the overhead of ADR at runtime?

Implementation and Baselines. We implement ADR in both Java and Go (~400 lines), and integrate it into HBase and CRDB. We use a keyword-based approach to automatically search for potential variables that can be traced. For example, in HBase, we first identify 57 variables. Among them, 9 exactly match the use case of ADR (comparing with a default static threshold) and are traced; the rest are either not timeouts, part of external APIs (*e.g.*, timeouts in protobuf), or triggering no mitigation action (*e.g.*, only logging the event).

We use the testing pipeline (Section 2.1) to inject slow faults with varying severity and workloads. Each experiment runs for 5 minutes and we inject a fault of 2 minutes starting from 90 seconds. Each set of experiments is repeated 50 times.

We compare ADR with both the original implementation (*i.e.*, vanilla) and three sets of static schemes (see Table 4). Since most studied systems (including HBase and CRDB) employ a typical two-stage timeout mechanism (§4.1), we set two thresholds (a warning and a fatal one) for each setup. The threshold values of the three static schemes are chosen according to the delays injected, so that they are the optimal ones for the corresponding slow faults. For example, Static-1 with a warning threshold of 1 ms would be the optimal one to detect a delay of 1 ms.

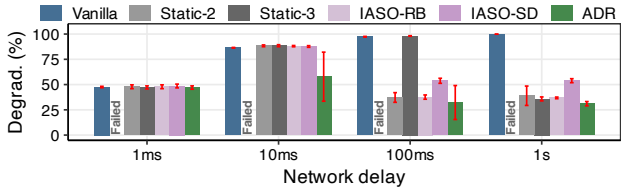


Figure 11: Degradation of HBase using ADR and the baselines under different network delays. Static-1 is too sensitive and fails the cluster. *IASO-RB*: IASO with reboot; *IASO-SD*: IASO with shutdown.

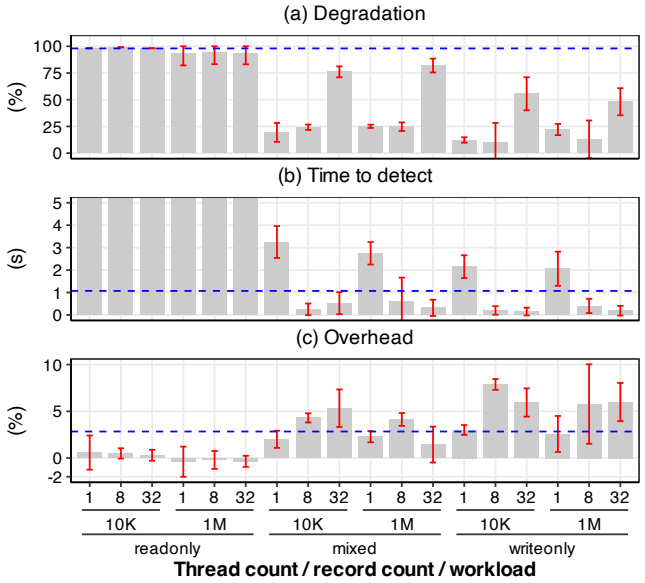


Figure 12: HBase using ADR under different workloads. The blue dashed line in (a), (b), and (c) corresponds to the degradation of vanilla HBase, the average time to detect in mixed and write-only workloads, and the average overhead, respectively.

We also compare ADR with IASO [57] (see Section 5). We have fine-tuned IASO's parameters and chosen the optimal ones under our testing pipeline. IASO employs its own mitigation: upon detecting fail-slow, it will first try to reboot the service instance (*i.e.*, IASO-RB). If the reboot does not work, IASO will then try to shut down the slow node (*i.e.*, IASO-SD). Perseus [49] is not chosen as it is an offline detection solution.

Tolerance of Varying Slow Faults. Figure 11 shows the degradation of HBase under different network delays. ADR helps mitigate slow faults under all injected delays and is always better even than the optimal static settings. For example, under a 100 ms delay, ADR reduces the degradation from 97% (vanilla) to 32% (65% reduction). In comparison, the optimal Static-2 case, IASO-RB, and IASO-SD yield a higher degradation of 37%, 38%, and 54%, respectively.

Similarly in CRDB, the average degradation of the vanilla,

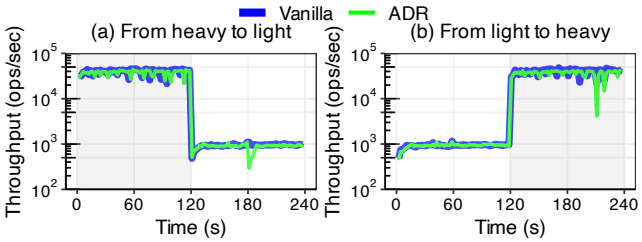


Figure 13: HBase under dynamic workloads. The light (heavy) one refers to mixed (write-only) workloads with 1 (32) YCSB client thread(s) and 10K (1M) records. Workload changes at 120 s.

Static-2, and Static-3 are 52%, 3%, and 44%, respectively, while ADR results in a 7% performance *gain*.

Tolerance Under Varying Workloads. Figure 12 shows the performance of HBase with ADR under diverse workloads. We inject a 100 ms network delay.

ADR decreases the degradation from 98% (vanilla) to as low as 10% (Figure 12a). Note that ADR does not mitigate read-only workloads. This is because the variables that ADR trace are all related to write operations, due to the fact that the system's fault-tolerance mechanisms are not engaged or required as much for reads. Nevertheless, we believe ADR will still work well given any critical read variables are traced. As a result, ADR helps reduce the degradation in mixed and write-only workloads by 16–80% and 43–90% in HBase, and by 20–96% and 5–66% in CRDB, respectively.

Recall the fine-tuned HBase incurs cluster failures under other workloads (Section 3.4). ADR, however, not only yields lower degradation than the optimal static settings, but also maintains system availability with zero failures.

As Figure 12b shows, the average time to detect in mixed and write-only workloads are 1.3 s and 0.9 s, respectively. This shows that ADR detects slow faults in a timely manner.

ADR is robust to dynamic workloads. As shown in Figure 13, system throughput is on par with or without ADR. This indicates that ADR can adapt to workload changes with minimal impact on system performance.

Overhead. Figure 12c shows the overhead of ADR. The average overhead across all workloads is 2.8%. With this low overhead, ADR can be integrated into even more modules.

7 Related Work

Since faults are frequent in distributed systems and fault tolerance is crucial, understanding failures has been a constant theme in distributed system research. Numerous empirical studies [24, 25, 27, 28, 32, 34–36, 41–43, 46, 47, 56, 65] have characterized failures in various settings. They generally have different focuses and are complementary to each other.

One common category is to analyze why certain systems fail and identify the high-level causes. For example, an early prominent study by Oppenheimer *et al.* [56] examined the root causes of failures in commercial Internet services. The findings pointed out that operator errors caused most failures. Yuan *et al.* [65] studied user-reported failures in open-source distributed systems and found that more than half of the catastrophic failures could have been detected by simple testing of error-handling code. Other studies in this category were conducted on Hadoop cluster [58], Microsoft Azure services [46], jobs in Google computer cluster [29], SCOPE jobs in Microsoft Bing [45], cloud servers [62], etc.

More recent studies analyze complex failure modes that are under-explored yet increasingly common in large distributed systems, such as gray failures [42], partial failures [47], metastable failures [41], limplock [32], and silent semantic violations [48]. These studies focus on software failures. Several other studies focus on failures related to hardware, such as network partitions [25], fail-slow hardware [36, 49, 50], and silent data corruption in CPUs [31, 40, 63]. Our study is motivated by recent fail-slow hardware studies. We shift the focus to the impact of fail-slow events on software. Our work revisits the early limplock study [32], which discusses the worst-case scenarios using microbenchmarks. In contrast, we focus on the operational impact of slow faults for modern distributed systems under realistic workloads. We explore a broad spectrum of failure space. For example, we analyze how nuanced slow faults with diverse severity (not just worst case), location, *etc.*, degrade system performance. We measure how varying cloud benchmarks, danger zones, deploying environments, tail latency measurements, and fine-tuning configurations impact slow-fault tolerance. We also evaluate the effectiveness of latest fail-slow detection and mitigation solutions.

A number of solutions have been proposed to tolerate slowness (performance issues) in distributed systems. A typical technique is speculative execution [26, 60, 66], which launches a backup task when a task is slow (straggling). Copilot [54] enhances the consensus protocol by using two distinguished replicas to tolerate slow faults. DepFast [64] provides programming interfaces for developers to implement slow-tolerant protocols. Our study aims to provide insights for distributed systems designers to improve slow-fault tolerance solutions.

8 Conclusion

Recent studies have increased the community's awareness of the fail-slow problem. Nevertheless, fail-slow tolerance remains a challenging topic to approach. We present a comprehensive study that analyzes the nuances of fail-slow tolerance in current systems. Our study reveals that existing efforts do not accommodate the highly dynamic and sensitive characteristics of slow faults. We showcase how adopting a simple adaptive design can significantly increase the effectiveness of a system's existing slow-fault handling mechanisms.

Acknowledgments

We thank our shepherd Kang Chen, and the anonymous reviewers for their insightful comments. We thank CloudLab [33] for providing the resources for us to run experiments. This work was supported in part by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937. Ruiming Lu's travel expenses to attend the conference are funded by NSFC 623B2072.

References

[1] Baseline scheme of the openmessaging benchmark framework. https://github.com/openmessaging/benchmark/blob/master/driver-kafka/kafka-throughput.yaml.

[2] Blockade: testing network failures and partitions in distributed applications. https://github.com/worstcase/blockade.

[3] Charybdefs: Scylladb fault injection filesystem. https://github.com/scylladb/charybdefs.

[4] Default theshold of slow sync count in hbase. https://github.com/apache/hbase/blob/rel/2.5.6/hbase-server/src/main/java/org/apache/hadoop/hbase/regionserver/wal/AbstractFSWAL.java#L136.

[5] Default value of slow sync threshold in hbase. https://github.com/apache/hbase/blob/rel/2.5.6/hbase-server/src/main/java/org/apache/hadoop/hbase/regionserver/wal/AbstractFSWAL.java#L131.

[6] etcd v3 benchmark tool. https://github.com/etcd-io/etcd/tree/main/tools/benchmark.

[7] Github issue of stalled leader detection in etcd. https://github.com/etcd-io/etcd/issues/15247.

[8] Hardware recommendations for etcd v3.5. https://etcd.io/docs/v3.5/op-guide/hardware/.

[9] Hbase-22301: Consider rolling the wal if the hdfs write pipeline is slow. https://issues.apache.org/jira/browse/HBASE-22301.

[10] Official document of kafka on delivery timeout. https://kafka.apache.org/documentation/#producerconfigs_delivery.timeout.ms.

[11] Official document of kafka on request timeout. https://kafka.apache.org/documentation/#brokerconfigs_request.timeout.ms.

[12] Openmessaging benchmark framework. https://github.com/openmessaging/benchmark.

[13] Public development document of etcd livez and readyz probe. https://docs.google.com/document/d/1PaUAp76j1X92h3jZF47m32oVlR8Y-p-arB5XOB7Nb6U.

[14] Source code of slow disk detection in cockroachdb. https://github.com/cockroachdb/cockroach/blob/v23.1.11/pkg/storage/pebble.go#L1246-L1275.

[15] Source code of slow logging detection in cockroachdb. https://github.com/cockroachdb/cockroach/blob/v23.1.11/pkg/util/log/file.go#L242-L284.

[16] Source code of slow query detection in cassandra. https://issues.apache.org/jira/browse/CASSANDRA-12403.

[17] Source code of slow read processor detection in hdfs. https://github.com/apache/hadoop/blob/release-3.3.6-RC0/hadoop-hdfs-project/hadoop-hdfs-client/src/main/java/org/apache/hadoop/hdfs/DataStreamer.java#L1138-L1149.

[18] Synchronous scheme of the openmessaging benchmark framework. https://github.com/openmessaging/benchmark/blob/master/driver-kafka/kafka-sync.yaml.

[19] Sysbench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[20] Test case of handling stalled writes in etcd. https://github.com/etcd-io/etcd/blob/main/tests/e2e/v3_lease_no_proxy_test.go#L116-L117. Accessed Apr 18, 2024 at commit 9ac4f33.

[21] Unit test of slow detection in cassandra using assert. https://github.com/apache/cassandra/blob/trunk/test/unit/org/apache/cassandra/db/monitoring/MonitoringTaskTest.java#L191-L201. Accessed Apr 18, 2024 at commit 209c35a.

[22] Unit test of slow detection in cassandra using thread.sleep. https://github.com/apache/cassandra/blob/trunk/test/unit/org/apache/cassandra/db/monitoring/MonitoringTaskTest.java#L118. Accessed Apr 18, 2024 at commit 209c35a.

[23] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Computational Statistics*, 2010.

[24] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 351–368. USENIX Association, November 2020.

[25] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 51–68, USA, 2018. USENIX Association.

[26] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, NSDI '13, pages 185–198, Lombard, IL, April 2013. USENIX Association.

[27] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01, pages 33–, Washington, DC, USA, 2001. IEEE Computer Society.

[28] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 221–227, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure analysis of jobs in compute clouds: A Google cluster case study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE '14, pages 167–177, 2014.

[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale, 2021.

[32] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[33] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[34] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, page 61–74, USA, 2010. USENIX Association.

[35] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SOCC '16, pages 1–16, October 2016.

[36] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, February 2018. USENIX Association.

[37] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 139–152, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure recovery: when the cure is worse than the disease. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS '13, page 8, USA, 2013. USENIX Association.

[39] T. Herman and M. Gouda. Adaptive programming. *IEEE Transactions on Software Engineering*, 17(09):911–921, sep 1991.

[40] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs,

Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[42] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, British Columbia, Canada, May 2017. ACM.

[43] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics. In *6th USENIX Conference on File and Storage Technologies*, FAST '08, San Jose, CA, February 2008. USENIX Association.

[44] Leslie Lamport. The part-time parliament. In *ACM Transactions on Computer Systems*, volume 16 of *TOCS*, page 133–169, 1998.

[45] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 963–972. IEEE Press, 2013.

[46] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 155–162, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[48] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and checking silent semantic violations in large distributed systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107, Carlsbad, CA, USA, July 2022. USENIX Association.

[49] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. Perseus: A Fail-Slow detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies*, FAST '23, pages 49–64, Santa Clara, CA, February 2023. USENIX Association.

[50] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. NVMe SSD failures in the field: the Fail-Stop and the Fail-Slow. In *2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 1005–1020, Carlsbad, CA, July 2022. USENIX Association.

[51] David Mazières. Paxos made practical. https://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, 2007.

[52] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. Fast in-network gray failure detection for isps. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 677–692, New York, NY, USA, 2022. Association for Computing Machinery.

[53] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery.

[54] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 583–598. USENIX Association, November 2020.

[55] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[56] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, USITS '03, March 2003.

[57] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A Fail-Slow detection and mitigation framework for distributed storage services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 47–62, Renton, WA, July 2019. USENIX Association.

[58] Ariel Rabkin and Randy Katz. How Hadoop clusters break. *IEEE Softw.*, 30(4):88–94, jul 2013.

[59] Erich Schubert, Jörg Sander, Martin Ester, Hans Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: Why and how you should (still) use dbscan. *ACM Transactions on Database Systems*, 2017.

[60] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Maheswara Rao G. Uma, and Haryadi S. Gunawi. PBSE: A robust path-based speculative execution for degraded-network tail tolerance in data-parallel frameworks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 295–308, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, Boston, MA, February 2019. USENIX Association.

[62] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 193–204, New York, NY, USA, 2010. Association for Computing Machinery.

[63] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 216–230, New York, NY, USA, 2023. Association for Computing Machinery.

[64] Andrew Yoo, Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. Fail-slow fault tolerance needs programming support. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 228–235, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, page 249–265, USA, 2014. USENIX Association.

[66] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 29–42, USA, 2008. USENIX Association.

A Fault Injection Tools

We use blockade [2] to inject network slow faults and charybdefs [3] to inject filesystem ones.

Blockade is a network testing tool based on Docker utilities, allowing users to inject network failures in distributed applications. In essence, it employs Linux traffic control (`tc`) to inject network delays, packet loss, and partitions. In conclusion, our testing pipeline leverages blockade to inject delays and packet loss to running distributed nodes.

Charybdefs is a FUSE-based fault injection filesystem with a Thrift RPC interface for instrumentation. It simulates various filesystem faults, such as delays, read/write errors, and disk full. We use charybdefs to inject filesystem delays to the running distributed systems.

B Larger Cluster Size

In Section 3, we discuss the nuanced fault tolerance of distributed systems against diverse slow faults (§3.1), different resource limits (§3.2), and varied workloads (§3.3). However, the discussion is based on a small cluster size (3–6 nodes). In this section, we evaluate the impact of cluster size and check whether our findings and observations still persist. We validate the findings (Findings 1-7) presented in Section 3 using CRDB and etcd under a 10-node and 20-node setup.

B.1 How Systems React to Varying Slow Faults

Figures 14 and 15 present the performance degradation of etcd under a 10-node and 20-node setup, respectively. Although the absolute numbers of degradation do not exactly match with those in Figure 2, they still vary across systems as in Finding 1. For example, under a mixed workload with delayed network in the leader node, the degradation of 3-node etcd cluster is 0.5%, 2%, 19%, 75%, and 96% for 100 us, 1 ms, 10 ms, 100 ms, and 1 s delays, respectively. In comparison, the corresponding degradation of a 10-node etcd cluster is 0.5%, 1%, 9%, 51%, and 88% for the same delays, while that of a 20-node etcd cluster is 0.1%, 0.3%, 7%, 46%, and 74%.

Regarding different fault types, the average degradation of a 10-node etcd cluster (mixed workload, injected on the follower) under network packet loss and network delay is 10% and 37%, respectively. This indicates that etcd is much more sensitive (3.7× differences) to packet loss than network delay, which is consistent with Finding 3.

Moreover, slow faults still impact systems differently conditioned on the fault locations. In a 10-node etcd cluster, when we inject a 100 ms network delay to either the leader or the follower, the degradation is 8% and 51%, respectively. This translates to a 6.4× difference. As a result, Finding 4 also holds in a larger cluster size.

Previously, we summarized in Finding 2 that the performance degradation does not always increase as slow-fault severity increases. We find this finding also holds for CRDB with a larger cluster size. Figure 16 presents the degradation of 10-node and 20-node CRDB under different filesystem delays. As we can see, in a 10-node CRDB cluster under the read-only setup, the degradation under a 100 ms delay (82%) is 2.4× that of a 1 s delay (34%). We also observe a similar

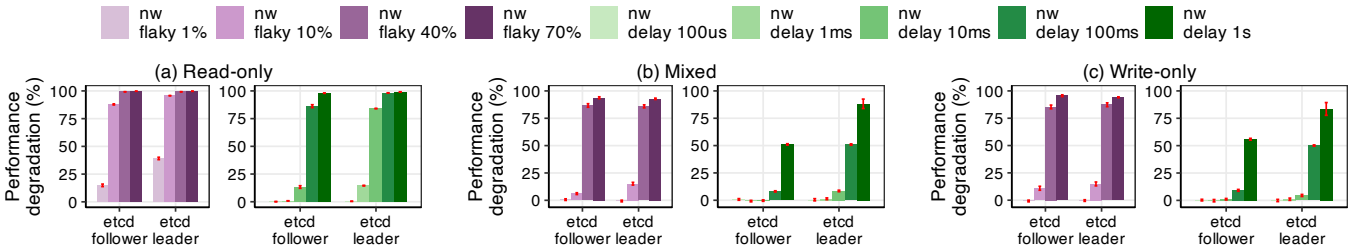


Figure 14: Performance degradation of 10-node etcd under different network packet loss and network delay. The darker the colors, the more severe the faults are. Each experiment was repeated 50 times for generalizability. Error bars refer to 95% confidence intervals.

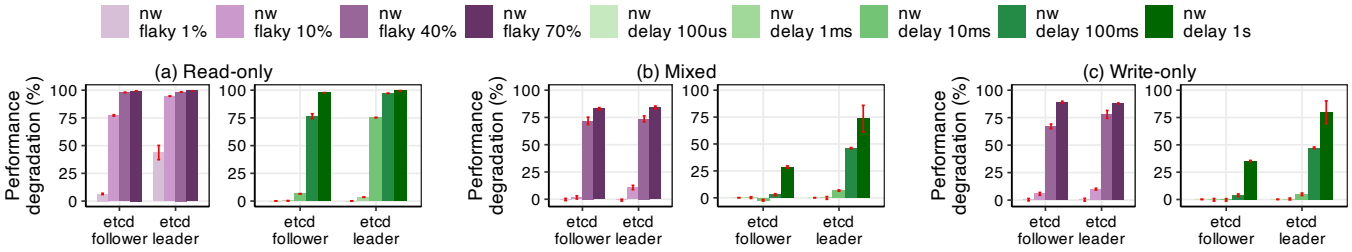


Figure 15: Performance degradation of 20-node etcd under different network packet loss and network delay. The darker the colors, the more severe the faults are. Each experiment was repeated 50 times for generalizability. Error bars refer to 95% confidence intervals.

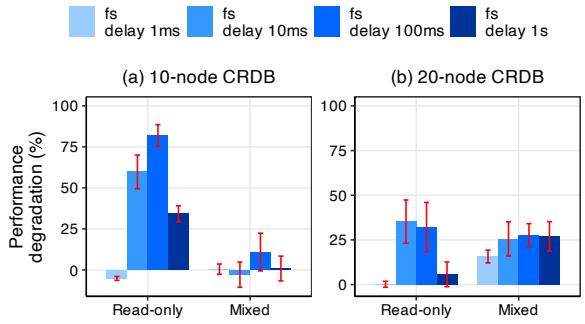


Figure 16: Performance degradation of 10-node and 20-node CRDB under different filesystem delays. The darker the colors, the more severe the faults are. Each experiment was repeated 50 times for generalizability. Error bars refer to 95% confidence intervals.

trend when the workload changes to mixed or the cluster size increases to 20 nodes.

To sum up, as the cluster size increases to 10 or even 20, our general observations and findings (Findings 1-4) regarding how systems are impacted by different fault types, fault severities, and fault locations are still consistent with those in Section 3.1.

B.2 Do Resource Limits Amplify Slow Faults?

Figure 17 presents the performance degradation when a 1 s network delay is injected to the etcd leader node. We set different resource limits on CPU cores (ranging from 1 to 5 cores) and memory sizes (ranging from 8G to 32G). As we can see, either under a 10-node or 20-node setup, the degradation of etcd is still sensitive to the amount of resources. For example, under the 20-node setup with a per-node memory

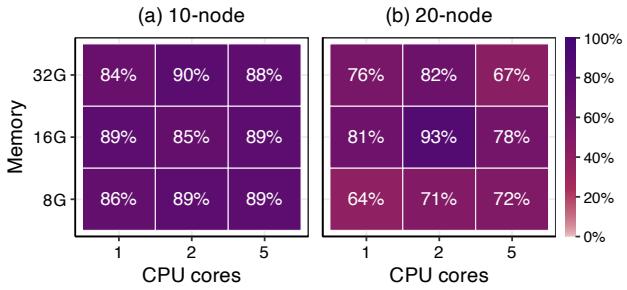


Figure 17: Performance degradation of etcd under different CPU and memory limits. Values in each cell correspond to the average degradation of the same set of experiments repeated 50 times. A 1 s network delay is injected to the etcd leader node. The darker the colors, the more severe the degradation is.

of 32G, degradation ranges from 76%, 82%, to 67%, with a per-node CPU core limit of 1, 2, 5, respectively. This is consistent with our Finding 5 in Section 3.2.

B.3 How Workloads Affect Slow Tolerance

Figure 18 presents the performance degradation of etcd conditioned on different workload patterns (read-only, balanced mix, and write-only) and injected network delays (from 0 to 1 s, injected on the leader). Obviously, system degradation is sensitive to workload patterns. For example, for a 10-node etcd cluster under a 10 ms network delay (see Figure 18b), the performance degradation of etcd under read-only, balanced mix, and write-only workloads is 84%, 9%, and 4% (up to 21× differences). We also obtain similar results in a 20-node etcd cluster or when slow faults are injected to the follower node (see Figure 20). This is consistent with Finding 6 in Section 3.3.

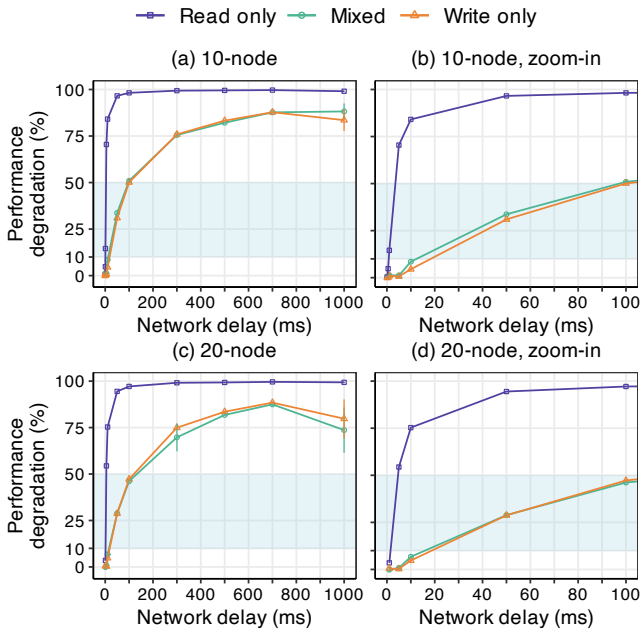


Figure 18: Performance degradation of etcd under different network delays (injected on the leader). Each set of experiment was repeated 50 times for generalizability. The shaded area highlights the "danger zone" where performance degradation escalates dramatically. Mixed: 50% reads and 50% writes.

Moreover, danger zone (between 10% and 50% degradation) still exists. For example, the danger zone for 10-node etcd under a read-only workload (injected on the follower) is from 10 ms to 50 ms (see Figure 20d). We also observe notable danger zones in the 20-node setup or when slow faults are injected to the leader node. This is consistent with our Finding 7 in Section 3.3.

C Leader Re-Election

Figure 19 shows the logic diagram of why severe slow faults on the leader can lead to better performance in etcd.

D Client-Side Reconnection in gRPC

Figure 21 shows the logic diagram of why a single slow follower can possibly degrade the performance of the whole

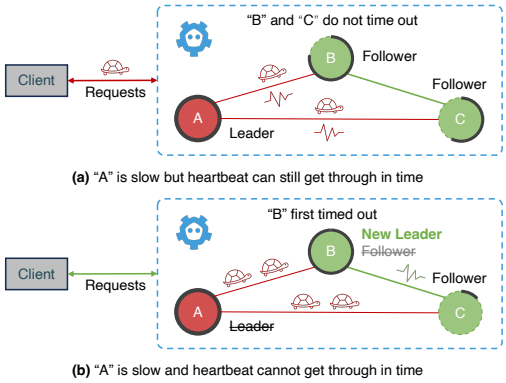


Figure 19: Severe slow faults trigger leader re-election.

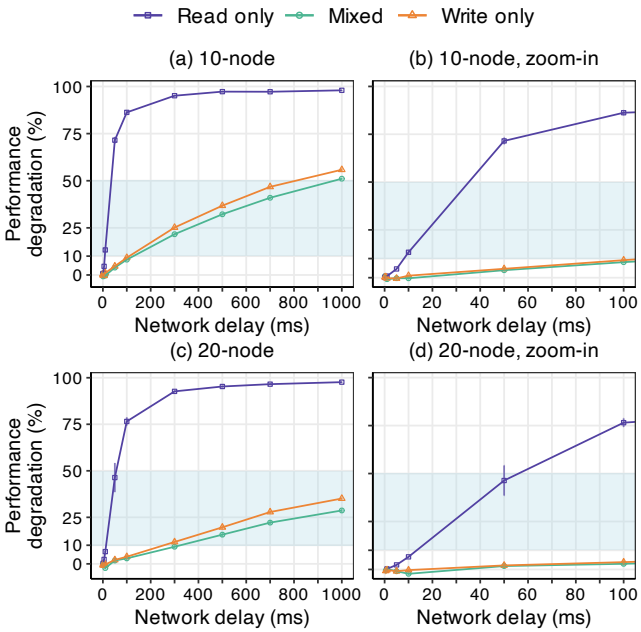


Figure 20: Performance degradation of etcd under different network delays (injected on the follower). Each set of experiment was repeated 50 times for generalizability. The shaded area highlights the "danger zone" where performance degradation escalates dramatically. Mixed: 50% reads and 50% writes.

cluster in etcd.

E Time to Recover

Metrics and notations. To determine when the system has recovered from slow faults, we calculate *recovery time* as the time it takes for the system to reach back a throughput that is higher than a *threshold*. The *threshold* is calculated as *avg_normal* – *sd_normal*, where *sd_normal* is the standard deviation of the series of throughput records before fault injection. Figure 22 presents the distribution of recovery time among all studied systems.

Recovery time in delayed filesystem and network. Systems can recover quickly from delayed filesystem and network. As shown in Figure 22, the average recovery time for all studied systems under a delayed network and filesystem is 0.8s and 1.0s respectively. We do not observe a notable jump

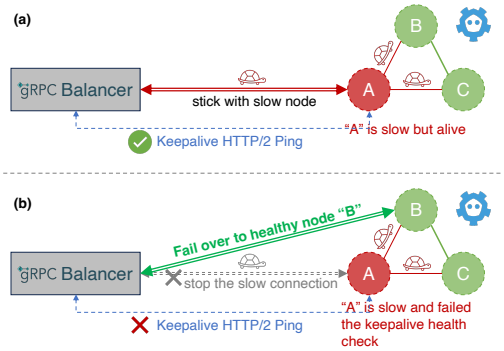


Figure 21: A slow follower can degrade cluster performance.

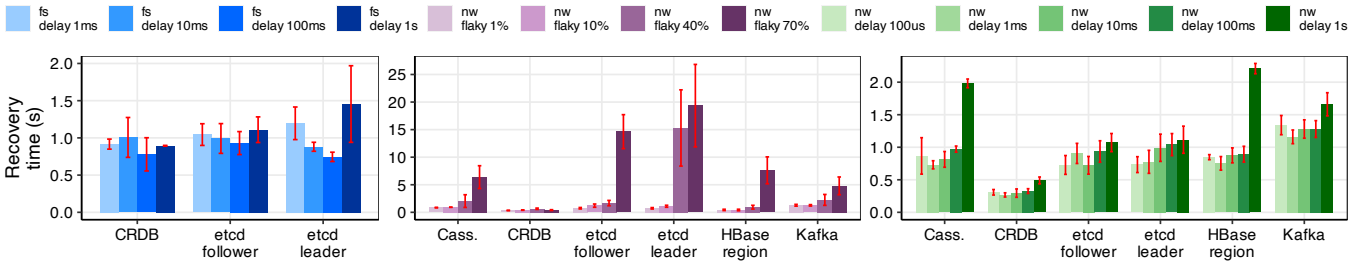


Figure 22: Time for performance to recover after filesystem delay (left), network packet loss (middle), and network delay (right). The darker the colors, the more severe the faults are. Each experiment was repeated 50 times for generalizability. Error bars refer to 95% confidence intervals.

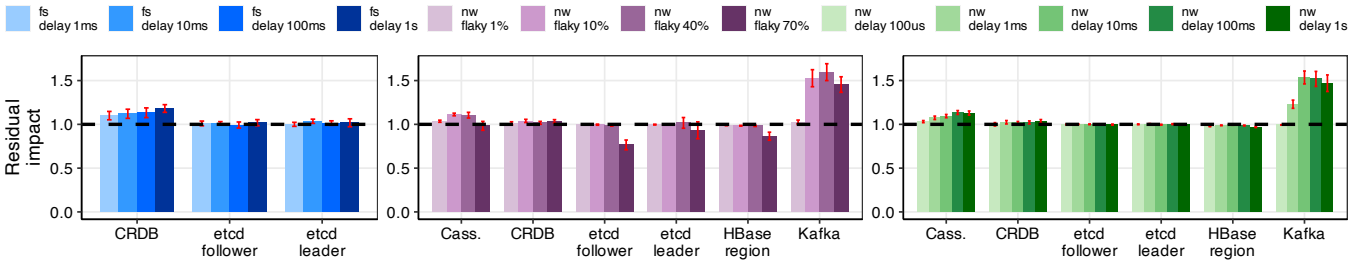


Figure 23: Residual performance after filesystem delay (left), network package loss (middle), and network delay (right). The darker the colors, the more severe the faults are. Each experiment was repeated 50 times for generalizability. Error bars refer to 95% confidence intervals.

in the recovery time as we iterate through higher delays. Kafka takes a slightly longer recovery time (an average of 1.3s). This is because of the distributed nature of Kafka (i.e., data streaming) which relies more heavily on network communication. Nevertheless, it is still acceptable as the recovery time is still within seconds. Also, CRDB generally needs more time to recover from filesystem delays (an average of 0.9s) than network delays (0.3s).

Recovery time in flaky network. However, in the face of network packet loss, the recovery time could be much longer. In general, the recovery time of all systems increases as their network becomes more flaky: the average time to recover under a 10%, 40%, and 70% packet loss is 0.8s, 3.6s, and 7.1s respectively.

We notice that the recovery time for a 70% packet loss is notably higher than in the other two cases. This is because the injected node sometimes fails to establish a connection with the cluster due to high packet loss and even experiences some downtimes. In this case, after faults are cleared, systems will schedule a considerable amount of background tasks to help the injected node catch up with the cluster (e.g., hinted handoff in Cassandra). In this case, system performance cannot recover until background tasks are mostly finished and no longer serve as bottlenecks. However, etcd may intentionally shut down faulty nodes, thus no longer recovered.

Relationship between degradation and recovery time. A higher performance degradation does not necessarily yield a much longer recovery time. For example, CRDB degrades by 22% under a 1ms filesystem delay and then recovers in 0.92s, while it degrades much more by 72% under a 1s filesystem delay but then recovers faster in 0.78s. Similar observations can be obtained in most fault setups (except for those with high packet loss of 40% and 70%). This phenomenon can be attributed to the fact that most slow faults we introduced, while significantly impeding a distributed system, do not crash nodes. Thus, systems can quickly recover to normal once the faults are cleared.

F Residual Impact

Metrics and notations. We calculate the average throughput after slow faults are cleared (denoted as *avg_residual*), and compare it with the average throughput before fault injection (denoted as *avg_normal*). We quantify the residual impact as $\frac{avg_residual}{avg_normal}$ where a value higher than 1 indicates that the system is fully recovered and even outperforms the normal state. On the other hand, a value smaller than 1 indicates that the system's overall performance is still affected by the residual impacts of slow faults. Figure 23 presents the distribution of residual impacts among all studied systems.

Performance drop. Severe slow faults could incur notable residual impacts on system performance. For example, soon after the 70% packet loss is cleared, etcd follower still suffers from severe slowdowns for a while, while its performance gradually increases over time. As a result, it can only achieve an average performance that is only 77% of the normal state. Similar observations can also be obtained in HBase.

Performance gain. We observed that some systems issue bursty requests after slow faults finish. For example, the performance gains in CRDB (after filesystem delays) and Cassandra (after network delays) are all above 1, with an average of 1.13×

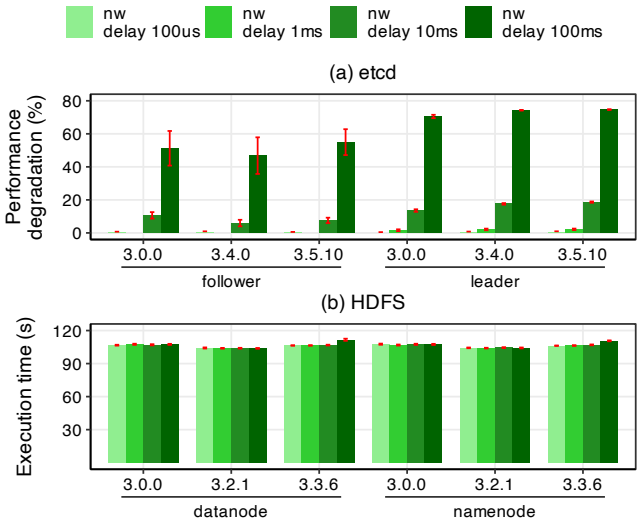


Figure 24: Cross-version analysis of HDFS and etcd under network slow faults. The x-axis shows different fault injection locations and system versions. The y-axis shows the performance degradation in etcd or the execution time of the TeraSort benchmark in HDFS.

and 1.08× respectively. Moreover, the more severe prior faults are, the more performance gain we can observe. This is due to systems' internal recovery mechanisms, such as backlog in CRDB and hinted handoff in Cassandra.

Kafka demonstrates the most performance gains among all systems, with an average of 1.4× after network faults are cleared. Also, we notice that its cumulated performance gain resembles its cumulated performance degradation during slow faults (see Figure 2). This is due to the delayed flush of messages. Producers in Kafka by default will buffer unsent messages (due to a slow network in our case) in the memory and then flush them all once the network faults are cleared. In this case, the performance gain we observe is similar to the performance degradation caused by slow faults. To validate this, we configured Kafka to force flushing every message, which eliminated such notable performance gains.

G Cross-Version Analysis

Distributed systems are constantly evolving, integrating bug fixes and new features that also influence their fault tolerance capabilities (e.g., HBase's log roll mechanism discussed in Section 4.2.3). In this section, we evaluate and compare the nuances of slow-fault tolerance across different versions (see Table 5) of two representative systems, HDFS and etcd. HDFS is a relatively mature system while etcd represents newer generations of distributed systems. Our goal is to understand the evolving trajectory of these systems in terms of their tolerance to slow faults.

System	Version	Year
HDFS	3.3.6	2023
	3.2.1	2019
	3.0.0	2017
etcd	3.5.10	2023
	3.4.0	2019
	3.0.0	2016

Table 5: Versions of HDFS and etcd used in the cross-version analysis.

Figure 24 presents the evaluation of etcd and HDFS under network slow faults. In both HDFS and etcd, the performance degradation does not vary much across different versions. As we increase the degrees of severity, the performance degradation in etcd leader or follower also increases accordingly. We also check other fault types (e.g., filesystem delays), showing similar results. In conclusion, there is no significant improvement in the slow-fault tolerance of these systems over time. Their version updates highlight a lack of focus on this field. Yet, the evident performance and awareness discrepancies across these systems show the feasibility and improvement we can obtain through moderate efforts.

Pyrrha: Congestion-Root-Based Flow Control to Eliminate Head-of-Line Blocking in Datacenter

Kexin Liu⋆, Zhaochen Zhang⋆*, Chang Liu⋆, Yizhi Wang⋆, Vamsi Addanki△, Stefan Schmid△, Qingyue Wang⋆, Wei Chen⋆, Xiaoliang Wang⋆, Jiaqi Zheng⋆, Wenhao Sun+, Tao Wu+, Ke Meng+, Fei Chen+, Weiguang Wang+, Bingyang Liu+, Wanchun Dou⋆, Guihai Chen⋆, Chen Tian⋆*
⋆Nanjing University, △TU Berlin,+Huawei, China

Abstract

In modern datacenters, the effectiveness of end-to-end congestion control (CC) is quickly diminishing with the rapid bandwidth evolution. Per-hop flow control (FC) can react to congestion more promptly. However, a coarse-grained FC can result in Head-Of-Line (HOL) blocking. A fine-grained, per-flow FC can eliminate HOL blocking caused by flow control, however, it does not scale well.

This paper presents Pyrrha, a scalable flow control approach that provably eliminates HOL blocking while using a minimum number of queues. In Pyrrha, flow control first takes effect on the root of the congestion, i.e., the port where congestion occurs. And then flows are controlled according to their contributed congestion roots. A prototype of Pyrrha is implemented on Tofino2 switches. Compared with state-of-the-art approaches, the average FCT of uncongested flows is reduced by 42%-98%, and 99th-tail latency can be 1.6×-215× lower, without compromising the performance of congested flows.

1 Introduction

Given the increasingly stringent performance requirements on datacenter networks, avoiding congestion and the resulting delays has become critical for many applications. Indeed, measurement studies show that congestion events are frequent in today's datacenters, *e.g.*, bursty key-value stores [18, 30], web search services with massive queries [11], and data-parallel/machine-learning systems with partition/aggregation traffic patterns [29, 90, 92, 13, 26, 91, 4, 70, 33]. Generally, congestion occurs at an output port when the arrival rate of traffic exceeds its link bandwidth. Queues build up at congested ports. With an inflated buffer, flows could endure a long queuing delay or even face packet loss, hence flows' completion times (FCT) can be prolonged [11, 56].

State-of-the-art approaches to handle congestion is end-to-end congestion control (CC) [11, 95, 65, 56, 19, 55, 61, 15, 35, 6, 14, 86, 58]. Congestion can be detected by senders when congestion signals are sent back or feedback delay is observed. Usually, it costs senders at least one Round-Trip Time (RTT) to be aware of the congestion, and then senders may take several RTTs to converge to an appropriate transmission rate. In modern datacenters, the effectiveness of end-to-end CC is quickly diminishing with the rapid bandwidth evolution [36, 43, 74, 22](§ 2.1).

With the increasing link speed and performance requirements of datacenter networks, an intriguing and emerging alternative is per-hop flow control (FC), which can react to congestion much more promptly. It suppresses the transmission of the upstream entity before overwhelming the downstream queue, which avoids a large buffer occupancy. Generally, traffic in the same queue is controlled as a whole. Once the queue length exceeds a given threshold, a pause frame can be sent to pause the upstream entity [16, 17], avoiding further buffer build-up where congestion occurs. However, a coarse-grained flow control might spread the congestion to the whole network, inducing Head-Of-Line (HOL) blocking, and hurting the performance of victim flows [40, 54, 62, 57]. Here HOL blocking refers to flows being paused innocently (§ 2.2). A naïve approach to eliminate HOL blocking could be to isolate each flow into different queues and control each of them separately. However, such a per-flow granularity flow control is not scalable since the hardware resources of switches are limited. State-of-the-art flow control approaches hence aim at reducing the number of queues required by compromising the granularity of isolation [36]. Thus, the HOL blocking can not be eliminated entirely (§ 2.3).

This paper explores how to eliminate HOL blocking in a scalable manner, *i.e.*, minimizing the required number of queues. We observe that when congestion occurs, flow control first takes effect on the root of the congestion, *i.e.*, the port where congestion occurs. Then several congestion hotspots (*i.e.*, output port with buffer build-up) can appear along the back pressure path of flow control. These involved hotspots form a congestion tree, where the root of the

*First author and second author contribute equally to this work.

tree is the root of the congestion. Simply controlling the transmission of flows based on hotspots could induce HOL blocking. Instead, if we separately control the transmission of flows according to the congestion root they participate in, flows will not be paused innocently. As the number of concurrent congestion roots observed on each port is moderate, only a reasonably small number of queues on each port are required for traffic isolation in each switch [20].

Based on these insights, this paper presents Pyrrha, a congestion-root-based per-hop flow control protocol (§ 3). A framework is constructed for analyzing egress-based flow control algorithms. We analytically prove that Pyrrha is a HOL-blocking-free per-hop flow control protocol with the minimum queue requirements.

In Pyrrha, each switch maintains a snapshot of the congestion status of the downstream network. Flows that would pass through the same congestion root in their downstream paths could be pushed into a dedicated congestion queue locally in each upstream switch. Then, flows passing through different congestion roots can be controlled separately as soon as possible. Pyrrha solves a set of challenges.

- ***How to identify a hotspot's role?*** When congestion occurs, multiple congestion roots may be claimed in succession. This could occur when an upstream congestion hotspot claims itself root while its downstream hotspot disagrees, and vice versa. With a local view, it is hard to tell which root is the exact root in the upcoming congestion. Intuitively, given the behavior of flow control, congestion roots always locate on the most downstream ports that flows pass through. Inspired by the root-selection procedure of classic spanning tree algorithms [7, 32], Pyrrha employs a distributed self-stabilizing *merge* mechanism. A port can claim itself a *self-nominated* congestion root independently when it detects its queue buildup for the first time. It then abdicates its claim in favor of a *self-nominated* downstream root if (part of) its flows pass through the downstream claimer. Quickly, participating hotspots can converge to a congestion tree. Naturally, the congestion root is detected. (§ 4.1).
- ***How to identify a congested flow upon its arrival?*** For each arriving flow to a switch, its entire following path should be deterministic to the switch to identify a congested flow. Inspired by recent industrial path control practice [73, 52], Pyrrha proposes a hash-function-aware design for switches. Every switch can determine the path that a flow will take. With that information, the switch could match the path against the congestion status snapshot of the downstream network to determine whether it is a congested flow (§ 4.2).
- ***How to handle events-tangling scenarios?*** Congestion trees could overlap with each other which could result in a congested flow traversing several congestion roots. Besides, the congestion root in networks may vary with transient bursty traffic. Without careful scheduling among congestion queues, such flows could be mistakenly paused/resumed or delivered out-of-order. Pyrrha proposes a resource-efficient hierarchical queue structure corresponding to the physical topology. The design ensures both correct flow control semantic and in-order delivery even in highly dynamic scenarios (§ 4.3).

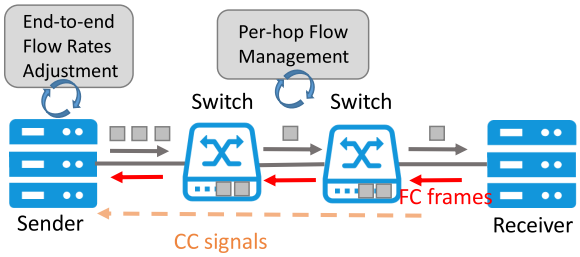


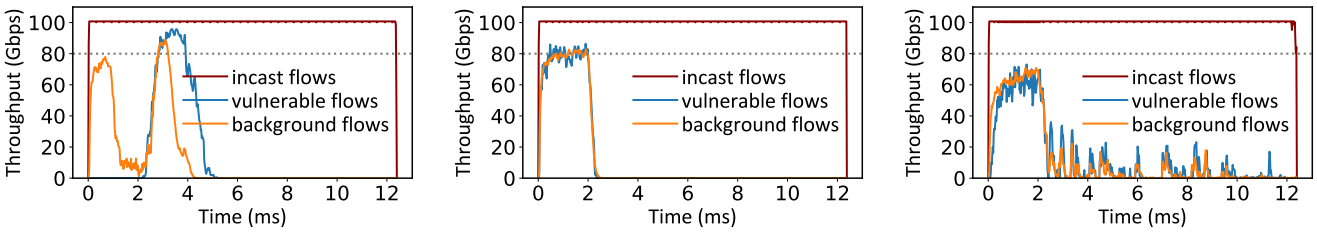
Figure 1: Our vision of labor division between CC/FC.

A prototype of Pyrrha is implemented on Tofino2 [48]. Testbed evaluations and large-scale NS-3 simulations have been performed. We compare Pyrrha with existing flow control protocols (*e.g.*, Priority Flow Control (PFC) [46], and BFC [36]). And we also incorporate Pyrrha with existing congestion control protocols (*e.g.*, DCQCN [95], TIMELY [65], and HPCC [56]). We find that the average FCT of uncongested flows is reduced by 42.8%-98.2%, and 99th-tail latency can be 1.6×-215× lower, without compromising the performance of congested flows. In addition, Pyrrha reduces the maximum buffer occupancy by up to 1.8×-6.2× (§ 6). As a contribution to the research community and to ensure reproducibility, our artefacts is made publicly available online [59]. *This work does not raise any ethical issues.*

2 Background and Motivation

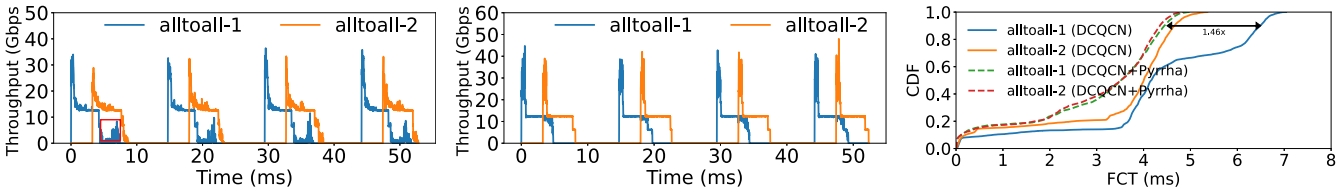
2.1 CC is Falling and FC is Rising

A variety of datacenter applications produce bursty traffic, which can result in different types of congestion, *e.g.*, incast, and load imbalance. To handle congestion, existing efforts focus on developing end-to-end congestion control (CC). CC can be classified into *reactive* and *proactive*. With reactive CC, congestion can be detected by switches (*e.g.*, ECN in DCTCP [11] and DCQCN [95], INT measurements in HPCC [56], PINT [19], PowerTCP [6] and Poseidon [86]) or end-hosts (*e.g.*, Timely [65], Swift [55], and On-Ramp [61]). After receiving congestion signals or if packet delays are observed, senders adjust the transmission rate. It may cost a flow several RTTs to converge to an appropriate rate even in a stable network condition. With proactive CC, bandwidth is allocated before the transmission (*e.g.*, ExpressPass [25, 58], Homa [67], NDP [41], Aeolus [43], and pHost [31]). However, whether to transmit in the first RTT is a dilemma, and proactive CC either wastes the first RTT or risks reintroducing congestion. Recently there are CCs [14, 69] which detect congestion at sub-RTT by



(a) PFC. (b) per-flow queue flow control. (c) BFC.

Figure 2: Performance when incast flows are mixed with non-incast flows.



(a) Throughput (DCQCN). (b) Throughput (DCQCN+Pyrrha). (c) FCT comparison.

Figure 3: Performance comparison under MoE workloads (collided phase).

leveraging switches to send back control frames directly. However, they can not quickly react to congestion especially when congestion occurs at the last hop (*e.g.*, incast).

Several trends. The control loop of CC is too long to handle transient congestion, given the fast evolution of datacenter networks: (i) The high port bandwidth allows to send out more flows within the first RTT, even before congestion control could step in [36, 43]. Transient bursty traffic results in a large buffer occupancy and at the same time mislead the rate adjustment of CC. (ii) The buffer size cannot catch up with the increased speed of its high bandwidth per port [36, 22]. It becomes harder for switches to buffer transient congestion and wait for end-to-end CC's intervention. (iii) The growing scale of datacenter networks and the emerging workloads (*e.g.*, distributed training) lead to more bursty traffic (*e.g.*, a larger scale incast) [74, 42, 75].

Our vision. To handle bursty traffic, per-hop FC protocols should step in. As shown in Figure 1, we propose a labor division between CC and FC:

- **(i) Per-hop FC handles transient congestion.** A switch can control the traffic transmission quickly by per-hop flow control frames. It is in a unique position to quickly manage flows that have already been injected into the network to avoid performance downgrades.
- **(ii) CC takes its part to handle persistent congestion.** CC can adjust the flow rates to increase/decrease the forthcoming traffic injection into the network when congestion occurs and to handle fairness issues.

2.2 HOL Blocking Problem of Simple FC

In RoCEv2 [16], PFC [46] ensures that the buffer does not overflow. PFC pauses the upstream entity at a per-port or per-priority-class queue granularity when the ingress queue length exceeds a given threshold. Further, when the upstream ingress queue exceeds the threshold, a pause frame will be sent to its upstream entities.

However, the intervention of PFC could spread congestion. When PFC is applied, flows that do not contribute to downstream congestion could be paused when they share the same queue with congested flows. The congestion scope can spread from congested ports to piles of innocent ports. Hence, it could downgrade the performance of uncongested flows. We define the above congestion spreading phenomenon as the **HOL blocking caused by flow control**, *i.e.*, a flow is paused innocently by the congested port that it does not pass through. HOL blocking could cause a throughput downgrade. More severely, PFC is vulnerable to deadlock with routing loops [40, 54, 62, 57].

Typical incast workloads. To demonstrate the HOL blocking problem, we conduct a simulation where incast flows are mixed with non-incast flows. 720-to-1 incast flows are generated with an average size of four Bandwidth-Delay Product (BDP) and non-incast flows are generated with a load of 0.8 following the Poisson arrival process (setting details in § 6). Figure 2(a) depicts the real-time throughput. To make it more clear, we use ***vulnerable flows*** to denote uncongested flows sharing paths with congested flows in the remainder of the paper since they are more likely to be hurt by congested flows. Other uncongested flows are denoted as ***background flows***. Hence, the throughput of vulnerable flows is severely hurt since they are paused by downstream switches with congested flows as a whole, leading to a large queuing delay. Besides, since a PFC pause frame storm occurs, congestion is spread to the whole network. Consequently, background flows suffer a throughput downgrade from 1ms to 4ms.

MoE workloads. We investigate the performance of Pyrrha under the traffic of a popular type of pre-trained large language model called Mixture-of-Expert (MoE). According

to [42], the traffic pattern can be characterized by an imbalanced alltoall where a significant portion of the traffic is sent to a few 'hot' experts. Owning to the synchronization nature of the training process, the traffic exhibits a periodic on-off pattern [75, 76]. Following [75], two groups of periodic traffic are generated.

Figure 3 shows the performance of flows with collided phase, *i.e.*, where their phase overlaps (the results of the interleaved phase are detailed in Appendix B). In the case of DCQCN+PFC, two groups of alltoall suffer from HOL-blocking as they compete for bandwidth, which in turn triggers PFC. The peak bandwidth lasts for approximately 1 ms during which non-hot experts complete their traffic reception, followed by hot experts continue receiving their traffic. The throughput of the alltoall-1 group is notably suppressed, even dropping to zero upon the arrival of the alltoall-2 group. Once a portion of the alltoall-2 flows finishes, alltoall-1 begins to grasp some of the bandwidth, as indicated by the red rectangle in the figure. While for DCQCN+Pyrrha, benefiting from the rapid reaction to the congestion, two groups of flows do not disturb each other, accelerating the tail latency by a factor of 1.46×.

2.3 State-of-the-art Flow Control is Flawed

To overcome the HOL blocking problem caused by coarse-grained control on queues, a naïve scheme is per-flow queue FC scheme. Figure 2(b) demonstrates the simulation results under the same settings as in Figure 2(a) when the switch assigns a dedicated queue to each flow passing through it. Vulnerable and background flows fully utilize the link, at the same time the throughput of incast flows does not downgrade (see [59] for theoretical analysis). However, tens of thousands of flows can be observed on ports [84]. Hence a per-flow granularity scheme is non-scalable.

Existing flow control approaches try to reduce the number of queues by compromising the isolation granularity.

Destination-based flow control. This line of work tries to isolate congestion by separating flows transmitting to different destinations. Revisiting super-computing literature decades ago, per-destination Virtual Output Queues (VOQs) [23, 27, 51] are assigned to separate flows with different destination addresses [85, 68]. However, per-destination VOQs are not scalable since the number of VOQs required scales with the number of hosts in networks. Floodgate [60] is a per-hop flow control leveraging per-destination windows to identify incast traffic in datacenter networks. Then, incast traffic can be isolated from non-incast traffic. However, it should maintain a per-destination state of the remaining sending window which demands much memory resources on switches. In summary, they only aim at eliminating HOL blocking caused by the last-hop incast and cannot handle other types of congestion such as load imbalance. In addition, they require per-destination resources which build barriers to deployment at scale.

Queue-based flow control. A second line of work targets at assigning flows into a limited number of queues to alleviate HOL blocking. Since it cannot isolate congested flows from uncongested flows entirely, HOL blocking cannot be avoided. In BFC [36], flows are assigned to a number of queues (*i.e.*, 32-128 queues per port) according to their *flow-id*s and hash functions. A flow is assigned to an empty queue if possible and could share it with other flows when all queues are occupied. As shown in Figure 2(c), with relatively large incast flows, both vulnerable and background flows maintain a very low throughput from 2ms to 12ms. The tail latency of flows is prolonged by 6× compared with the per-flow queue scheme. This is because incast flows can occupy queues for a long time. Vulnerable and background flows sharing the same queue with incast flows are severely hurt, and their transmission rate is mistakenly controlled by the network bottleneck (*i.e.*, the destination ToR of incast).

To sum up, existing solutions cannot totally avoid HOL-blocking, and some of them are impractical.

3 Pyrrha Overview

Our target is to eliminate HOL blocking caused by flow control in the most cost-effective way. In this section, we first illustrate why congestion root is the appropriate granularity of flow control along with the basic idea of Pyrrha, followed by a list of challenges.

3.1 Basic Idea

Before illustrating our basic idea, we introduce several concepts in flow control when congestion occurs. Figure 4 shows a typical congestion tree rooted at P5.

- **Congestion Hotspot.** When the congestion occurs, a flow control scheme starts to pause upstream, inducing several hotspots along its back-pressuring path. A congestion hotspot is an output port whose input rate exceeds its output rate and its queue accumulates.
- **Congestion Root.** As its name implies, a congestion root is the root cause of the congestion, where congested flows finally aggregate. Meanwhile, it is the root of the corresponding congestion tree.
- **Congestion Tree.** A congestion tree can be made up of a root (*e.g.*, P5), non-root hotspots (*e.g.*, P1-P3) and leaf ports (already controlled by the root but have not paused its upstream yet). In our paper, a congestion tree is named after its root (*e.g.*, T5 denotes the tree whose root is P5).

Why congestion-root-based FC? A non-differentiating treatment of flows passing through congestion roots and hotspots could result in HOL-blocking since flows passing through hotspots might not contribute to the congestion. To avoid involving innocent flows, flow control should decide the right scope of flows to control. Intuitions are that if a flow control only applies pause to flows contributing to the congestion root, HOL-blocking can be eliminated.

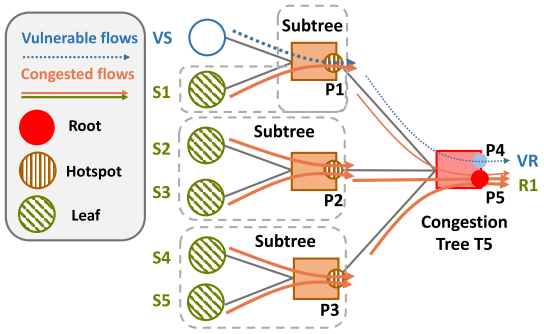


Figure 4: Congestion tree illustration.

Meanwhile, flows passing through different congestion roots should be handled separately to avoid interfering with others.

Quick reaction to congestion: identify congested flows upon arrival. Once a congestion root is detected, the congestion root information is propagated to its upstream switches when it is detected. Then each switch maintains a congestion status snapshot of its downstream networks. A congested flow can be detected upon its arrival in networks by checking whether its path matches existing congestion roots. Hence, its transmission can be controlled several hops earlier before it arrives at congestion roots. It reduces the occurrence of severe congestion and relieves the buffer pressure on following hops, especially on congestion roots.

Fine-grained isolation: manage traffic according to its contributed congestion roots. By default, flows are pushed into a physical output queue (OQ). For separate control, congested flows passing through the same congestion root should be pushed into a dedicated isolation queue (IQ) assigned to the corresponding congestion root. Different categories of flows, *e.g.*, congested flows contributing to different congestion roots, and uncongested flows are isolated respectively. Then, the transmission of congested flows can be controlled precisely by pausing the exact queue assigned to the congestion root, which avoids congestion spreading.

Based on the above ideas, we propose Pyrrha, a practical fine-grained flow control scheme based on congestion roots. Pyrrha achieves good properties, as shown next.

Theorem 3.1. *Pyrrha is a HOL-blocking free per-hop flow control protocol requiring the minimal number of queues.*

Proof. Inspired by the gradient graph [78, 77], we represent flows and queues as vertices, and their traversing relationship as edges. Congested flows contributing to the congested queue's congestion can be represented as its immediate neighbors. Based on this representation, congested flows can be identified as the neighbors of a congested queue. Similarly, potentially paused flows by a congested queue can be identified in our graph representation. In order to prove that Pyrrha is HOL-blocking-free, we formally prove that the set of congested flows and the set of potentially paused flows are identical for any congested queue. Futhermore, we prove that any flow control using fewer queues than Pyrrha cannot avoid HOL blocking. We use Dirichlet's box principle and proof by contradiction to establish this impossibility result. Our formal proof appears in [59]. □

3.2 Design Challenges

Identify congestion roots. Correctly identifying the congestion root is the prerequisite of proper flow control. In tree T5 of Figure 4, P1-P3, and P5 should agree that P5 is the root of incast. Otherwise, if P1 incorrectly identifies itself as an independent root, vulnerable flow VS→VR becomes a victim.

Identify congested flows. To control the transmission of flows separately, congested flows should be identified precisely. When an uncongested flow is mistakenly identified (*e.g.*, VS→VR) as a congested flow, it can be paused innocently. To recognize a congested flow quickly, a switch should obtain the flow's path to be traversed in its downstream network so that it can check the path against its downstream network snapshot.

Handle tree-tangling scenarios. Several concurrent congestion trees can be intertwined among themselves. If non-root ports of trees are overlapped, these ports can play different roles in different congestion trees. These ports should not entangle transmission control received from different trees. When a tree is covered by another one, it results in a flow contributing to several congestion roots, which appear frequently in real systems. One may wonder why congestion still occurs at a port after part of its passing flows is controlled by its downstream congestion roots. This is because the congestion at the port is caused by itself other than its downstream ports. Figure 5(b) depicts an example that the path of a flow can first match a congestion root (P6) and then match another congestion root (P5). The transmission of the congested flow should be controlled by both congestion roots, *i.e.*, only when both congestion roots send a RESUME frame can the flow be transmitted. Besides, the transmission control of the flow should not interfere other flows sharing one of the congestion roots.

In addition, when the competing traffic changes, the congestion tree can shrink or expand, leading to congestion roots varying over time. For P3 in Figure 4, initially, packets of flows {S4-S5}→R1 are controlled in the IQ_{P5}. If flows {S1-S3}→R1 finish, P3 is likely to become the new congestion root later. Flows {S4-S5}→R1 are no longer the congested flows of P5, instead, they are congested flows of P3. Later-arrived packets of these flows should be carefully scheduled in case they are transmitted before packets previously queued into the IQ_{P5}, which results in out-of-order delivery.

4 Pyrrha Design

Figure 5 demonstrates the architecture of Pyrrha. The bottom part of the figure denotes the packet propagation among Pyrrha switches. And the upper part depicts the three major components of a Pyrrha switch. Congestion

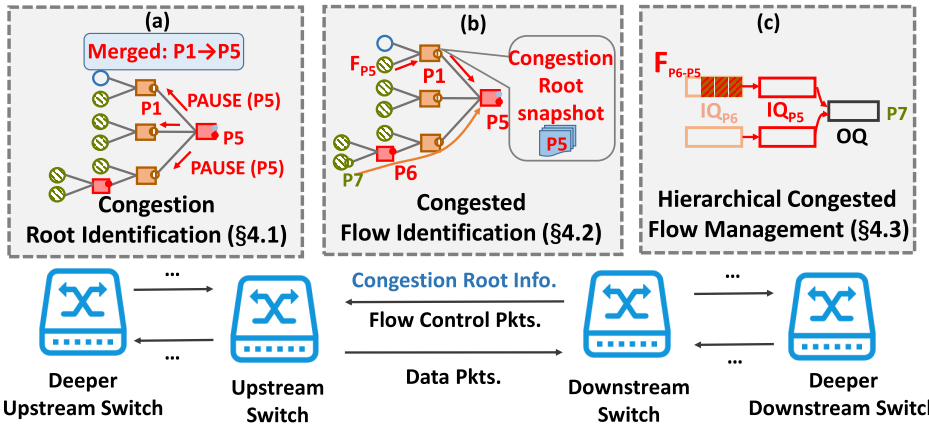


Figure 5: Pyrrha architecture.

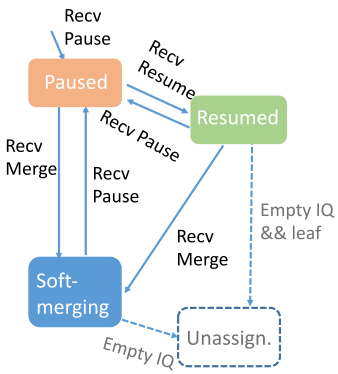


Figure 6: IQ state transitions.

Root Identification (§ 4.1) responds to downstream switches to detect congestion and identify the corresponding root. The congestion information is carried in flow control frames (*e.g.*, PAUSE) and propagated to upstream switches in a hop-by-hop manner. Congested Flow Identification (§ 4.2) maintains the snapshot of downstream network congestion states to help quickly recognize congested flows through path matching. Isolation queues (IQs) are structured in a hierarchy corresponding to the topology by Congested Flow Management (§ 4.3). Besides, details and discussions are put in § 4.4 and § 4.5.

4.1 Congestion Root Identification

Initial detection. Inspired by the root-election process of spanning tree protocols, a congested port can claim itself a *self-nominated* congestion root candidate independently. Initially, each port is attached with a OQ and flows are pushed into the OQ by default. Hence the queue length increase on the OQ can be regarded as an indication of congestion. When a data packet arrives at the OQ, a switch checks whether the queue length exceeds a given threshold K_{pause} (*e.g.*, several per-hop BDPs). If so, a hotspot is detected and the hotspot regards itself as a congestion root. Subsequent packets that arrive at the hotspot trigger a PAUSE frame to the corresponding upstream port from which the packet arrived.

Congestion root identification. According to behavior of flow control, a root is always the most downstream hotspot in a tree. Hence, we can identify the real congestion roots by merging upstream congestion tree into a more downstream one. As shown in Figure 5(a), when a congestion-root-candidate hotspot (P1) receives a PAUSE frame from a downstream root (P5), it indicates that part of its passing-through flows also traverses this downstream hotspot. Hence it recognizes itself as a false-positive congestion root. A new IQ for this new congestion root (P5) is assigned. All following packets matching the new root will enter the corresponding IQ. Then, the old congestion tree is canceled and merged into the new congestion tree. Note that this process can be iterative when there exist multiple layers of hotspots in a congestion tree. The false-positive congestion roots are eventually merged and the root of the new congestion tree is the real congestion root.

Merging process. To start merging, the false-positive congestion root notifies all its child nodes by sending a control message MERGE. MERGE is sent to all its upstream entities belonging to the (old) false-positive congestion root, carrying the ID of both old and new congestion roots. As shown in Figure 6, switches receiving the MERGE frame change the state of the corresponding IQ to soft-merging and propagate the notification to its upstream further. Soft-merging means that the old IQ now belongs to no congestion tree and can be unassigned once empty. The packets queuing in the old IQ are not controlled by the false-positive congestion root. Instead, only packets passing through the real congestion root are controlled (§ 4.3). The merging process finishes within a one-way delay, hence false-positive congestion roots have a negligible impact on performance.

4.2 Congested Flow Identification

Intuitions are that a congested flow passes through at least one congestion root.

Determining a flow's exact path. To determine whether a data packet belongs to a congested flow, the entire onward-path of each arriving packet at a switch should be deterministic. Pyrrha is compatible with traffic load balancing protocols that can locally get deterministic onward paths for flows [1, 44, 79, 64, 52]. Among those load balancing protocols, hash-based protocols (*e.g.*, per-flow ECMP and PLB [73]) are most widely deployed given its no-reordering properties [87]. Pyrrha proposes a hash-function-aware design. Pyrrha's switch calculates a packet's onward-path by using its IP tuples, routing hash functions, and seeds of its downstream switches, together with flow labels carried in its header if necessary as input. Besides, source routing is compatible with Pyrrha naturally since Pyrrha's switch can derive the onward-path of a packet by parsing its header.

The memory and computation resource to get a flow's path

is moderate, and following optimizations are facilitated by leveraging the industry-standard practices in datacenters. (i) Given that switches in datacenter only support limited types of hash functions (*e.g.*, CRC or XOR) to conduct efficient calculation [87, 93, 72], Pyrrha switch only needs to store the type of the hash function of other switches along with their hash seeds. (ii) In many widely deployed topologies [8, 37, 62, 82], the multiple equal path property and the up-down routing strategy can be leveraged to optimize the overhead. In those topologies, the forwarding tables of all core switches are identical. Hence, Pyrrha switch only needs to store one replica for redundant forwarding tables. Furthermore, for fat-tree, the path from a core switch to a given destination is unique, hence Pyrrha switch only conducts hash function calculation for the first two hops.

For other adaptive load balancing approaches, *e.g.*, per-packet spraying and DRILL [67, 34], which determine the path of flows through dynamic states, Pyrrha is a complementary solution to handle the hotspots caused by destination collision (*i.e.*, incast), where load balancing falls short. Experiments in Appendix 6.3 show that Pyrrha is compatible with DRILL and further improves the tail latency by 18.3% compared to pure Pyrrha. Pyrrha also handles corner cases where flows are re-routed to different paths due to link failures (Appendix § A).

Matching and maintaining snapshot. Pyrrha's switch maintains the *congestion-root table* of the downstream networks. As shown in Figure 5(b), the table maintains a snapshot of the congestion states of its downstream networks. When a PAUSE frame indicating a new congestion root is received, the congestion root is recorded in the table. When a packet arrives at the switch, its path can be obtained via above mentioned methods. The switch checks whether its path matches any entry in the congestion-root table. For port P1 in Figure 5(b), packets that will traverse P5 in its downstream path are identified as belonging to a congested flow. Hence, it's enqueued to a separate IQ, which is paused/resumed based on the state of the corresponding root. Otherwise, the packet belongs to an uncongested flow and is put in the OQ.

4.3 Congested Flow Management

To handle tangled scenarios where congestion trees are intertwined among each other or congestion roots vary over time, Pyrrha leverages a hierarchical methodology to manage congested flows corresponding to that of the topology. It can be supported by a Hierarchical Isolation Queue (HIQ) architecture, which manages congested flows in a hierarchy. Pyrrha installs HIQ during compilation according to its location in networks and manages the usage of queues dynamically during runtime through a mapping table. We also provide single-tier IQs prototype to fully support the function of HIQ.

Handling congested flow in hierarchy. Congestion trees are intertwined when non-root ports of trees are overlapped or a tree is covered by another one. For the first scenario, IQs on ports can naturally isolate control from different congestion roots on non-root ports. For the latter one, a congested flow could pass through multiple congestion roots. To ensure precise control isolation, a congested flow should match all corresponding IQs before it is forwarded. A hierarchical organization of IQs based on the location of their corresponding congestion roots in the topology enables a congested flow to match appropriate IQs *sequentially*. Especially, when per-flow load balancing is used, a flow at most encounters one congestion root among switches at the same level. It ensures in-order delivery when a congested flow alters its matched IQs.

Hierarchical Isolation Queue (HIQ) architecture. HIQ consists of several levels of IQs. Each IQ is positioned in a hierarchy according to its distance to the corresponding congestion root in the physical topology. Hence, the number of layers of the HIQ is determined by its location in the network. As shown in Figure 5(c), in a two-tier network, an uplink port of a ToR switch maintains two levels of IQs, since the farthest potential congestion root is two hops from it. Figure 5(c) depicts the HIQ architecture on P7, *i.e.*, the leaf port in Figure 5(b). P7 fully utilizes the two-level architecture of the HIQ, since the farthest congestion root P5 is two hops from it. Especially, the OQ is connected to the last level of the HIQ architecture. A dedicated scheduler is equipped for each level of queues to schedule the traffic transmission. Only when an IQ is in a resumed or soft-merging state can its packet be dequeued and pushed into the next-level IQs/OQ. From the perspective of a packet, this process is performed iteratively until it reaches the OQ. In the scenario depicted in Figure 5, when a flow that will traverse congestion root P6 and P5 arrives at the upstream switch of port P6, it matches the HIQ *from-near-to-far*. After a packet is dequeued from IQ_{P6}, it is pushed into a next-level IQ_{P5}.When there is no more matched IQ, it is pushed into the OQ. Hence, the packet can be forwarded to the next hop only after all matched congested roots are resumed. In this way, congested flows are controlled locally precisely.

IQs in HIQ are arranged by levels, supporting in-order delivery naturally. Considering the merging procedure in Figure 5(a), the congestion root is changed from P1 to P5. IQ_{P1} is in the soft-merging state and packets in it can be mixed with congested and uncongested traffic. Pyrrha should handle them separately to avoid HOL blocking. Congested flows of root P5 are dequeued from IQ_{P1} and then pushed into the next level IQ_{P5}. Uncongested flows are forwarded to OQ. In this way, precise isolation is achieved without inducing re-ordering.

Handling secession of the congestion root. Once the OQ length of the congestion root decreases below the resume threshold, it sends back RESUME to its upstream. Likewise, upstream switches could resume their upstream when their own IQ decreases below the resume threshold. It is an

iterative process that the congestion tree eliminates starting from the leaf switches to the root. A congestion port becomes a leaf when it has not sent PAUSE yet, or when the status of all its upstream entities is set to *unassigned*. The IQ of the leaf switch is unassigned when it becomes empty. When all the upstream IQs of a congestion root are marked unassigned and the queue length of OQ of the congestion root is below the resume threshold, the congestion root disappears naturally. Figure 6 depicts the state transition of the IQ usage. Only when an empty IQ is in a resumed or soft-merging state can it be marked as unassigned.

4.4 Miscellaneous Detailed Design

Congestion information propagation. There are three types of flow control frames in Pyrrha, *e.g.*, PAUSE, RESUME, and MERGE. These control frames carry the congestion information and control the transmission of congested flows accurately in a hop-by-hop manner. Correspondingly, there are three states for an IQ, *i.e.*, paused, resumed, or soft-merging. The state transitions of an IQ are shown in Figure 6.

When a congestion root is detected, a PAUSE frame is sent back to the upstream port through the ingress port of which the data packet is just received. Once a data packet is pushed into the OQ which is attached to the congestion root, Pyrrha switch checks the packet's ingress port and sends back the PAUSE frame. A PAUSE frame carries the ID of the congestion root (*i.e.*, identified as *switch-id:port-id*). Likewise, when the queue length of IQ exceeds the threshold K_{pause}, a PAUSE frame carrying the root ID is sent back to its upstream switch.

Cooperation on end-hosts. To handle persistent congestion, end-hosts should control the upcoming traffic into networks. Pyrrha can cooperate with congestion control protocols. And Pyrrha can perform better if end-hosts can respond to PAUSE (or RESUME and MERGE) frames. To pause and resume at a per-flow granularity, end-hosts could leverage a pull-based transmission model, which can be implemented by programmable smart NICs in RDMA networks. Especially, Pyrrha can also handle end-host congestion (*e.g.*, PCIe congestion) by backpressuring the traffic it receives.

Handling rare packet loss. Pyrrha reduces queuing length significantly. Hence, buffer overflow rarely occurs. However, Pyrrha does not guarantee lossless for rare cases where every port of a switch becomes the victim port of a k-1:1 incast lasting for one-hop RTT in turn, where k denotes the number of ports on switches. In this certain case, Pyrrha can start to drop packet and leverage IRN [66] for fast retransmission.

4.5 Scalability Discussions

Queue consumption. In the most extreme scenario, the number of congestion roots can be the number of ToRs in the network. However, the concurrent amount of congestion in networks is usually moderate. It is reported that only 3% of the links in edge and aggregation layers appear as a hotspot

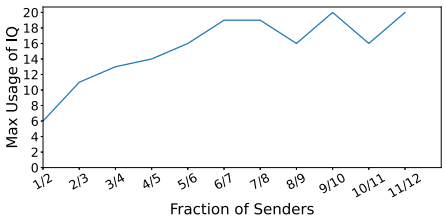


Figure 7: Queue usage analysis.

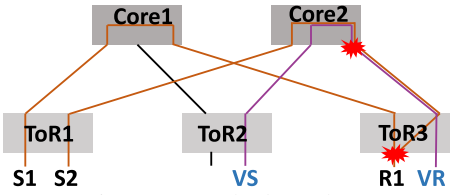


Figure 8: Testbed topology.

for more than 0.1% of time intervals [20, 21]. Moreover, the concurrent roots can be much less than that of hotspots.

To investigate the IQ usage, evaluations under stressful workloads where $(m-1)$ out of m ToRs send traffic simultaneously to the left 1/m of the ToRs are conducted. A k=12 fat-tree topology is employed and each host sends 40 one-BDP flows continuously to create a substantial network burst. Figure 7 illustrates the IQ usage of Pyrrha as the fraction of senders varies. The IQ utilization ranges from 6-20, approximately proportional to the number of pods k, which considered to be relatively moderate. Since commodity switches can support thousands of VOQs [23, 27, 51], assigning a dedicated queue to each downstream root is feasible. To handle corner cases where IQs are not enough, similar to BFC, Pyrrha leverages hash functions [39, 71] to choose an IQ according to the congestion root ID, at a cost of sacrificing precise isolation.

Deadlock prevention. Pyrrha is deadlock robust since OQs never get paused. To prevent cyclic buffer dependencies (CBD) caused by routing loops, Pyrrha switch checks whether the congestion root carried in the PAUSE frame is identical to its own identifier. If so, it ignores the PAUSE frame directly. (see detailed discussions in [59].)

5 Implementation and Testbed Experiments

We implement a Pyrrha prototype on Tofino2, a state-of-the-art programmable switch ASIC [48] with Reconfigurable Match Table (RMT) architecture. In this section, we briefly describe the key modules of the prototype, followed by the overhead analysis. Testbed evaluations show that Pyrrha can achieve good performance (§ 5.2). More implementation details are deferred to Appendix D and E.

5.1 Prototype of Pyrrha

We implement a Pyrrha prototype on Tofino2 with 2.5k lines of P4 code and 2k lines of Python code. The operations of Pyrrha is implemented entirely in the data plane at line rate.

Key modules and the pipeline. The Pyrrha prototype is

mainly composed of several modules, *i.e.*, (i) congestion root matcher, (ii) queue manager, (iii) queue state detector, and (iv) signal packet module.

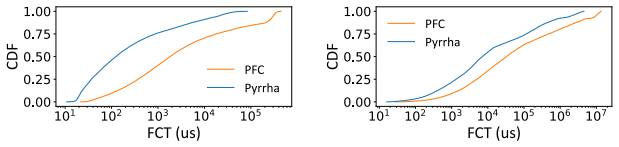
(i) Upon arrival, the data packet first undergoes a standard processing procedure, including forwarding and admission control. Subsequently, the data packet is forwarded to the congestion root matcher, an integral component of the path calculation unit and a congestion root table, facilitating the congested flow identification. Specifically, the path calculation unit calculates the packet's egress ports of its onward path. The congestion root table records the status of ports, indicating whether the port is congested or not. (ii) Then packets enter the queue manager for queue assignment. The queue manager assigns queues to traffic based on the congestion roots it would pass through. Central to its design is a multi-segment stack, wherein each segment manages the available queues for a specific egress port.

(iii) The queue state detector checks whether the length of the assigned queue exceeds the pause threshold or decreases below the resume threshold, and then triggers appropriate signal packets. The queue length is retrieved by utilizing *ghost threads* in Tofino2.

(iv) When it is necessary to send a signal packet, the signal packet module leverages the packet trigger functionality to construct signal packets, such as PAUSE and RESUME. Upon receiving a PAUSE or RESUME, the module engages Tofino2's *AFC (Advanced Flow Control)* mechanism to pause (or resume) the queue.

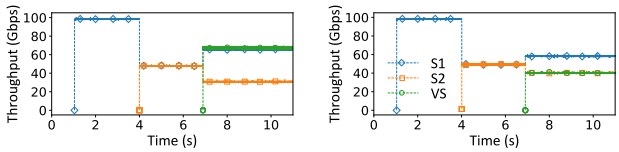
Feasibility of HIQ The architecture of HIQ is supported in current Metro Ethernet (MetroE) service routers [3, 2]. And a recent work of implementing multi-level scheduler on ASIC [94] also verifies its feasibility. According to private talks with chip vendors, they consider it possible to implement HIQ in their next-generation switching chips. For instance, the two-layer HIQ can be obtained by connecting two traffic manager models in series and specifying the next-level IQ to be pushed in when a packet is dequeued. Although HIQ is not supported by the architecture of Tofino2 currently, the features of HIQ can be fully supported via single-tier queues (Appendix C).

Complexity and overhead. Tofino2 adopts pipeline architecture, wherein the resource allocation is determined at compile time. It enables us to ascertain Pyrrha's resource requirements without running it in a large-scale cluster. According to the statistics reported in megascale [49], the scale of current data centers can reach up to 10,000 hosts. Therefore, we use a k=36 fat-tree topology with 11,664 hosts as a representative case. Pyrrha prototype can easily scale to it, with around 11 MB (*i.e.*, 44.5% of Tofino2) of the memory resource consumption. Specifically, the memory usage of Pyrrha prototype is mainly composed of three units, *i.e.*, path calculation unit, congestion root table, and queue manager, overall consuming 9.25 MB. And the processing logic consumes around 1.88 MB. (i) As analyzed in § 4.2,



(a) Vulnerable flows (Web Server). (b) Vulnerable flows (Web Search).

Figure 9: FCT of testbed experiments.



(a) Pyrrha (b) PFC

Figure 10: Throughput of testbed experiments.

the memory consumption of the path calculation can be optimized by leveraging the industry standard, occupying 0.44MB SRAM. (ii) The congestion root table is organized hierarchically, where the n^{th} table records ports that are n hop away from the switch, and the port is identified as <switch-id, port-id>. Hence, the storage usage of the congestion root table is proportional to the number of ports in networks, occupying 176 KB SRAM. (iii) The queue manager firstly checks whether the congested flow is assigned with a queue through a *IsAssigned Table* and assigns a queue if necessary by looking up a multi-segment *QueueId stack* which records the available queue. Then it updates the *QueueId record table* to record the queue assignment status. This module overall consumes 8.64MB SRAM. Furthermore, leveraging the rapidly maturating HIQ technology [94], Pyrrha can scale to topologies that is an order of magnitude larger. (detailed analysis are deferred to Appendix E).

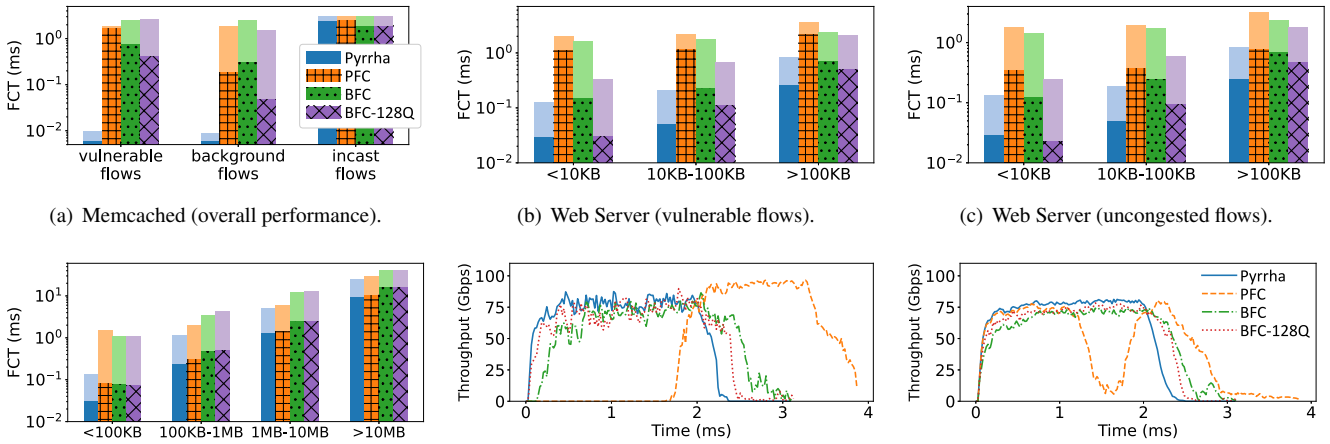
5.2 Testbed Evaluation

Topology. We use a 2-level leaf-spine topology as shown in Figure 8, consisting of three ToR, two core switches, and two hosts per rack, all connected via 100 Gbps links.

Workloads. We evaluate Pyrrha under incast-mix scenarios. Incast flows are generated by letting hosts S1 and S2 transmit flows to host R1 simultaneously. Vulnerable flows are generated by letting host VS send flows to host VR. Flows S2→R1 and VS→VR share the same port on the core switch. Web Server and Web Search flows are generated following a Poisson arrival process (§ 6).

Pyrrha reduces the FCT of vulnerable flows. Figure 9 demonstrates the FCT performance of vulnerable flows. For PFC, vulnerable flows are HOL blocked by incast flows, suffering a large queuing delay. Pyrrha quickly detects the congestion on the destination ToR switch and isolates incast flows into a dedicated queue. Thus, the FCT of vulnerable flows is greatly reduced.

Pyrrha improves the throughput. Figure 10(a) shows the throughput when flows on three hosts S1, S2, and VS start



(a) Memcached (overall performance). (b) Web Server (vulnerable flows). (c) Web Server (uncongested flows).

(d) Web Search (uncongested flows). (e) Web Server (vulnerable flows). (f) Web Server (background flows).

Figure 11: Performance of FC. The deep/light color in the figures represents the average/99th-tail value for each bar.

to arrive at 1s, 4s, and 7s, respectively. Pyrrha improves the throughput of vulnerable flows to 66.7 Gbps without compromising the overall throughput of incast flows (100 Gbps). The network throughput is improved by 26.7 Gbps.

6 Simulation Evaluation

Topology. A non-blocking clos-network is used. It contains 4 core switches, 10 ToRs, and 160 hosts, similar to the topology used in [67]). Each ToR is connected to its hosts and cores via 100/400 Gbps links, respectively. The per-hop propagation delay is 600ns. The base RTT is 5.1μs, and the base BDP is 64KB. A 3-tier fat-tree topology with 1024 hosts is also leveraged to investigate the scalability of Pyrrha.

Workloads. Under incastmix scenarios, flows following a Poisson arrival process with a load of 0.8 and periodic incast flows each composed of 30-40 MTUs with a load of 0.5 are generated. An incast destination does not receive Poisson arrival flows hence the traffic load does not exceed link bandwidth. The incast degree is 720-to-1. For Poisson arrival flows, three workloads are used [67, 80, 11], where Memcached is composed of small flows, where most of the flows are smaller than 1KB, and Web Server and Web Search are large flows mixed with small flows where a small ratio of large flows dominate the average flow size.

Parameters. There are two parameters of Pyrrha, *i.e.*, the threshold to pause (*i.e.*, K_{pause} times one-hop BDP) and the threshold to resume (*i.e.*, K_{resume} times one-hop BDP). In our evaluations, K_{pause} is set to 2, and K_{resume} is set to 1. Besides, the maximum number of IQs can be used is set to 100, but Pyrrha only uses a dozen of IQs in most cases. A dedicated subsection discusses why these values are used (§B.1). The switch buffer capacity is 20MB. Pyrrha uses shared buffer mode. PFC uses dynamic threshold and $\alpha = 2$.

Metrics. Average/99th-tail FCTs are evaluated. We monitor maximum buffer on each hop to investigate the composition of buffer reallocation that Pyrrha brings.

6.1 Comparing with Flow Control

In this section, we use large-scale NS3 simulations to compare Pyrrha [59] with existing flow control protocols (*e.g.*, PFC and BFC). For BFC, two versions with 32 and 128 queues per port are used, as in its paper (*e.g.*, BFC is used to denote BFC-32). Figure 11 shows the FCT and throughput.

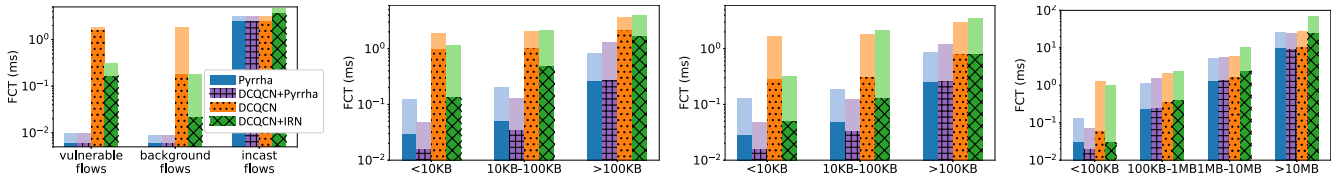
6.1.1 PFC

PFC hurts the performance of uncongested flows since they can be paused innocently by their downstream ports when incast occurs. This especially hurts the performance of workloads that are composed of small flows (*e.g.*, Memcached). For Web Server workload, PFC even spreads the congestion to the whole network thus background flows that do not share the same destination pods of incast flows also get hurt. The throughput performance depicted in Figure 11(f) is a side note to this issue. For background flows, it achieves stable high throughput until PFC pause frame storm occurs at 1.5ms. Then background flows endure a significant throughput loss that lasts for about milliseconds.

6.1.2 BFC

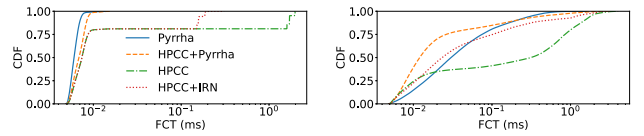
BFC assigns flows to multiple queues according to flow identifiers (FID) and hash functions. It can partially alleviate HOL blocking caused by congested flows and improve the performance of uncongested flows to an extent compared to PFC. However, HOL blocking occurs when congested flows and uncongested flows share the same queue, or flows are hashed into the same flow FIDs. Hence, BFC can not obtain extremely low latency as Pyrrha does. Along with the number of queues used by BFC increases, *i.e.*, from 32 to 128, the performance of BFC is improved. For Web Search workload, the tail latency of BFC is not good because BFC sets a relatively smaller threshold to detect congestion compared to PFC. It can risk spreading congestion.

The third group of bars in Figure 11(a) shows the performance of incast (congested) flows. More results of incast flows are deferred to Appendix B. Pyrrha does not



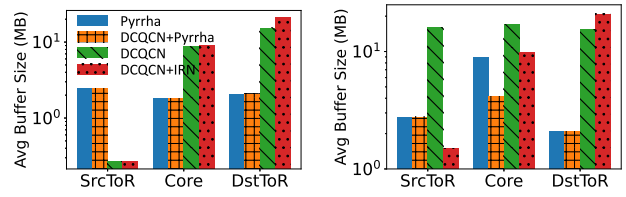
(a) Memcached (overall flows). (b) Web Server (vulnerable flows). (c) Web Server (uncongested flows). (d) Web Search (uncongested flows).

Figure 12: FCT performance (DCQCN).



(a) HPCC: Memcached (overall flows). (b) HPCC: Web Server (uncongested flows).

Figure 13: FCT performance.



(a) Memcached. (b) Web Server.

Figure 14: Buffer occupancy (DCQCN) is split into three parts, *i.e.*, queue length on SrcToR, core and DstToR.

compromise the performance of incast flows compared to PFC. BFC reduces the average FCT of incast flows because BFC splits different flows into different queues and incast flows may use several queues simultaneously. Incast flows can use more bandwidth during the resume phase via the round-robin scheduling mechanism among queues.

6.2 Cooperating with Congestion Control

In this section, we incorporate Pyrrha with state-of-the-art CC (*e.g.*, DCQCN and HPCC). Experiment results of TIMELY is left in Appendix B. The authors' contributed simulation codes, if available, are used in our evaluations [10, 53]. Following [56, 61], a per-flow sending window on hosts is added to all approaches, limiting the in-flight packets of a flow. PFC is used to provide a lossless network for CC by default. Besides, to fully understand the performance of CC, we also evaluate pure CC and use IRN [66] to handle packet loss. IRN leverages selective retransmission to perform an efficient loss recovery.

6.2.1 DCQCN

Since Pyrrha reduces the buffer occupancy by a large extent, we set $K_{min} = 50KB$ and $K_{max} = 200KB$ to cooperate with DCQCN. Figure 12 depicts the average and 99th-tail FCT across different workloads. DCQCN alone can not handle incast flows quickly thus PFC is triggered. Particularly, under Web Server, a PFC pause frame storm occurs. PFC is triggered from the destination ToR of incast and finally reaches the hosts that the source ToR switches connected to, which affects background flows.

Given that PFC has some side effects, one may wonder how DCQCN performs when leveraging IRN for retransmission instead of relying on PFC. However, IRN is not a cure for DCQCN. As shown in Figure 26(b), although DCQCN with IRN reduces the FCTs of small flows compared to DCQCN with PFC, FCTs of larger flows are prolonged. Packet loss hurts the goodput of networks. Besides, when all inflight packets are lost, loss recovery can only get triggered via timeouts. Timeout retransmission hurts the performance of small flows and the tail latency of large flows.

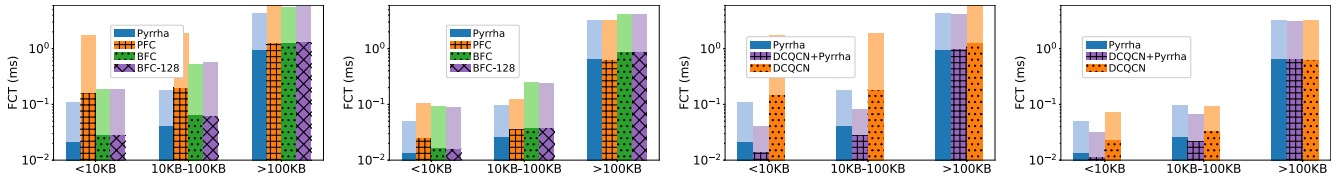
When adding Pyrrha to DCQCN, Pyrrha can significantly benefit uncongested flows (*e.g.*, vulnerable and background flows) by quickly recognizing and isolating incast flows. Pyrrha reacts quickly on in-network congestion by handling it locally, which is especially beneficial to small flows. Therefore, Pyrrha's performance improvement on small flows is the most significant. When the flow size increases, the improvement becomes relatively less obvious. Because large flows give more time for DCQCN to take effect.

Figure 14 shows the maximum buffer occupancy among different hops. Instead of letting congested flows overflow the congestion root, Pyrrha can quickly control their transmission locally. Then the buffer occupancy on the congestion root, *i.e.*, the destination ToR switch, can be greatly reduced. Since Pyrrha pushes back congested flows, the buffer occupancy on the source ToR switch is increased.

From another perspective, when adding DCQCN to Pyrrha, the FCT of small flows can be improved further compared to pure Pyrrha. This is because DCQCN can control the transmission of flows based on a finer granularity than a pause-resume manner. For flows sharing the same path, DCQCN can adjust the rate of each flow individually while Pyrrha controls them as a whole. And congestion control favors small flows since large flows are more likely to be marked and convey the congestion signals back to senders. In addition, the buffer occupancy on the core switch can be further reduced, as shown in Figure 14(b).

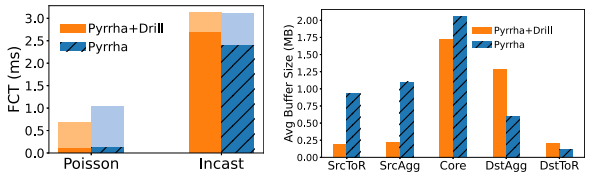
6.2.2 HPCC

When adding Pyrrha to HPCC, uncongested flows carry the queue length of the OQ to avoid unnecessary rate reduction on them. And congested flows carry the queue length of



(a) Web Server (vulnerable flows). (b) Web Server (uncongested flows). (c) Web Server (vulnerable flows). (d) Web Server (uncongested flows).

Figure 15: FCT performance under k=16 fat-tree.



(a) FCT performance. (b) Buffer occcupancy.

Figure 16: Cooperating Pyrrha with DRILL.

IQs. For clarity, Figure 13 shows the FCT distribution of HPCC. Pyrrha helps improve the performance of HPCC. While HPCC achieves good performance on a portion of small flows, as shown in Figure 26(g), this is because HPCC leverages INT to get the in-network information to reduce the queue length aggressively, which benefits small flows. As a cost, its tail latency is prolonged since the transient small flows could interfere with the rate adjustment process and influence the network throughput.

6.3 Additional evaluations and Discussions

Performance under Large-scale Topology. To investigate the scalability of Pyrrha, evaluations under k=16 fat-tree topology (*i.e.*, 1024 hosts) is conducted. Figure 15 shows the performance of different protocols under Web Server. The trend of performance differences between different algorithms is very similar to that of the results in § 6. We analyze the results from two perspectives: (i) When comparing FC with CC protocols, especially for flows smaller than 100KB, fine-grained FC protocols such as BFC and Pyrrha outperform PFC as well as CC protocols. This advantage is attributed to the rapid response of fine-grained FC to network congestion, which provides a degree of mitigation against HOL blocking. (ii) In terms of cooperation between FC and CC, as depicted in Figure 15(c), when Pyrrha is integrated with CC, small flows can achieve the lowest latency. This is because Pyrrha rapidly reacts to the network congestion, preventing HOL blocking for traffic that has already been injected into the network. Concurrently, CC address the persistent congestion associated with larger flows.

Cooperating with adaptive load balancing scheme. Load balancing schemes handle congestion by re-arranging the traffic to avoid path collision. However, they can not handle congestion caused by destination collision (*i.e.*, incast), and Pyrrha acts as a complementary solution to handle it. Pyrrha works together with adaptive routing by only recognizing and handling the congestion of incast.

We investigate the performance when Pyrrha works with DRILL, an adaptive load scheme that chooses the forwarding port according to the queue length dynamically. Similar to § 6, incast-mix traffic is generated and a k=8 fat-tree topology is used. Results in Figure 16 show that Pyrrha works well with DRILL. The integration of DRILL with Pyrrha further reduces the latency of Poisson flows compared to pure Pyrrha. The performance gain is attributed to DRILL's ability to distribute traffic effectively, thereby preventing queue buildup at the upstream (*e.g.*, SrcToR, SrcAgg, and Core), as shown in Figure 16(b). However, this optimization comes with a trade-off in the latency of incast flows, which is higher compared to the pure Pyrrha.

Additional evaluations. In Appendix B, we further explore the performance of Pyrrha under several scenarios, (i) when the size of incast flows varies, (ii) when facing multiple congestion roots, (iii) MoE traffic with interleaved phase, and (iv) when the congestion roots vary. We also compare Pyrrha with per-flow queue scheme and CC without sending window. Besides, the parameter selection and merging mechanism of Pyrrha is evaluated. A variant of Pyrrha using single-tier queues is investigated (§ C.2).

Discussions. Discussions regarding how Pyrrha handles link failures, as well as a review of related works, are put in the Appendix A.

7 Conclusion

This paper was motivated for a labor division between congestion control and flow control. It is time to embrace per-hop flow control in datacenter networks to react to congestion promptly. We presented Pyrrha, a congestion-root-based per-hop flow control. It controls the transmission of flows at a fine granularity without congestion spreading, requiring a minimum number of queues. The performance of flows can be significantly improved. We are currently discussing with a major vendor the implementation of Pyrrha in its products.

Acknowledgments

We sincerely thank our shepherd Gianni Antichi and the anonymous reviewers for their valuable feedback on this paper. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, 62072228, and 62172204, and the Fundamental Research Funds for the Central Universities.

References

[1] Source routing. http://en.wikipedia.org/wiki/Sourcerouting, 2018.

[2] Hierarchical class of service overview. https://www.juniper.net/documentation/us/en/software/junos/cos/topics/concept/hierarchical-cos-overview.html, 2021.

[3] Introduction to hqos. https://support.huawei.com/hedex/hdx.do?docid=EDOC1100168821&id=EN-US_TASK_0172371452&lang=en, 2022.

[4] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI* (2016), pp. 265–283.

[5] ADDANKI, V., APOSTOLAKI, M., GHOBADI, M., SCHMID, S., AND VANBEVER, L. Abm: active buffer management in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 36–52.

[6] ADDANKI, V., MICHEL, O., AND SCHMID, S. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), USENIX Association.

[7] AFEK, Y., KUTTEN, S., AND YUNG, M. Memory-efficient self stabilizing protocols for general networks. In *International Workshop on Distributed Algorithms* (1990), Springer, pp. 15–28.

[8] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review 38*, 4 (2008), 63–74.

[9] ALCOZ, A. G., DIETMÜLLER, A., AND VANBEVER, L. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 59–76.

[10] ALIBABA. HPCC simulator. https://github.com/alibaba-edu/High-Precision-Congestion-Control, 2019.

[11] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *ACM SIGCOMM* (2011).

[12] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM 2013* (2013), vol. 43, ACM, pp. 435–446.

[13] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI* (2010).

[14] ARSLAN, S., LI, Y., KUMAR, G., AND DUKKIPATI, N. Bolt:Sub-RTT congestion control for Ultra-Low latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 219–236.

[15] ARUN, V., ALIZADEH, M., AND BALAKRISHNAN, H. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 177–192.

[16] ASSOCIATION., I. T. Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE), 2014.

[17] ASSOCIATION., I. T. InfiniBandTM Architecture Specification Volume 1 Release 1.4. (2020). https://cw.infinibandta.org/document/dl/8567, 2020.

[18] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS* (2012).

[19] BEN BASAT, R., RAMANATHAN, S., LI, Y., ANTICHI, G., YU, M., AND MITZENMACHER, M. PINT: Probabilistic In-Band Network Telemetry. In *ACM SIGCOMM* (2020).

[20] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *ACM IMC* (2010).

[21] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review 40*, 1 (2010), 92–99.

[22] BROADCOM. Bcm56990 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series, 2019.

[23] BROADCOM. Bcm88800 traffic management architecture. https://docs.broadcom.com/doc/88800-DG1-PUB, 2021.

[24] CAI, Q., ARASHLOO, M. T., AND AGARWAL, R. dcpim: Near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 53–65.

[25] CHO, I., JANG, K., AND HAN, D. Credit-scheduled delay-bounded congestion control for datacenters. In *ACM SIGCOMM* (2017).

[26] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing data transfers in computer clusters with orchestra.

[27] CISCO. Cisco nexus 5548p switch architecture. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-5548p-switch/white_paper_c11-622479.html, 2010.

[28] CISCO. Cef polarization. https://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/116376-technote-cef-00.html, 2013.

[29] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[30] FITZPATRICK, B. Memcached: a Distributed Memory Object Caching System. http://www.memcached.org/, 2011.

[31] GAO, P. X., NARAYAN, A., KUMAR, G., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT* (2015).

[32] GÄRTNER, F. C. A survey of self-stabilizing spanning-tree construction algorithms.

[33] GHABASHNEH, E., ZHAO, Y., LUMEZANU, C., SPRING, N., SUNDARESAN, S., AND RAO, S. A microscopic view of bursts, buffer contention, and loss in data centers. In *Proceedings of the 22nd ACM Internet Measurement Conference* (2022), IMC '22, Association for Computing Machinery.

[34] GHORBANI, S., YANG, Z., GODFREY, P., GANJALI, Y., AND FIROOZSHAHIAN, A. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 225–238.

[35] GOYAL, P., NARAYAN, A., CANGIALOSI, F., NARAYANA, S., ALIZADEH, M., AND BALAKRISHNAN, H. Elasticity detection: A building block for internet congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 158–176.

[36] GOYAL, P., SHAH, P., SHARMA, N. K., ALIZADEH, M., AND ANDERSON, T. E. Backpressure flow control. In *NSDI* (2022).

[37] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Vl2: A scalable and flexible data center network. In *ACM SIGCOMM* (2009).

[38] GROSVENOR, M. P., SCHWARZKOPF, M., GOG, I., WATSON, R. N., MOORE, A. W., HAND, S., AND CROWCROFT, J. Queues don't matter when you can jump them! In *NSDI* (2015), USENIX.

[39] GROUP, T. P. A. W. $P4_{16}$ portable switch architecture (psa). https://p4lang.github.io/p4-spec/docs/PSA.pdf, 2021.

[40] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. Rdma over commodity ethernet at scale. In *ACM SIGCOMM* (2016).

[41] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM* (2017).

[42] HE, J., ZHAI, J., ANTUNES, T., WANG, H., LUO, F., SHI, S., AND LI, Q. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *PPoP* (2022).

[43] HU, S., BAI, W., ZENG, G., WANG, Z., QIAO, B., CHEN, K., TAN, K., AND WANG, Y. Aeolus: A building block for proactive transport in datacenters. In *ACM SIGCOMM* (2020), ACM.

[44] HU, S., CHEN, K., WU, H., BAI, W., LAN, C., WANG, H., ZHAO, H., AND GUO, C. Explicit path control in commodity data centers: Design and applications. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (2015), pp. 15–28.

[45] HU, S., ZHU, Y., CHENG, P., GUO, C., TAN, K., PADHYE, J., AND CHEN, K. Tagger: Practical pfc deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies* (2017), ACM, pp. 451–463.

[46] IEEE. 802.11qbb. priority based flow control. https://1.ieee802.org/dcb/802-1qbb/, 2011.

[47] IEEE. P802.1qcz – congestion isolation. https://1.ieee802.org/tsn/802-1qcz/, 2023.

[48] INTEL. Intel tofino2 – a 12.9tbps p4-programmable ethernet switch. https://ieeexplore.ieee.org/document/9220636, 2020.

[49] JIANG, Z., LIN, H., ZHONG, Y., HUANG, Q., CHEN, Y., ZHANG, Z., PENG, Y., LI, X., XIE, C., NONG, S., ET AL. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (2024), pp. 745–760.

[50] JOGLEKAR, A., KOUNAVIS, M. E., AND BERRY, F. L. A scalable and high performance software iscsi implementation. In *FAST* (2005), vol. 5, pp. 20–20.

[51] JUNIPER. Understanding cos virtual output queues (voqs) on qfx10000 switches. https://www.juniper.net/documentation/en_US/junos/topics/concept/cos-qfx-series-voq-understanding.html, 2017.

[52] KABBANI, A., VAMANAN, B., HASAN, J., AND DUCHENE, F. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), pp. 149–160.

[53] KAIST. Expresspass simulator. https://github.com/kaist-ina/ns2-xpass, 2017.

[54] KAROL, M., GOLESTANI, S. J., AND LEE, D. Prevention of deadlocks and livelocks in lossless backpressured packet networks. *IEEE/ACM Transactions on Networking* (2003).

[55] KUMAR, G., DUKKIPATI, N., JANG, K., WASSEL, H., WU, X., MONTAZERI, B., WANG, Y., SPRINGBORN, K., ALFELD, C., RYAN, M., WETHERALL, D., AND VAHDAT, A. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM* (2020).

[56] LI, Y., MIAO, R., LIU, H. H., ZHUANG, Y., FENG, F., TANG, L., CAO, Z., ZHANG, M., KELLY, F., ALIZADEH, M., ET AL. Hpcc: High precision congestion control. In *ACM SIGCOMM* (2019).

[57] LIM, H., BAI, W., ZHU, Y., JUNG, Y., AND HAN, D. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021).

[58] LIM, H., KIM, J., CHO, I., JANG, K., BAI, W., AND HAN, D. Flexpass: A case for flexible credit-based transport for datacenter networks. In *Proceedings of the Eighteenth European Conference on Computer Systems* (2023), pp. 606–622.

[59] LIU, K. Online resource of Pyrrha. https://github.com/NASA-NJU/Pyrrha, 2024.

[60] LIU, K., TIAN, C., WANG, Q., ZHENG, H., YU, P., SUN, W., XU, Y., MENG, K., HAN, L., FU, J., DOU, W., AND CHEN, G. Floodgate: Taming incast in datacenter networks. CoNEXT '21.

[61] LIU, S., GHALAYINI, A., ALIZADEH, M., PRABHAKAR, B., ROSENBLUM, M., AND SIVARAMAN, A. Breaking the transience-equilibrium nexus: A new approach to datacenter packet transport. In *NSDI* (2021).

[62] LIU, V., HALPERIN, D., KRISHNAMURTHY, A., AND ANDERSON, T. F10: A fault-tolerant engineered network. In *NSDI 13* (2013), USENIX Association.

[63] LIU, V., HALPERIN, D., KRISHNAMURTHY, A., AND ANDERSON, T. F10: A Fault-Tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 399–412.

[64] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review* (2008).

[65] MITTAL, R., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., ZATS, D., ET AL. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM* (2015).

[66] MITTAL, R., SHPINER, A., PANDA, A., ZAHAVI, E., KRISHNAMURTHY, A., RATNASAMY, S., AND SHENKER, S. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 313–326.

[67] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM* (2018).

[68] NACHIONDO, T., FLICH, J., AND DUATO, J. Buffer management strategies to reduce hol blocking. *IEEE Trans. Parallel Distrib. Syst.* (2010).

[69] PAN, R., PRABHAKAR, B., AND LAXMIKANTHA, A. Qcn: Quantized congestion notification. *IEEE802 1* (2007), 52–83.

[70] PENG, Y., ZHU, Y., CHEN, Y., BAO, Y., YI, B., LAN, C., WU, C., AND GUO, C. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 16–29.

[71] PETERSON, W. W., AND BROWN, D. T. Cyclic codes for error detection. *Proceedings of the IRE* (1961).

[72] QIAN, K., XI, Y., CAO, J., GAO, J., XU, Y., GUAN, Y., FU, B., SHI, X., ZHU, F., MIAO, R., WANG, C., WANG, P., ZHANG, P., ZENG, X., RUAN, E., YAO, Z., ZHAI, E., AND CAI, D. Alibaba hpn: A data center network for large language model training. In *SIGCOMM* (2024).

[73] QURESHI, M. A., CHENG, Y., YIN, Q., FU, Q., KUMAR, G., MOSHREF, M., YAN, J., JACOBSON, V., WETHERALL, D., AND KABBANI, A. PLB: congestion signals are simple and effective for network load balancing. In *ACM SIGCOMM* (2022), pp. 207–218.

[74] RACKSOLUTIONS. How many servers does a data center have? https://www.racksolutions.com/news/blog/how-many-servers-does-a-data-center-have/, 2020.

[75] RAJASEKARAN, S., GHOBADI, M., AND AKELLA, A. CASSINI: Network-Aware job scheduling in machine learning clusters. In *NSDI 24* (Santa Clara, CA, 2024), USENIX Association.

[76] RAJASEKARAN, S., GHOBADI, M., KUMAR, G., AND AKELLA, A. Congestion control in machine learning clusters. In *HotNets* (2022).

[77] ROS-GIRALT, J., AMSEL, N., YELLAMRAJU, S., EZICK, J., LETHIN, R., JIANG, Y., FENG, A., AND TASSIULAS, L. A quantitative theory of bottleneck structures for data networks. *arXiv preprint arXiv:2210.03534* (2022).

[78] ROS-GIRALT, J., AMSEL, N., YELLAMRAJU, S., EZICK, J., LETHIN, R., JIANG, Y., FENG, A., TASSIULAS, L., WU, Z., TEH, M. Y., ET AL. Designing data center networks using bottleneck structures. In *SIGCOMM* (2021).

[79] ROSEN, E., VISWANATHAN, A., AND CALLON, R. Rfc3031: Multiprotocol label switching architecture, 2001.

[80] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2015).

[81] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating fair queueing on reconfigurable switches. NSDI'18, USENIX Association.

[82] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM* (2015).

[83] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 44–57.

[84] SONCHACK, J., AVIV, A. J., KELLER, E., AND SMITH, J. M. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–16.

[85] W. DALLY, P. C., AND DENNISON, L. Architecture of the avici terabit switch/router. *Proc. of 6th Hot Interconnects* (1998).

[86] WANG, W., MOSHREF, M., LI, Y., KUMAR, G., NG, T. E., CARDWELL, N., AND DUKKIPATI, N. Poseidon: Efficient, robust, and practical datacenter CC via deployable INT. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 255–274.

[87] XU, Y., HE, K., WANG, R., YU, M., DUFFIELD, N., WASSEL, H., ZHANG, S., POUTIEVSKI, L., ZHOU, J., AND VAHDAT, A. Hashing design in modern networks: Challenges and mitigation techniques. In *ATC* (2022).

[88] YANG, M., BABAN, A., KUGEL, V., LIBBY, J., MACKIE, S., KANANDA, S. S. R., WU, C.-H., AND GHOBADI, M. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 633–648.

[89] YU, Z., WU, J., BRAVERMAN, V., STOICA, I., AND JIN, X. Twenty years after: Hierarchical core-stateless fair queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation* (2021), pp. 29–45.

[90] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI* (2012).

[91] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI* (2012).

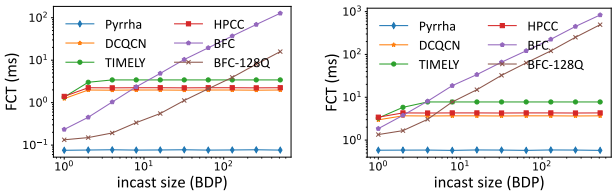
[92] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

[93] ZHANG, Z., ZHENG, H., HU, J., YU, X., QI, C., SHI, X., AND WANG, G. Hashing linearity enables relative path control in data centers. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 855–862.

[94] ZHIYU ZHANG, SHILI CHEN, R. Y. R. S. H. M. H. W. Z. C. G. F. Y. F. W. S. S. L., AND XU, Y. vpifo: Virtualized packet scheduler for programmable hierarchical scheduling in high-speed networks. In *SIGCOMM* (2024).

[95] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. In *ACM SIGCOMM* (2015).

APPENDIX



(a) Average FCT. (b) 99th-tail FCT

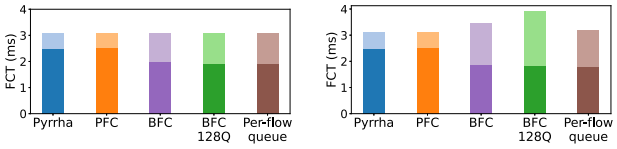
Figure 17: When the size of incast flows varies.

A Discussions

Dealing with link failures. For link failure that exactly occurs at the port of congestion roots, Pyrrha sends RESUME frames to its upstream switches to avoid unnecessary further control. For link failure on other ports, the failure is transparent to Pyrrha when source routing is used. When hash calculation is used to get flows' path, the calculation can be dynamically adjusted by introducing a rerouting table when route changes by leveraging existing rerouting protocols to propagate the rerouting information.

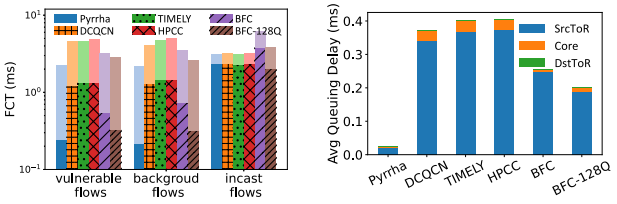
Related works. The IEEE standard group advocates Congestion Isolation (CI) [47], supporting the isolation of congested data flows within datacenter. An egress port can identify flows causing congestion and isolate them locally. The progress of the IEEE standard proved the necessity of Pyrrha from the side. However, CI can only control congested flows that already arrived at the congested port. Besides, CI uses only one dedicated queue for all congested flows, resulting in HOL blocking among congested flows passing through different congested ports.

In addition, there are several existing lines of work that go beyond flow control and congestion control. dcPIM [24] takes several RTTs to match senders and receivers before starting transmission to avoid congestion. TCD targets solving the problem that traditional flow control (*e.g.*, PFC and CBFC [17]) interferes with the congestion detection of congestion control protocols. It proposes an undetermined state by leveraging the ON-OFF pattern caused by flow control, and only the flow identified as congested reduces its rate according to congestion control. It can give more penalty to congested flows afterward but does not handle congestion that already occurs. ABM [5] is an efficient buffer-sharing scheme that leverages both spatial and temporal features of the buffer. Queue scheduling approaches target at providing approximate fair queuing [81, 89] or strict priority [12, 38, 9, 83]. Scheduling approaches do not differentiate congested flows from those uncongested thus they can not eliminate HOL blocking. CASSINI [76, 75] adjusts the communication phases of ML jobs to share the bandwidth. These works are orthogonal to Pyrrha.



(a) Web Server (incast flows). (b) Web Search (incast flows).

Figure 18: Incast flow performance.



(a) Average and 99th-tail FCT. (b) Queuing time (uncongested flows).

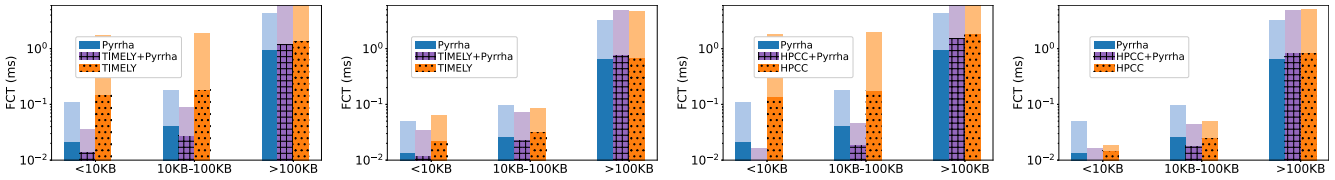
Figure 19: When facing multiple congestion roots.

B Supplemental Results

When the size of incast flows varies. Now we investigate the performance of uncongested flows when the size of incast flows varies. As shown in Figure 17, Pyrrha achieves a stable performance when the size of incast flows increases. This is expected since Pyrrha can already control the transmission of incast flows when the size of incast is smaller than one BDP (§ 6.2). The performance of BFC downgrades to a large extent with long incast flows. Because long incast flows occupy the queues continuously, HOL blocking uncongested flows sharing the same queue with them. For CC protocols, the FCT of uncongested flows becomes stable after the incast size increases to three BDPs since congestion control begins taking effect.

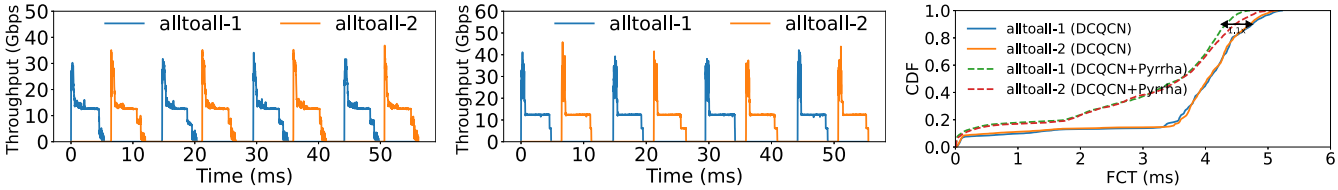
When facing multiple congestion roots. We investigate the scenario where multiple congestion roots exist. An uneven hash function is used to conduct load imbalance. ToR switches can choose among four core switches to forward, the possibility to route to the first core switch is 50%, and the other three paths share the left 50% uniformly. This is called hash polarization, which can occur in datacenter networks. Flows are generated as the incastmix scenario (§ 6.2). Hence, there are multiple congestion roots in networks, *e.g.*, load imbalance on the upstream ports of ToR switches and incast on their downstream ports.

Figure 19 shows the performance of different approaches. Pyrrha reduces the average and tail FCT of non-incast flows. Pyrrha detects the congestion on source ToR switches and sends congestion notifications to the connected source hosts. Then, senders can send other flows that do not pass the imbalanced ports to utilize the network. For congested flows passing through both load imbalance and incast ports, their transmission is well controlled. Hence, the queuing delay



(a) Web Server (vulnerable flows). (b) Web Server (uncongested flows). (c) Web Server (vulnerable flows). (d) Web Server (uncongested flows).

Figure 20: FCT performance under k=16 fat-tree.



(a) Throughput (DCQCN). (b) Throughput (DCQCN+Pyrrha). (c) FCT comparison.

Figure 21: Performance comparison under MoE workloads (interleaved phase).

of Poisson flows is quite small. For CC approaches, they can not alleviate the throughput downgrade caused by load imbalance, hence FCTs of Poisson flows are prolonged.

Supplemental results under k=16 fat-tree. Figure 20 shows the additional evaluations of Pyrrha, TIMELY, and HPCC under the k=16 fat-tree. The results are consistent with Figure 6.3 where Pyrrha is compared with DCQCN.

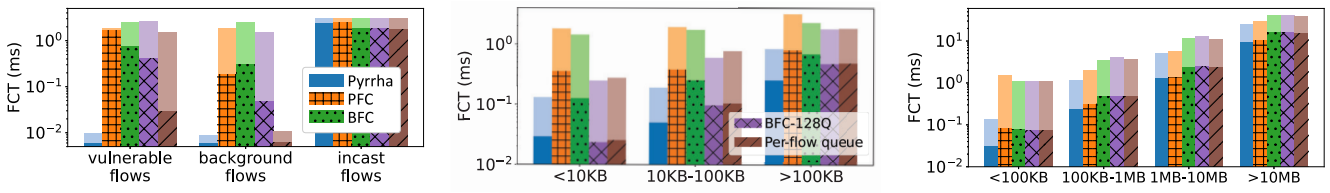
Per-flow queue scheme. Figure 22 shows the performance of flow control protocols in incastmix scenarios, including the per-flow queue scheme. For Memcached, the per-flow queue scheme can achieve as low latency as Pyrrha for background flows. While for vulnerable flows, the per-flow queue scheme can not reduce their tail latencies. Due to the fact that the per-flow queue scheme does not start to control a newly arrived congested flow until it is transmitted to the congested port. It induces a large buffer occupancy at the network bottleneck. Moreover, vulnerable flows and congested flows share the same bottleneck at core switches. When there are a large number of congested flows, vulnerable flows can only get a small fraction of bandwidth share. As a by-product, the per-flow queue scheme reduces the average FCTs of incast flows, as shown in Figure 18.

CC with no window. Figure 26 shows the performance of Pyrrha, DCQCN, and HPCC when no sending window is applied. In comparison to protocols with sending window, the same traffic pattern as Figure 12 is generated. Without the sending window, the queuing delay on source ToR is increased across all protocols, as shown in Figure 26(e). The rationale is the absence of initial transmission control allows for a greater volume of traffic to be injected into the network, leading to longer queue length and increased queuing delays at the source ToRs before CC or FC steps into. DCQCN experiences a larger degree of congestion as each flow can concurrently inject a larger volume of traffic into the network. Likewise, DCQCN+IRN suffers from a larger loss rate which results in performance degradation. In the absence of the sending window, HPCC endures a more severe performance degradation since HPCC is designed to be a window-based protocol. Benefiting from the fast reaction to network congestion, the latency of Pyrrha is slightly increased compared to that with sending window.

Performance under MoE workloads (interleaved phase). Figure 21 depicts the performance of two groups of alltoall flows with interleaved phase, *i.e.*, the two groups of traffic do not overlap with each other. With the interleaved phase, two groups of flows are slightly affected by each other. By adding Pyrrha to DCQCN, there is still performance improvement of the latency of flows, as depicted in Figure 21(c). This is because the traffic of alltoall-1/alltoall-2 itself can induce transient congestion and Pyrrha can handle it better.

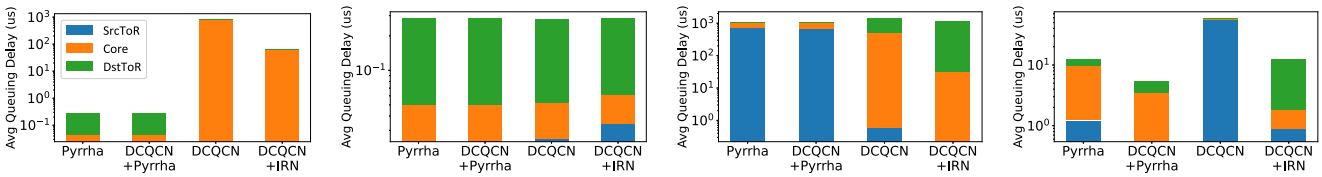
Comparing with TIMELY. We use the parameters recommended by the TIMELY authors in their paper. As shown in Figure 25, when adding Pyrrha on TIMELY, the performance improvement of TIMELY is similar to that for DCQCN.

Deep dive of the per-hop queuing delay. To fully understand how Pyrrha works, we investigate the flows' queuing time among different hops, where the experiments settings is the same as in § 6.2. Figure 23 shows the result. Queuing time is split into three parts, where SrcToR denotes the uplink ports of ToR, *i.e.*, the first hop of packets. Likewise, DstToR denotes the downlink ports of ToR, *i.e.*, the last hop of packets. The average queuing time reflects the extent of HOL blocking that uncongested flows encounter. For DCQCN, vulnerable flows are HOL blocked by incast flows at the core switches. When adding Pyrrha to DCQCN, it eliminates the queuing time for vulnerable flows significantly. For Web Server, Pyrrha also helps reduce the queuing delay of background flows since it does not cause congestion



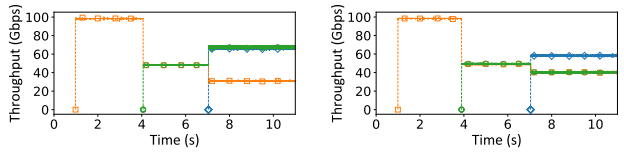
(a) Memcached (overall flows). (b) Web Server (uncongested flows). (c) Web Search (uncongested flows).

Figure 22: FCT performance of FC (per-flow queue included).



(a) Memcached (vulnerable flows). (b) Memcached (background flows). (c) Memcached (incast flows). (d) Web Server (background flows).

Figure 23: Queuing time (DCQCN) is split into three parts, *i.e.*, time spent on SrcToR, core, and DstToR.



(a) Pyrrha (congestion root varies.) (b) PFC (congestion root varies.)

Figure 24: Throughput of testbed experiments (additional).

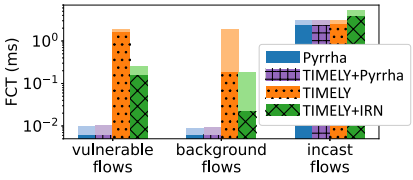


Figure 25: TIMELY: Memcached (overall flows).

spreading, and only flows causing congestion are paused. Pyrrha reacts quickly on congestion at the last hop, hence the queuing time of incast flows is mainly composed of the time spent on the srcToR and core switches.

Pyrrha is robust when congestion roots vary. We provide additional testbed results when congestion roots vary. The testbed setting is the same as in § 5.2. When flows on three hosts S2, VS, and S1 start to arrive at 1s, 4s, and 7s, respectively, the congestion root changes from the core switch to the destination ToR. Pyrrha can also converge quickly and achieve good throughput.

B.1 Parameter and Mechanism Validation

Parameter Selection. In this section, the performance of Pyrrha under different parameters is evaluated. We scan parameters under incastmix scenarios of Memcached. Figure 27 depicts the 99th-tail FCT of the vulnerable flows. K_{resume} is set to 1 when sweeping K_{pause}, and K_{pause} is set to 10 when sweeping K_{resume}. Pyrrha is relatively insensitive to different K_{pause} and K_{resume} value, which is preferred in the industry.

As shown in Figure 27(a), Pyrrha achieves the smallest tail latency when K_{pause} ranges from 2 to 35. When the value of K_{pause} is larger than 35, the tail latency increases. This is because Pyrrha reacts slowly on congestion with a large pause threshold.

As shown in Figure 27(b), the tail latency of Pyrrha remains unchanged when sweeping K_{resume}. For that, with a fixed value of K_{pause}, the timing and correctness of congestion root identification are the same, and K_{resume} only affects the queuing delay arrangement of congested flows on different hops.

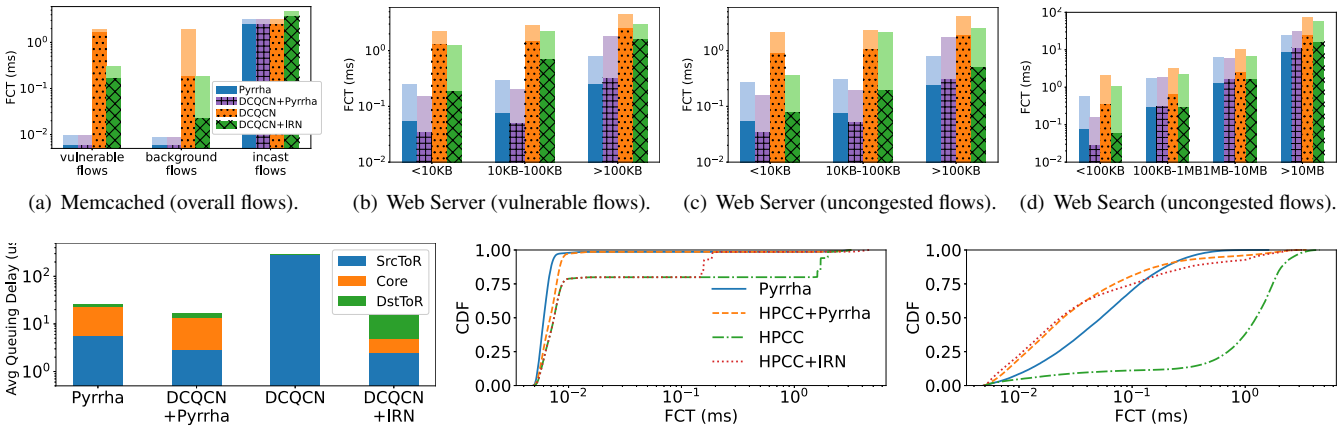
Mechanism validation. With merging, Pyrrha is insensitive to K_{pause} and K_{resume}. As shown in Figure 27(a), without merging, when K_{pause} varies (*e.g.*, 2-35), the inaccurate root detection induces queuing of vulnerable flows. While with merging, those false-positive congestion roots caused by inappropriate parameters could be corrected. The queuing delay shown in Figure 28(a) and 28(b) validates it.

Figure 27(b) demonstrates the performance when K_{resume} varies. Notice that K_{pause} here is set to 10, and the incorrect congestion detection exists. Figure 28(c) shows that the queuing delay increases when K_{resume} is set to a smaller value. This is due to a small resume threshold could be strict to resume and vulnerable flows in the IQ of the false-positive root can be blocked for a long time. With merging, different values of K_{resume} does not influence the queuing delay, as shown in Figure28(d).

C IQ Management

C.1 Methodology

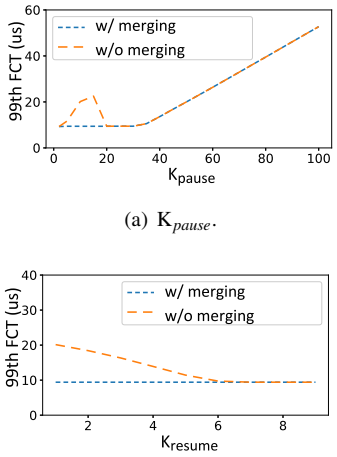
The function of HIQs (*e.g.*, hierarchical congestion matching and in-order delivery) can be achieved by leveraging single-tier IQs. Considering a two-tier HIQ matches congestion roots belonging to two different levels. We denote the single-tier IQ matching two-level congestion roots as IQ_{A_i} and IQ_{B_j}



(a) Memcached (overall flows). (b) Web Server (vulnerable flows). (c) Web Server (uncongested flows). (d) Web Search (uncongested flows).

(e) Queuing time (Web Server background). (f) HPCC: Memcached (overall flows). (g) HPCC: Web Server (uncongested flows).

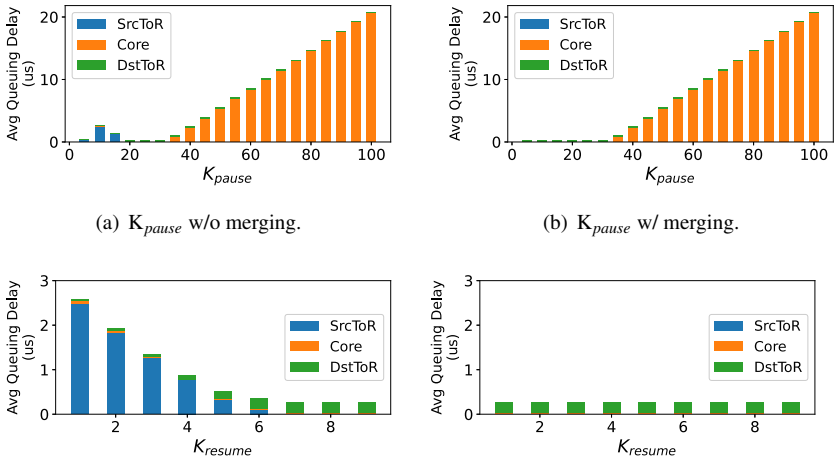
Figure 26: Performance w/o sending window.



(a) K_{pause}.

(b) K_{resume}.

Figure 27: Parameter selection.



(a) K_{pause} w/o merging. (b) K_{pause} w/ merging.

(c) K_{resume} w/o merging. (d) K_{resume} w/ merging.

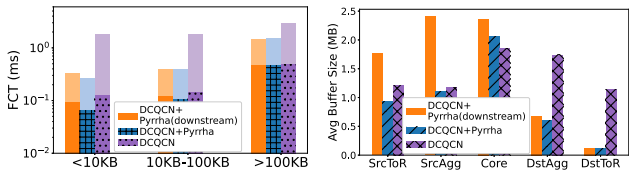
Figure 28: Queuing time variation of vulnerable flows across different value of K_{pause} and K_{resume}.

respectively and use lower case letters a_i and b_j to denote the corresponding congestion roots. For a network with two congestion roots a_1 and b_1, there can be flows matching congestion roots a_1, b_1, and $a_1 \wedge b_1$. To handle these three types of congestion flows, for HIQs, IQ_{A_1} and IQ_{B_1} can be arranged into two levels. For single-tier IQs, three dedicated IQs (*i.e.*, IQ_{A_1}, IQ_{B_2} and $IQ_{A_1 \wedge B_2}$) are needed. Generally, to isolate different congested flows as in two-tier IQs (*e.g.*, n first level IQs $IQ_{A_1}, \ldots, IQ_{A_n}$ and m second level IQs $IQ_{B_1}, \ldots, IQ_{B_m}$), the single-tier IQs need additional $n * m$ IQs (*i.e.*, $IQ_{A_1 \wedge B_1}, \ldots IQ_{A_n \wedge B_m}$).

For single-tier IQs, to ensure in-order delivery, it should be careful when a congested flow alters its chosen queue. Scenarios where a flow changes the queue it uses only occur when (i) a congestion root vanishes , or (ii) a new congestion root is detected. For the first scenario, since the IQ to be unassigned must be empty, packets of the flow are not in different queues hence in-order delivery is satisfied naturally. The second scenario is described in § 4.3, where a new congestion root b_1 is detected after a_1. Then IQ_{B_1} and $IQ_{A_1 \wedge B_1}$ are assigned to traffic passing b_1 and traffic passing both congestion roots. Following arrival packets passing through both congestion roots a_1 and b_1 alters from IQ_{A_1} to $IQ_{A_1 \wedge B_1}$. A order mark is inserted into the IQ_{A_1}, and $IQ_{A_1 \wedge B_1}$ can not be resumed until the order mark is dequeued from IQ_{A_1}. Note that IQ_{A_1} can be paused by congestion root b_1 before the order mark is dequeued due to that flows passing through both port a_1 and b_1 can be pushed into IQ_{A_1} before congestion root b_1 is detected. Given that Pyrrha detects congestion roots rapidly, packets in IQ_{A_1} can be moderate hence the order mark can be dequeued in a short time.

C.2 Experimental Verification

To reduce the complexity, we can analog two-tier HIQs which is enough for two-tier Clos-networks. For three-tier



(a) FCTs. (b) Buffer Occupancy.

Figure 29: Performance when using single-tier congestion queue to perform a two-tier HIQ.

Clos or fat-tree networks, it only matches congestion roots that two hops away from the destination host. And we leverage CC protocols to handle the congestion close to the source host.

We evaluate the performance of Pyrrha when single-tier congestion queues are used to analog the behavior of a two-tier HIQ in a three-tier topology. Figure 29 shows the performance of Pyrrha and Pyrrha handling downstream congestion roots by leveraging single-tier IQs. Results show that when comparing Pyrrha (downstream)+DCQCN with Pyrrha+DCQCN, the latency of flows smaller than 100KB is increased. But Pyrrha (downstream)+DCQCN reduces the latency of flows significantly compared to pure DCQCN.

D Pyrrha Prototype and Discussion

In this section, we present our Tofino2 prototype, with Figure 30 depicting its pipeline. We divide the pipeline into three parts, each detailed in a separate subsection (Appendix D.1 - D.3). Finally, we summarize the Tofino2 features leveraged in our prototype and examine the practicality of implementing Pyrrha on the RTC (Run-To-Completion) architecture, *e.g.*, Juniper Trio [88].

D.1 Ingress Pipeline

Packet Classifier. The packet classifier is located at the front of the ingress pipeline. The module determines the packet type according to the Ethertype field in the Ethernet header. (We use Ethertypes that are not used by the IEEE standard to mark different types of packets.) There are three types of packets: data packet, signal packet, and generated packet. Different types of packets then go through different modules.
Forward Engine. Only data packets go through the forwarding engine. The forwarding engine looks up the table deployed in advance and determines the egress port of the data packet. This module is similar to that of most commercial switches.
Congestion Root Matcher. This module contains two submodule: route calculation unit and hierarchical congestion root table. The methodology of route calculating is described in § 4.2, and the optimization to reduce the resource usage is detailed in Appendix E.3. And as described in § 5.1, the hierarchical congestion root table records the congestion roots and their states in a hierarchical manner. When a data packet passes, the module first calculates its route, then matches its route against the congestion root table hierarchically. If the packet matches one or more congestion roots in PAUSE state, the matched congested root id(s) is (are) carried in its metadata, which is stored in PHV and will be used in the following modules. When a signal packet passes, the module first calculates the location of the congestion root it indicates, then modifies the state of the corresponding congestion root in the hierarchical congestion root table.

Queue Manager. This module manages the mapping between the congestion root id and queue id. When a packet passes, the module tries to match the congestion root id it carried in metadata against mapping table. If there is no matched entry, the module assigns a new queue to this congestion root. When a queue is unassigned, the module deletes the corresponding entry.

Queue State Detector. *Ghost* is used in this module to detect the change in queue state. When Ghost is triggered by predefined thresholds (K_{pause} or K_{resume}), a Ghost packet is generated to the ingress pipeline and writes the queue length into a register in this module. When passing through this module, the data packet tries to read the value in the register corresponding to the queue in which it will be queued. Then the data packet compares the queue length it read with the previous queue length stored in another register. If the queue length exceeding K_{pause} or falling below K_{resume} for the first time, a PAUSE/RESUME packet is needed to be sent. In such case, the module marks the packet.

At the end of the ingress pipeline is the signal packet module, which contains three sub-modules corresponding to three types of packet.

Signal Packet Creator. Only data packets will pass through this module. Recalling that in the Queue State Detector module, the data packet is marked if a signal packet is needed to be sent. Then this module will check the marker and uses Pktgen to generate a new packet if the marker is valid. The *new programmable Pktgen trigger* of Tofino2 enables carrying some data plane metadata in the generated packet, which greatly expands programmability.

Signal Packet Constructor. The generated packets will pass through this module. This module will reorganize the metadata and construct the header of the signal packet.

Signal Packet Handler. This module will pause/resume the IQ based on the queue information carried in the signal packet. The dataplane pause/resume is achieved via *AFC*, a new feature of Tofino2. If the signal packet switches some flows from a working queue to another queue, the signal will be modified to a Order Mark and pushed into the origin queue. Otherwise, the signal packet is useless and dropped by the module.

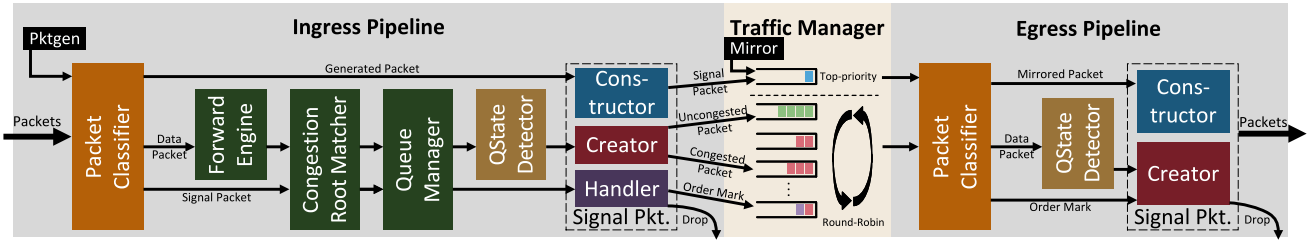


Figure 30: Pipeline of Pyrrha implementation on Tofino2

D.2 Traffic Manager

In the Traffic Manager, we configure a highest priority queue as the queue through which the signal packets pass. Other queues are configured as the same priority and scheduled in a Round-Robin manner. Among these queues, one is used as OQ, and the rest of the queues are used as IQ, which are managed by Queue Manager in Ingress Pipeline.

D.3 Egress Pipeline.

Packet Classifier. The functionality of this module is similar to the Packet Classifier in Ingress Pipeline. All packages are divided into three types: mirror packet, data packet, and Order Mark.

Queue State Detector. Unlike the Queue State Detector in Ingress Pipeline, this module uses the queue length carried by the egress packet to detect the change of queue state. This module mainly focuses on one kind of queue state: CLEAR. If an IQ is clear, it means we can unassign it when other conditions are met.

Signal Packet Creator. Because the programmable Pktgen trigger is not available in Egress Pipeline, this module uses *mirror*, another Tofino feature, to create signal packets. To deliver the signal packet to Ingress Pipeline, the module mirrors the packet to the top-priority queue of the loopback port. For the data packet with a CLEAR mark, this module just mirrors the header to create a small-size signal packet. For the Order Mark, this module mirrors the whole packet and then drops the origin packet.

Signal Packet Constructor. In this module, the mirrored packet is constructed as a signal packet. For the packet mirrored from the data packet's header, the module discards all the origin headers and reconstructs the signal header. For the packet mirrored from the Order Mark, the module only makes minor modifications.

D.4 Feasibility on RTC switches

In our Tofino2 prototype, we utilized the following features that may not be supported by all programmable switches: Ghost thread, packet trigger, and AFC (Advanced Flow Control). (i) Ghost thread is used to get queue lengths at the ingress pipeline. (ii) The packet trigger functionality can generate new packets at the end of both the ingress and egress pipelines. (iii) AFC enables the pausing and resuming of queues in the programmable data plane. We then discuss how to implement these features on other types of programmable switches that adopt RTC (Run-To-Completion) architecture, *e.g.*, Juniper Trio [88].

(i) In programmable switches, queue lengths are typically visible after packets pass through TM (Traffic Manager), *i.e.*, during the process of egress. For Tofino, the Ghost thread is required due to the hardware limitation that information can not be directly conveyed from to egress to ingress. However, in the RTC architecture, the memory is shared between ingress and egress, thus Ghost thread is naturally not required.

(ii) The packet trigger functionality is not directly supported by current RTC switches. Fortunately, through private discussion with a vendor of the RTC switch chips, we have learned that the feature can be easily implemented using background threads and will be integrated into its forthcoming RTC architecture switch chip. More precisely, the design involves dedicated background threads and associated data structures that are reserved for packet generation. These background threads are continuously active, polling the data structures for the notifications of packet generation. When a need arises to generate new packets, the necessary packet generation information is written into the corresponding data structure. Upon detection of this information, the background threads initiate the generation process accordingly.

(iii) Currently, AFC is not supported by existing RTC chips. The support for AFC in Tofino2 demonstrates the feasibility of exposing this functionality to the programmable data plane. Considering the necessity to support prevalent flow control protocols such as PFC, the capability to pause and resume queues should be a standard functionality in RTC. We are optimistic that future releases of RTC switch chips will support this essential functionality.

E Resource Usage on Tofino2

In this section, we detail the resource overhead of the Pyrrha prototype on Tofino2. Tofino2 adopts pipeline architecture, wherein the overhead of storage and computational resources is determined at compile time. It enables us to ascertain Pyrrha's resource requirements without running it in a large-

scale cluster. We separately compile three key data structures that are related to the size of the topology to report their overhead across different topological scales (Appendix E.1-E.3). Subsequently, we present the resource usage of processing logic, which remains constant irrespective of the scale of topology (Appendix E.4). Ultimately, we analyze the computational and storage resource complexities of Pyrrha and provided an exhaustive account of resource utilization under several typical network conditions. (Appendix E.5)

According to the statistics reported in megascale [49], the scale of current data centers can reach up to 10,000 nodes. Therefore, we use a fat-tree topology with 11,664 hosts as a representative case for our analysis. In addition, we investigated the largest scale of topology that Pyrrha can support theoretically, *i.e.*, approximately an order of magnitude larger than the state-of-the-art datacenter networks. The results demonstrate that Pyrrha is capable of scaling to large-scale networks. Our prototype of Pyrrha can support a typical modern data center with over 10,000 hosts. Leveraging the rapidly maturating HIQ technology [94], Pyrrha is capable of scaling up to support topology approximately an order of magnitude larger than the typical modern datacenter networks.

Our overhead analysis focuses on the ToR switches since they should track the most extensive status information for their downstream networks, thereby consuming the largest amount of resources. It is notable that the data plane storage and the forwarding plane buffers are segregated in the Tofino2. Thus Pyrrha's consumption of storage does not impact the forwarding performance.

E.1 Path Calculation

This section details how we implement hash-based path calculation for fat-tree topology in our Tofino2 prototype. The path calculation can be divided into two parts: determining available egress ports and selecting the forwarding port from available ports by hashing. Given the fact that switches in datacenter only support limited types of hash functions (*e.g.*, CRC or XOR) to conduct efficient calculation [87, 93] and use switch-specific hash seeds to avoid hash polarization. Hence, Pyrrha switch only needs to store a bitmap to indicate the type of the hash function of other switches along with their hash seeds (§ E.1.1).

In many widely deployed topologies [8, 37, 62, 82], the characteristics of up-down routing and equal paths can be leveraged to optimize the overhead of storing other switches' forwarding tables. We demonstrate that path calculation can be done within $O(n)$ resource usage where n is the number of hosts in the topology. For a fat-tree with 11,664 hosts, Pyrrha's path calculation occupies 0.44MB SRAM (*i.e.*, 1.8% of Tofino2) (§ E.1.2). If IP addresses are organized regularly as discussed in [8], the storage cost can be further reduced to 0.18% of Tofino2 (§ E.1.3). Otherwise, a Pyrrha switch needs to store forwarding tables on other switches to calculate the path, incurring extra storage overhead.

E.1.1 Hash functions used in the data center

Current switches only provide a limited number of hash functions to prevent complex hashing from impeding forwarding rates [87, 50]. To avoid traffic polarization caused by identical hash functions across different switches, it is a common practice to employ switch-specific seeds to derive switch-specific hash functions [28]. Before performing the hash function, the input data (*e.g.*, IP tuple) is XORed with the switch-specific seed, making the output of identical input data and hash function different among switches. Switches' hash seeds are not hardcoded but accessible through the control plane interface [28]. Therefore, a switch can calculate the hash outcomes of other switches by storing a limited number of hash functions and other switches' hash seeds.

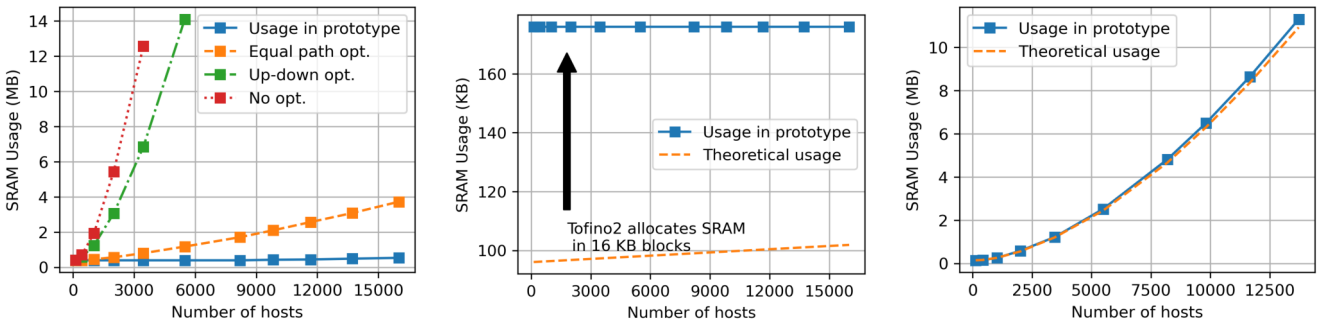
E.1.2 Path Calculation without regular addressing.

To calculate the path, Pyrrha switch needs to store the forwarding tables of other switches. In this subsection, we mainly focus on the structure and the storage overhead of forwarding tables in fat-tree topology. As shown in Figure 31(a), under two topology-relevant optimization, Pyrrha switch can calculate path in $k = 36$ (*i.e.*, 11,664 hosts) fat-tree with 0.44 MB of SRAM. The SRAM usage is acceptable in Tofino2, which has 25 MB SRAM in total. These optimizations can also be applied to other clos-based topologies.

No optimization. Without optimization, a switch needs to store all forwarding tables on other switches. Take fat-tree topology as an example; the size of the forwarding table on a switch is of the order $O(n)$, where n is the number of hosts. The number of the switch is of the order $O(n^{2/3})$. Thus, the size of extra storage usage to calculate the path is of the order $O(n^{5/3})$. Storing all forwarding tables will occupy 12.6 MB in the fat tree with a host number of 3456 (k=24).

Up-down routing optimization. In many topologies, routing adheres to the up-down principle, where packets are first forwarded upward and then downward [45]. In such topologies, switches only need to store upstream routing entries for upward hops and downstream routing entries for downward hops. In a fat-tree, a switch must store entries for all switches in its source pod, all entries for core-layer switches, and only downstream entries for switches in other $(k-1)$ pods. Though it can not reduce the complexity of the resource usage, this optimization can save storage resources on the order of $O(n^{5/3})$ in the fat tree. Storing forwarding tables will occupy 14.1 MB in the fat tree with a host number of 5,488 (k=28) under up-down routing optimization.

Equal path optimization. In many topologies, there are



(a) Forwarding table in path calculation.

(b) Hierarchical congestion root table.

(c) Queue assignment module.

Figure 31: The SRAM usage of three data structures.

multiple equal paths between nodes. Take the source aggregation switch as an example. When a packet destined for another pod arrives, the switch only needs to forward it to any one of the upward links without differentiating between the various upward links. Therefore, the edge switch only needs to store the forwarding table entries of the hosts within their own pod to determine the forwarding decisions on aggregation switches within that pod. Similarly, edge switches only need to store one routing table that maps different IPs to different pods for all core switches, as core switches are only concerned with which pod the packet needs to be forwarded to. For the aggregate switches of the other $k-1$ pods, the edge switch only needs to store $k-1$ different forwarding tables. This optimization can reduce the storage resources usage to the order of $O(n^{4/3})$. Storing forwarding tables will occupy 2.56 MB in the fat tree with a host number of 11,664 (k=36) under equal path optimization.

Full optimization (used in our prototype). Combining the two optimizations mentioned above, we can further reduce the storage resource overhead to the order of $O(n)$. The storage usage of storing forwarding tables is reduced to 0.44 MB in the fat tree with a host number of 11,664 (k=36) under full optimization. In addition, we investigated the largest scale of topology that Pyrrha can support theoretically, *i.e.*, approximately an order of magnitude larger than the state-of-the-art datacenter networks. In our prototype, storing forwarding tables will occupy 8.7 MB in the fat tree with a host number of 524288 (k=128) under full optimization.

Next, we introduce the specific content of the forwarding tables to storage and then analyze the complexity. (i) A ToR switch is required to store $K/2$ downstream entries for itself. If the destination IP of a packet is missed in the downstream entries, the switch should select a port from upward ports by hash to forward the packet. The complexity is of $O(n^{1/3})$. (ii) The ToR should store store $k^2/4$ downstream entries, for all aggregation switches within the same pod. If the destination IP is not within the pod, the aggregation should select an upward port by hash to forward the packet. The complexity is of $O(n^{2/3})$. (iii) The ToR should store $k^3/4$ entries for all core switches, containing the mappings between destination IPs and their corresponding pods, with a complexity of $O(n)$. (iv) Additionally, for the aggregation switches in each of the other $k-1$ pods, $k-1$ tables need to be stored, each with a size of $k^2/4$, to store the mappings of different destination IPs within that pod to the appropriate edge switches, with a complexity of $O(n)$. (v) For all other edge switches, tables of size $k/2$ that record the mappings between destination IPs and hosts should be stored, also with a complexity of $O(n)$. Therefore, under full optimization, the storage overhead complexity for maintaining the forwarding tables is $O(n)$.

Apply optimizations on other topologies. The two optimizations mentioned above can be applied to most clos-based topologies [37, 63, 82].

E.1.3 Path Calculation with regular addressing

In this subsection, we discuss how to conserve storage overhead for forwarding tables based on the regularity of the topology's addressing. Our discussion is based on the fat-tree topology with addressing method presented in [8]. We believe that this approach can be applicable to the vast majority of topologies with regular addressing schemes.

IP address format. The address of a host is $10.pod.switch.host$, where *pod* is the pod number, *switch* is the edge switch's position in the pod (counting from left to right) the host connected to, and *host* is the host's position among hosts connected to the switch.

Switch ID and port ID format. The result of path calculation is the switch ID and egress port ID that a packet traverses at each hop. The switch ID and port ID format given below supports fat-tree topology with k up to 128. Aligned with IP addresses, IDs are counted from left to right.

The switch ID is 16 bits in length. For edge and aggregation switches, the upper 8 bits represent the pod number, while the lower 8 bits denote the switch's position among the

Algorithm 1: Path Calculation In Fat-tree

Input: *tuple*: IP-tuple of the incoming packet.

```
1  Procedure PathCalculation(tuple)
2      if tuple.dstIP.pod == tuple.srcIP.pod then
3          if tuple.dstIP.switch == tuple.srcIP.switch then
4              edgeLocal ← true
5          else
6              aggLocal ← true
7      if edge_local == false then
8          edgePortID ← {1, Hash(tuple, localSeed)}
9          if agg_local == false then
10             aggSeed ← LookupSeed(edgePortID)
11             aggPortID ← {1, Hash(tuple, aggSeed)}
```

Source edge switch	SwitchID	It is already known
	PortID	$edgePortID$
Source aggregation switch	SwitchID	$\{srcIP.pod, 0, edgePortID[6:0]\}$
	PortID	$aggPortID$
Core switch	SwitchID	$\{0, edgePortID[6:0], 0, aggPortID[6:0]\}$
	PortID	$\{dstIP.pod\}$
Destination aggregation switch	SwitchID	$\{dstIP.pod, 0, edgePortID[6:0]\}$
	PortID	$\{1, dstIP.switch\}$
Destination edge switch	SwitchID	$\{dstIP.pod, 1, dstIP.switch\}$
	PortID	$\{dstIP.host\}$

Table 1: Obtain switch ID and port ID from existing data.

same layer switches in the pod. For core switches, the upper 8 bits indicate the position of the connected aggregation switch (noting that a core switch connects to aggregation switches at the same position across different pods). The lower 8 bits are the switch's position among core switches connected to the same aggregation switch. Switch IDs may be identical across different layers by design to conserve encoding space and reduce computational overhead. This does not cause issues since Pyrrha's congestion root table is hierarchical (§ 5.1).

The port ID is 8 bits in length. Edge switches and aggregation switches' ports can be categorized as uplink and downlink ports. The highest bit of the uplink port ID is 1, while it is 0 for a downlink port. The remaining 7 bits are the port's position within its respective uplink or downlink category. For core switches, the port ID is the connected pod number.

Path Calculation. After careful optimization, the path calculation in fat-tree only needs up to one lookup and two hash functions. We next explain how to calculate path hop-by-hop on the source edge switch. The pseudocode for path calculation is shown in Algorithm 1. Note that the logic of the P4 code is Match-Action Table (MAT) based, and the pseudocode only presents the calculation logic. After two rounds of hashing, the path for each hop can be obtained using the method described in Table 1. In both the algorithm and the tables, curly braces $\{\}$ denote the bit concatenation, and square brackets $[\,]$ denote the bit slice, which can be easily implemented by setting the key of the MAT. In the concatenation, the 0 and 1 are both 1 bit in width.

(i) Path at source edge switch. The source edge switch is aware of its own switch ID. When calculating the port ID, it is necessary to first determine whether the destination is under the same source edge switch (line 2-4). If so, there is no need to perform any calculations. Instead, the switch simply considers itself as the destination edge switch and uses $\{dstIP.host\}$ as the port ID. If not, it calculates the port ID via hashing IP-tuple and its own hash seed (line 8).

(ii) Path at source aggregation switch. The source aggregation switch id is $\{srcIP.pod, 0, edgePortID[6:0]\}$ as $edgePortID[6:0]$ is both the edge switch's up port position and the aggregation switch's position according to fat-tree's scheme. Similar to the prior case, Pyrrha needs to determine whether the destination is in the same pod (line 2-6). If so, no calculation is needed and this aggregation switch is considered as a destination aggregation switch in the following logic. If not, Pyrrha will perform the hash function. Before calculating the source aggregation switch's hash output, the source edge switch needs to look up the hash seed (line 10). The hash seed table does not need to store the hash seeds for all switches; for an edge switch, it only needs to store the hash seeds of the aggregation switches within the same pod.

(iii) Path at core switch. The switch ID of the core switch the packet will pass by is $\{0, edgePortID[6:0], 0, aggPortID[6:0]\}$. The upper 8 bits are the position of the aggregation switch it connected to and the lower 8 bits are the position of the aggregation switch port it connected to. The egress port ID at the core switch is the destination pod ID, which is available in the IP-tuple.

(iv) Path at destination aggregation switch. The switch ID of the destination aggregation switch the packet will pass by is $\{dstIP.pod, 0, edgePortID[6:0]\}$. The switch position (*i.e.*, the lower 8 bits) is the same as that of the source aggregation switch, eliminating the need for calculation. The port ID at the destination aggregation switch can be obtained from the *switch* field of the destination IP address.

(v) Path at destination edge switch. The switch ID and port ID can be directly obtained from the destination IP address.

Resource usage. Table 2 shows the resource usage of path calculation with regular addressing on the edge switch. Pyrrha use only a slight amount (0.18%) of SRAM to store hash seeds and only two hash function calls. The hash function is built-in in Tofino2, thus do not occupy any extra resource. Among those resources, the most utilized resource is the TableID, which only occupies 2.5% of the total TableID resources available in the Tofino2.

E.2 Hierarchical Congestion Root Table

We store the downstream congestion snapshot in a hierarchical congestion root table. The k^{th} table records ports that

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	40	1.56%
	Ternary Match Input xbar	0	0.00%
	VLIW Instructions	7	1.09%
	Hash Bits	60	0.77%
	Hash Calls	2	1.67%
	Stateful ALU	0	0.00%
	Logical TableID	8	2.50%
Stor.	SRAM	5	0.31%
	TCAM	0	0.00%

Table 2: Resource usage of path calculation with regular addressing.

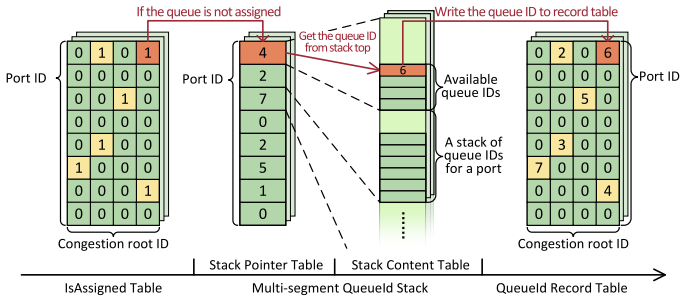


Figure 32: The components of queue assignment module with the process of queue assigning depicted.

are k hop away from the switch, whose keys are <switch-id, port-id>, and the value is a bit, denoting its congestion status. As shown in Figure 31(b), the storage usage of the congestion status snapshot is of the order $O(p)$ where p is the number of ports in the network (*i.e.*, $O(n)$ where n is the number of hosts for most topology). When the number of hosts is small, the usage of SRAM does not increase with the number of hosts. This is because Tofino2 allocates SRAM in blocks, and when the number of hosts is low, the size of each data structure fits within a single block. In our prototype, the hierarchical congestion root table occupies 176 KB in a fat tree with a host number of 11,664 (k=36) and 0.44 MB for 524,288 host (k=128).

E.3 Queue Assignment Module

The queue assignment employs a lazy update design to conserve queue resources. The components of the dynamic queue assignment module and the procedure of queue assignment are shown in Figure 32. The queue allocation module introduced in this section is applicable to both single-layer IQs and HIQs. For HIQs, the queue allocation module is also hierarchical, with each layer of queues being allocated by a corresponding layer of the queue allocation module.

Before entering queue assignment module, a packet is first processed by a congestion root matching module to determine whether it will pass through congestion roots (Appendix D.1). If it will, the packet will carry the IDs of all the congestion roots it will pass and proceed to the queue assignment module.

The packet first passes through *IsAssigned Table*, which has two dimensions: port ID and congestion root ID. This table stores bits to indicate whether a queue has been assigned to a congestion root on a port. Note that in our Tofino2 prototype, we use single-tier IQs to emulate two-level HIQs (§ C). Therefore, the congestion root ID is a concatenation of the IDs of the two potential congestion roots in two layers. If the corresponding bit in the table is 0, it indicates that the corresponding IQ has not been assigned. The packet then modifies this bit to 1 and enters the Multi-segment QueueID Stack to attempt to obtain a queue ID.

The *Multi-segment QueueID Stack* utilizes only two tables, *Stack Pointer Table* and *Stack Content Table*, yet it can independently assign queues in each port. The Stack Pointer Table is indexed by port ID and stores the top pointer of the queue ID stack corresponding to each port. The Stack Content Table holds the remaining available queue IDs for each port. When queue assignment is required, the packet first accesses the Stack Pointer Table to obtain the stack top pointer and then increments the top pointer in the table by 1. Subsequently, it reads the available queue ID from the top of the Stack Content Table. Finally, this queue ID is written into the *QueueID Record Table*, which has the same indexing as the IsAssigned table and stores the queue IDs that have already been assigned.

The pipeline architecture of Tofino2 ensures a strict order in table access among packets. After queue assignment is completed, subsequent packets that pass through the same port and congestion roots will find the corresponding bit in the IsAssigned table set to 1. Then, they will not perform any operations when passing through the Multi-segment QueueID stack. Instead, they will directly read the queue ID from the QueueID Record Table.

When unassigning a queue, the operations are basically the reverse of queue assignment. When an IQ is clear and the IQ is the leaf of the congestion tree, a signal packet is created, which carries the IQ ID to be unassigned as well as the port ID and congestion root ID. It sets the corresponding bit in the IsAssigned table to 0. Then, it increments the stack top pointer in the Stack Pointer Table by 1 and writes the unassigned IQ ID into the new stack top in the Stack Content table. Finally, it clears the corresponding record in the QueueID Record Table.

The SRAM usage of the queue assignment module is shown in Figure 31(c). The resource usage of the queue assignment module mainly stems from the IsAssigned table and the QueueId Record table. When using single-tier IQs, the congestion root ID is a concatenation of the IDs of two layers of congestion roots, which results in a space complexity of $O(n^{5/3})$ for these two tables where n is the

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	89	3.48%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	45	7.03%
	Hash Bits	684	8.22%
	Hash Calls	6	5.00%
	Stateful ALU	14	17.50%
	Logical TableID	63	19.59%
Stor.	SRAM	120	7.50%
	TCAM	31	6.46%

Table 3: Resource usage of Pyrrha's precessing logic.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	277	10.82%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	123	19.22%
	Hash Bits	1647	19.80%
	Hash Calls	36	30.00%
	Stateful ALU	61	76.25%
	Logical TableID	124	38.75%
Stor.	SRAM	712	44.50%
	TCAM	31	6.46%

Table 4: Resource usage of Pyrrha prototype under 11,664 hosts topologies with irregular addressing.

number of hosts in the topology. The SRAM usage of the queue assignment module is shown in Figure 31(c). Queue assignment module for single-tier IQs will occupy 8.64 MB in the fat tree with a host number of 11,664 (k=36), which is adequate to support the scale of typical modern topology.

Leveraging the rapidly maturating HIQ technology [94], on Pyrrha switch that support HIQ, the IsAssigned table and QueueId Record table can be decomposed hierarchically as in Appendix E.2, thereby reducing the space complexity to $O(n)$. We also implement a queue assignment module suitable for HIQ on Tofino2 to investigat the largest scale of topology that Pyrrha can support theoretically, the result shows only 2.2MB of SRAM is required to support up to 524,288 hosts.

E.4 Resource Usage of Processing Logic

In addition to the above three data structures, whose resource usage varies with the network scale, our prototype also includes some processing logic with fixed resource usage. This includes queue length detection, signal packet creating, constructing, handling, as well as queue pausing and resuming (Appendix D). The resource overhead of this processing logic is shown in Table 3. The results indicate that in our prototype, Pyrrha's resource usage is moderate and acceptable for Tofino2.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	233	9.10%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	98	15.31%
	Hash Bits	1251	15.04%
	Hash Calls	33	27.50%
	Stateful ALU	44	55.00%
	Logical TableID	104	32.50%
Stor.	SRAM	294	18.38%
	TCAM	31	6.46%

Table 5: Resource usage of Pyrrha with HIQs under 524,288 hosts topologies with regular addressing.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	250	9.77%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	89	13.91%
	Hash Bits	1426	17.14%
	Hash Calls	29	24.17%
	Stateful ALU	35	43.75%
	Logical TableID	98	30.62%
Stor.	SRAM	450	28.13%
	TCAM	31	6.46%

Table 6: Resource usage of Pyrrha with HIQs under 221,184 hosts topologies with irregular addressing.

E.5 Overall Resource Usage

In this subsection, we analyze the complexity of the overall resource utilization of Pyrrha on the Tofino2 prototype. Subsequently, we present Pyrrha's detailed resource usage across several representative network conditions. According to the statistics reported in megascale [49], the scale of current data centers can reach up to 10,000 nodes. Therefore, we use a fat-tree topology with 11,664 hosts as a representative case for our analysis. In addition, we investigated the largest scale of topology (*i.e.*, a k = 128 fat-tree with 524,288 hosts) that Pyrrha can support theoretically.

Complexity analysis. The computational resources utilized by Pyrrha are predominantly determined by the on the maximum number of network hops, with a minor correlation to the network's scale. Regarding storage resources, the overhead of Pyrrha's data structures is moderate. Denote the number of hosts in the topology as n. With irregular addressing, the storage overhead can be reduced to an order of $O(n)$ through two optimizations. With regular addressing, the storage cost for path calculation is negligible. The storage overhead of a hierarchical congestion root table is also on the order of $O(n)$. In our prototype, the storage cost for the queue assignment module is $O(n^{5/3})$. On switches that support HIQs, the storage cost for the queue allocation module is $O(n)$.

Resource		Usage	Percentage
Comp.	Exact Match Input xbar	297	11.60%
	Ternary Match Input xbar	76	5.76%
	VLIW Instructions	108	16.87%
	Hash Bits	1799	21.62%
	Hash Calls	33	27.50%
	Stateful ALU	44	55.00%
	Logical TableID	112	35.00%
Stor.	SRAM	845	52.81%
	TCAM	31	6.46%

Table 7: Resource usage of Pyrrha with HIQs under 524,288 hosts topologies with irregular addressing.

Overall, the computational resource overhead of Pyrrha remains constant across most network topologies. In our prototype, the storage overhead for Pyrrha is $O(n^{5/3})$. On switches that support HIQ, the storage overhead is $O(n)$.

Detailed usage. We present detailed reports of resource usage for Pyrrha on the Tofino2 prototype across several typical network environments. We first report the resource usage of Tofino2 prototype under the scale of a typical modern data center [49], which is shown in Table 4. In a network with 11,664 hosts (*i.e.*, fat-tree with k=36), the Stateful ALU and Logical TableID are the most utilized computational resources, accounting for 76.25% and 38.75% of the capacity, respectively. And SRAM is the most utilized storage resource, occupying 44.5% of the availale capacity.

In addition, we investigated the largest scale of topology that Pyrrha can support theoretically. Table 5 illustrates the resource usage of Pyrrha on a switch supporting HIQ within a 524k hosts network (*i.e.*, fat-tree with k=128) with regular addressing. The Stateful ALU and Logical TableID are the most utilized computational resources, accounting for 55% and 32.5% of the capacity, respectively. The storage usage is merely 18.38%. Table 6 and 7 shows Pyrrha 's resource usage on a switch supporting HIQ and the topology has 221k and 524k irregular addressed hosts (*i.e.*, fat-tree with k=96 and 128), respectively. In the case of irregular addressing, the computational overhead remains nearly identical, while the SRAM usage increases. In a network with 221k hosts, the SRAM overhead is 28.13%, and in a network with 524k hosts, the SRAM overhead rises to 52.81%, demonstrating that the storage overhead is linear.

eTran: Extensible Kernel Transport with eBPF

Zhongjie Chen[†] Qingkai Meng[‡] ChonLam Lao[¶] Yifan Liu[†]
Fengyuan Ren[†] Minlan Yu[¶] Yang Zhou[§]
[†]*Tsinghua University* [‡]*Nanjing University* [¶]*Harvard University* [§]*UC Berkeley & UC Davis*

Abstract

Evolving datacenters with diverse application demands are driving network transport designs. However, few have successfully landed in the widely-used kernel networking stack to benefit broader users, and they take multiple years. We present eTran, a system that makes kernel transport extensible to implement and customize diverse transport designs agilely. To achieve this, eTran leverages and extends eBPF-based techniques to customize the kernel to support complex transport functionalities safely. Meanwhile, eTran carefully absorbs user-space transport techniques for performance gain without sacrificing robust protection. We implement TCP (with DCTCP congestion control) and Homa under eTran, and achieve up to 4.8×/1.8× higher throughput with 3.7×/7.5× lower latency compared to existing kernel implementation.

1 Introduction

Datacenter applications are always evolving and becoming increasingly diverse. For example, recent microservice applications heavily rely on Remote Procedure Calls (RPCs) to communicate among loosely-connected components; storage applications require bulk data transfer and replications for fault tolerance; popular ML applications employ collective communication primitives such as AllReduce to exchange large volumes of model weights and gradients.

To better support these diverse applications, datacenter researchers have proposed many network transport designs. For example, the Homa transport [50, 55] with SRPT (Shortest Remaining Processing Time) and receiver-driven designs target low-latency RPCs; The DCTCP congestion control (CC) protocol [2] enhances the TCP transport for various datacenter workloads, including bulk transfers; The MLT transport [74] is designed specifically for distributed DNN training. There are many more transport designs for emerging datacenter workloads [14, 26, 30, 44, 45, 49, 51, 82].

However, very few transport designs have landed in the widely-used Linux networking stack to benefit broader users. For example, DCTCP took 4 years to land [18], while MPTCP took nearly a decade [51]; Homa (2018) [50] has been an author-maintained kernel module since 2021 but remains unmerged in mainline. To the best of our knowledge, no transport protocol has landed on the kernel since DCTCP. The key reason for the above status quo is that kernel transports take extremely heavy efforts to develop and maintain, and often give unsatisfactory performance. As a result, it cannot keep up with the fast evolvement and high performance requirements of datacenter transports.

People nowadays mainly resort to kernel-bypass techniques by implementing transport directly in the user space or in hardware. However, this approach largely sacrifices kernel-provided protection and manageability [4, 57, 61, 80, 81], not friendly to public cloud users. For instance, integrating transport within the application process [38, 78] risks malicious manipulation or unintended interference with transport events (e.g., acknowledgments, timeout). Alternatively, running transport in a separate process [41, 47] or on hardware [3, 82] introduces new infrastructure and tools, incurring high management burdens such as telemetry and debugging.

Instead, this paper takes the kernel approach and tries to tackle the following challenging problem: *how to make the kernel transport extensible to enable agile customization, while achieving kernel safety, strong protection, and high performance?* This is challenging because 1) customizability might expose new attack surfaces in the kernel, creating safety challenges; and 2) strong protection requires putting the entire transport states inside the kernel (e.g., the kernel TCP stack), which is usually contrary to high performance due to high kernel overhead (e.g., user-kernel crossing), and makes customization hard due to fixed kernel implementation.

Fortunately, we find that the growing trend of safe kernel extensions via the increasingly mature eBPF technique provides promising opportunities. This technique allows applications to inject statically-verified code into kernel eBPF hooks to change how the kernel reacts to certain events, e.g., incoming packets. Note that this is different from writing unsafe kernel modules that might crash the whole kernel. A bunch of recent eBPF verification work [8, 46, 69, 73, 76] has largely strengthened the safety of eBPF programs running inside the kernel. Recent advances in eBPF functionalities like XDP programmable network data path [58], dynamic memory allocation [20], exception handling [40], and exposing more kernel functions [42] make eBPF more and more capable.

We introduce eTran (**e**xtensible kernel **Tran**sport), a system for agilely customizing kernel transports. eTran achieves agile customizability and kernel safety by 1) leveraging existing eBPF infrastructure such as built-in data structures (eBPF maps), BPF timer, and XDP for fast packet IO, and 2) extending it with new eBPF hooks and maps to support complex transport functionalities while conforming to the strict eBPF verifier for safety. eTran allows safely offloading full trans-

port states and performance-critical operations into the kernel, achieving strong protection. For example, it supports packet acknowledging, flow pacing, fast retransmission, and more in eBPF. In terms of performance, eTran shows similarities with user-space transports for the following design choices: direct packet delivery to user-space without kernel-to-user copies; lightweight packet buffer management and syscall-free IO batching; streamlined transport implementation but preserving genericity. However, transport processing under eTran remains in the kernel, fully isolated from applications. Crucially, transport states are safeguarded by kernel eBPF, preventing direct access from untrusted libraries.

Under eTran, we implemented two representative and vastly different transports, the sender-driven TCP (with DCTCP congestion control) and the receiver-driven Homa, and show great performance gains. For TCP, eTran version achieves 2.4-4.8× higher throughput and 3.2-3.7× lower latency than kernel-native TCP, while being only 1.6-1.8× slower than a state-of-the-art microkernel-inspired kernel-bypass solution with comparable latency. For Homa, eTran version outperforms the kernel module implementation by 1.7-1.8× higher throughput and 3.9-7.5× lower latency for RPCs. We further showcase that eTran supports multi-tenancy, coexisting transport protocols and flexible traffic management.

2 Background and Motivation

2.1 Network Transports

Diverse transports for applications. Modern datacenters have developed various transports to better meet their workloads with two main categories as follows:

- *Sender-driven transports*, such as DCTCP [2], Swift [44], and more [1,45,49,72,75,82], heuristically control the sending rate on the sender side. These protocols are generally beneficial for elephant flows [32] or traffic patterns with low entropy. They are commonly used in machine learning training and storage applications [25,48,60].
- *Receiver-driven transports*, such as NDP [30], Homa [50, 55], and more [14, 26], assume last-hop congestion and proactively control the sending rate by allocating credits from receivers to senders [14,26,30,50]. These protocols prevent queuing and improve tail performance, especially for small flows that dominate RPC applications [15,50].

While datacenter transports differ in how to control the sending rate (e.g., different congestion signals), their implementation share the following common components:

- *Segmentation and reassembly* involve chunking data into packets for sending and reassembling them upon receiving.
- *Congestion control* explores the right sending rate to utilize the available network bandwidth while avoiding congestion. To adjust the rate, transports maintain various congestion states, such as RTT. To enforce rate control, both sender-driven and receiver-driven transports require pacing engines based on target rate, credit availability, or window size.
- *Loss recovery* detects packet loss and triggers retransmission. Transports usually leverage multiple ACK responses to infer packet loss and trigger fast retransmission. When this mechanism is insufficient, timeout-based retransmission serves as the last line of defense for reliable transfer.

Protection for transports. Transport implementation requires protection or isolation from the user-space applications, typically through kernel-based networking stacks. Here, we assume the following threat model that is common in public clouds: applications using transports might be malicious; the eBPF programs written by applications might be malicious but can be detected and prevented by the existing kernel verifier, recent advanced runtime verifier [40], or others (see §8); the kernel and eBPF subsystems are trusted, e.g., through verification techniques like [69]. Without protection, application bugs, crashes, or malicious behaviors could corrupt the transport states and impact other colocating applications [5]. Kernel-based protection has its additional benefit where the networking stacks can firewall malicious applications, enforce access control, and implement various rate-limiting policies [4,57]. On the other hand, eBPF, as a kernel-native mechanism, naturally provides kernel-based protection against user-space applications, which motivates us to explore the possibility of running transports inside eBPF.

2.2 eBPF Basics

eBPF (extended Berkeley Packet Filter) is a promising technique that allows users to run customized, verified programs at specific kernel hooks. It uses eBPF maps and eBPF helper functions to store and exchange various data types between kernel and user space. Overall, eBPF enables safe and efficient extensions of kernel capabilities without modifying kernel code or loading unverified (thus unsafe) kernel modules.

Increasingly capable eBPF. The Linux kernel community is advancing the eBPF subsystem to handle more complex logic. For instance, `kfunc` complements eBPF helper functions by exposing kernel functions and data structures with restrictions; `dynptr`, added in v5.19, allows pointing to dynamically-sized memory; a dynamic memory allocator was added in v6.1; and advanced structures like rbtree (Red-Black Tree) were added in v6.3. Although eBPF still has limitations (e.g., no floating-point arithmetic to avoid non-determinism [53]), its growing capabilities enable safe kernel transport customization.

Fast packet processing with eBPF. XDP (eXpress Data Path) [58] is an eBPF hook in the Linux kernel for fast packet processing; AF_XDP is a specialized socket family built on XDP for fast packet processing in the user space. Packets processed by XDP can be dropped, forwarded back, passed to kernel networking stacks, or redirected to AF_XDP sockets based on the return code specified by the eBPF program.

AF_XDP uses four rings for packet IO: TX/RX rings per socket and Fill/Comp rings per NIC queue. The application provides buffers via the Fill ring for the kernel to receive packets, which are then placed in the RX ring by the kernel.

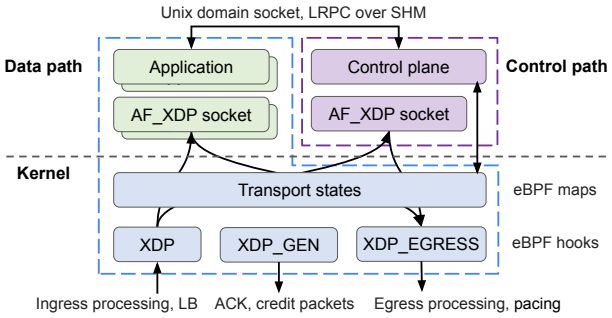


Figure 1: Overall architecture of eTran.

It submits buffers to the TX ring for transmission, and the kernel returns transmitted buffers to the Comp ring for reuse. All operations are asynchronous and optimized for syscall-free. Kernel operations of AF_XDP run in the NAPI context, which is Linux's event-driven packet processing mechanism. AF_XDP uses UMEM, a pre-allocated memory pool of fixed-size chunks, as the packet buffer pool. Currently, an AF_XDP socket can only bind to one NIC queue and UMEM while a NIC queue or UMEM can bind to multiple AF_XDP sockets.

3 eTran Design

eTran has the following four design goals:

- **Agile customizability**: eTran should allow agilely customizing diverse transports and their full functionalities.
- **Kernel safety**: eTran should prevent eBPF programs from tampering with or even crashing the kernel.
- **Strong protection**: eTran aims to maintain *all* transport states inside the kernel with corresponding state operations in eBPF, protecting against untrusted applications.
- **High performance**: eTran aims to achieve much higher performance than existing kernel transports and comparable performance to kernel-bypass ones.

Figure 1 shows the high-level architecture of eTran, which separates any transport into a control path and a data path. The control path handles non-performance-critical eBPF programs attaching/detaching, AF_XDP socket and UMEM creation/destroying, connection setup/teardown, and complex tasks that are unsuitable for eBPF (§3.1). It runs inside a root-privilege daemon, and multiple control paths from different transports can share the same daemon. Through this control path separation, eTran allows non-privilege applications to attach eBPF programs to the kernel through the control path daemon.

The data path handles performance-critical transport operations (e.g., packet reassembly, ACK/credit response, flow pacing) with streamlined implementation for high performance; it maintains all transport states (with state operations) inside the kernel for strong protection (§3.2). The data path runs across the kernel and user space: the kernel part runs in eBPF for kernel safety with direct access to transport states; the user-space part runs through untrusted libraries that cannot directly access transport states. Both parts can be agilely customized.

The data path and control path communicate through Unix domain socket and efficient LRPC [6, 24, 54] over shared memory. Additionally, eTran features flexible user-space IO by layering atop of the AF_XDP sockets (§3.3), multi-tenancy support, and flexible traffic management (§3.4).

3.1 eTran Control Path

Managing eBPF program. Applications submit transport-specific eBPF programs to our control path daemon to attach to the kernel, as such attaching requires root privilege. The daemon will first do a series of checks to make sure these programs submitted by different applications won't conflict with each other, e.g., overlapping ports or eBPF maps. Then, it will try to attach the eBPF programs to the hooks and create entries in `BPF_TYPE_PROG_ARRAY` for them, which will go through the kernel verification for liveness and safety.

Managing AF_XDP socket and UMEM. The control path daemon creates all AF_XDP sockets on behalf of applications and then transfers the file descriptors to applications through the Unix domain socket. This allows applications to use AF_XDP without privileges (e.g., `sudo`). Similarly, the daemon is also responsible for creating an individual UMEM for each application, mapping it into its address space. All sockets for a single application share the same UMEM.

Managing transport connections. The control path daemon handles connection management for connection-oriented transports such as TCP. Applications post control operations (i.e., `open/connect/listen/accept/close`) via LRPC channels; then the daemon intercepts the connection-related packets through its own AF_XDP socket and handles connection setup/teardown, then returns results to applications. eTran decides to put connection management into the control path due to the complexity of handshaking and timeouts, which are challenging for eBPF while being non-critical to transport performance. A similar design is adopted by TAS [41].

Managing CC and loss recovery. eTran decides to put few challenging-to-offload functionalities to the control path, including running advanced CC algorithms (e.g., involving floating-point arithmetic that is prohibited in eBPF) and handling severe loss recovery triggered by timeouts (not TCP fast retransmission). We will elaborate on this in §4.

3.2 eTran Data Path: Kernel Offloading

Running transport protocols and maintaining transport states inside the constrained eBPF environments requires enhancing the existing eBPF subsystem. We use the following three example issues to motivate the necessity of such enhancement:

① Many transports update states when packets get sent out, e.g., Homa decreasing the remaining bytes, but the XDP hook only works on ingress packets.

② Transports usually generate ACK and credit packets to coordinate senders and receivers, but the XDP hook does not support packet generation inside the kernel.

③ Almost all transports require a pacing mechanism to regulate when to send packets, but the XDP hook does not

support any buffering mechanism to defer packet sending.

The challenge is how to efficiently enhance the eBPF subsystem while not incurring destructive chances to eBPF, e.g., keeping similar semantics on the introduced changes, and reusing the existing kernel verifier and data structures. To address this challenge, eTran provides two new eBPF hooks, `XDP_EGRESS` and `XDP_GEN` (§3.2.1), and one new eBPF map, `BPF_MAP_TYPE_PKT_QUEUE` (§3.2.2).

3.2.1 New eBPF Hooks for Egress and Pktgen

XDP_EGRESS is a new eBPF hook that eTran designs to allow eBPF to efficiently customize egress packet processing, addressing the issue ①. eBPF programs attached to this hook accept an egress packet context as the input parameter and specify a return code to determine how this packet gets transmitted. We make the hook transparent to the NIC driver layer to avoid per-vendor code modifications. To achieve this, we place the hook in the vendor-agnostic AF_XDP kernel code, specifically, the `xsk_tx_peek_desc` function. This function fetches packet descriptors from AF_XDP TX rings and delivers them to the NIC drivers, finishing packet transmissions.

We also aim to reuse as many existing eBPF infrastructure as possible. First, `XDP_EGRESS` hook uses the same packet context (i.e., `struct xdp_md`) as XDP, facilitating the use of eBPF helpers designed for XDP. We add a new `umem_id` field at the end of the struct to indicate the UMEM of each packet, preventing unauthorized access to resources like connections and ports. For XDP programs, this field is invalid and inaccessible (see §4.1 for more detail). Second, the `XDP_EGRESS` hook shares the same return codes and actions as XDP:

- XDP_TX: Transmit the packet immediately.
- XDP_REDIRECT: Redirect the packet to a special map for buffering or another AF_XDP socket.
- XDP_DROP: Drop the packet.

XDP_GEN is another new eBPF hook that eTran designs to allow eBPF to efficiently generate packets. These packets can be customized based on the transport, e.g., ACK and credit packets, addressing the issue ②. eBPF programs attached to this hook also use the same packet context as XDP, and make packet-generating decisions based on the states from eBPF maps. Figure 2 shows how this hook works : first, the eBPF program at the XDP hook pushes necessary metadata (e.g., `seq` and `ack` in TCP) into a per-CPU eBPF queue, if it needs to generate an ACK packet; then the eBPF program at the `XDP_GEN` hook will pop any such metadata from the queue, fill it into a new packet frame, and send the packet out. eTran places the `XDP_GEN` hook in the `xdp_do_flush` function that is called at the end of a NAPI poll to generate packets.

A strawman approach dynamically allocates packet frames on-demand but incurs high overhead due to complicated and slow kernel memory management. Instead, we leverage batching to amortize the overhead: we pre-allocate enough frames and call the `XDP_GEN`-attached eBPF program in a finite loop whose budget is the same as NAPI; in each iteration, the eBPF

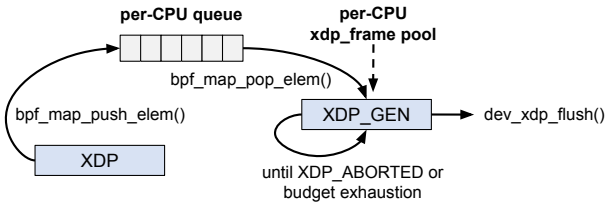


Figure 2: Coordination between XDP and XDP_GEN.

program may use a pre-allocated packet frame to generate a packet. We further leverage the *page_pool* allocator, an efficient memory allocator optimized for XDP, to do the frame pre-allocations. Doing batched packet generations essentially coalesces ACK or credit packets and sends them in batches for high performance; a similar coalescing mechanism has been leveraged in the transport implementations from Google Falcon [28], TAS [41], Caladan [24], and eRPC [38].

Our `XDP_GEN` hook supports three actions:

- XDP_TX: Transmit the ACK/credit packet immediately.
- XDP_DROP: Drop the ACK/credit packet.
- XDP_ABORTED: No more ACK/credit packets are generated, and the finite loop will be terminated.

3.2.2 New eBPF Map for Pacing

To address the issue ③, eTran designs a new eBPF map `PKT_QUEUE` (i.e., `BPF_MAP_TYPE_PKT_QUEUE`), serving as the backbone data structure for various pacing mechanisms. A `PKT_QUEUE` contains multiple buckets, each with slots storing pointers to `xdp_frames`, which describe packet metadata (e.g., packet address). We leverage the eBPF `dynptr` feature (§2.2) to allow eBPF programs to safely access these `xdp_frames` by the pointers stored in `PKT_QUEUE` buckets (i.e., by functions `bpf_dynptr_from_xdp_frame` and `bpf_dynptr_slice_rd/rdwr`).

eTran's `PKT_QUEUE` supports enqueueing and dequeueing `xdp_frame` to/from a specified bucket. Depending on the pacing mechanism, the bucket index is determined based on the packet transmission timestamp (i.e., `tx_timestamp`) or explicitly specified; we will cover more detail at the end of this subsection. We further wrap a series of eBPF kfuncs (§2.2) to help eBPF programs manipulate `xdp_frame` pointers. For example, an existing kernel function `dev_xdp_enqueue` is wrapped into a kfunc to transmit packets through NIC queues.

In a quick summary, our `PKT_QUEUE` allows safely tracking packets deferred for transmission, and its associated kfuncs allow transmitting packets from eBPF programs. Now, the next question is: *how to asynchronously execute an eBPF program (that transmits packets)?*

BPF timers as triggers. To answer this question, eTran's approach is leveraging the mature "callback" infrastructure of BPF timers [67]. The original BPF timer works as follows: eBPF programs call `bpf_timer_init` to register an eBPF callback function, then use `bpf_timer_start` to start the timer with input parameters of the desired timeout value and some flags; when the timeout is reached, the eBPF callback

function will be executed asynchronously. Instead of using the timeout functionality of BPF timers, eTran leverages their callback registration and asynchronous execution abilities. Concretely, we extend BPF timers with two new modes by adding new flags to the `bpf_timer_start` function:

- In the first mode, this function will enqueue the callback function registered with this timer into a per-CPU backlog queue and schedule `NETTX_SOFTIRQ`. The softirq context at the same CPU would dequeue and execute callback functions based on the transport-specific pacing mechanism.
- In the second mode, the function enqueues the callback into the backlog queue of a dedicated kthread created at kernel boot, and wakes it up. The thread will dequeue and execute callbacks based on the specific pacing mechanism.

The first mode triggers per-CPU callback executionn (common for most protocols), while the second mode suits transport protocols that require global scheduling such as Homa. Both modes can be used together if callbacks properly synchronize shared states. In our implementation, TCP uses the first mode, while Homa uses both modes.

The remaining question is: *how to support various pacing mechanisms used in different transports?* We classify pacing mechanisms into several categories and discuss individually.

Rate-based pacing. This is the most widely-used pacing mechanism as it effectively prevents bursts and reduces congestion spikes [28, 44]. eTran leverages `PKT_QUEUE` to implement a scalable and CPU-efficient timing wheel, following the state-of-the-art design Carousel [62]. Specifically, each `PKT_QUEUE` bucket serves as a timing wheel slot, indicating when stored packets should be transmitted. For details on eTran's `tx_timestamp` mapping to buckets and timing wheel operation, see the Carousel paper [62]. Note that eTran also supports complementing rate-based pacing with window boundaries. Our TCP implementation adopts this pacing.

Credit-based pacing. This is widely used in receiver-driven transport protocols such as Homa. Take Homa as an example: when an RPC cannot send packets because of insufficient credits, Homa puts the RPC packets into a pacing engine and waits for the credits to replenish. To support credit-based pacing, eTran lets each `PKT_QUEUE` bucket store packets belonging to a Homa RPC; when the credit of a RPC is replenished, eTran will transmit packets in the corresponding bucket.

Out-of-order completion for AF_XDP. For the existing AF_XDP, packets are always completed (i.e., transmitted) in the order of being pushed into the TX ring, and its buffer recycling mechanism only supports in-order completion. However, with pacing introduced in `XDP_EGRESS`, the transmission of some packets will be delayed, causing out-of-order completion. To address this, we enhance the AF_XDP packet buffer management with minor code changes. See §5 for more detail.

3.3 eTran Data Path: User-Space IOs

eTran carefully made design choices inspired by user-space transports for IOs, which are critical for performance gain over

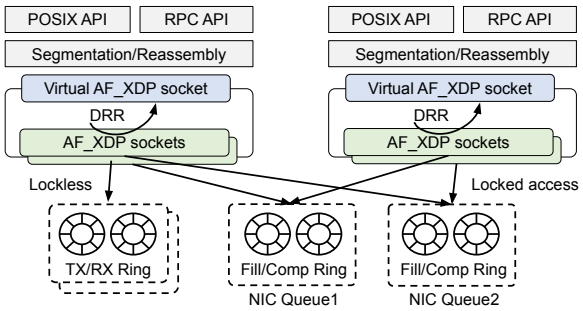


Figure 3: User-space packet IOs in eTran.

kernel-native transports. It offloads core transport operations to the kernel via eBPF and delivers data packets to the user space via AF_XDP sockets. A thin and untrusted user-space library reassembles raw data packets into transport-specific abstractions such as TCP flows or Homa RPCs and versa visa. However, we note that the untrusted library cannot directly access any transport state in the kernel, as it is within the non-privilege applications. Additionally, the library cannot access another application's UMEM (§3.1). This prevents untrusted libraries from corrupting kernel states or disrupting other applications, ensuring the transport stack remains protected.

Virtual AF_XDP socket. Figure 3 shows how our user-space library transforms AF_XDP packets into application-facing flows or RPCs. eTran utilizes multiple NIC queues, allowing different application threads to do packet IOs independently for high performance. Ideally, each application thread owns an AF_XDP socket accessing multiple NIC queues for higher parallelism and lower head-of-line blocking [13,37]; however, AF_XDP has an inherent constraint that one AF_XDP socket can only be bound to one NIC queue. To address this constraint, eTran proposes a *virtual AF_XDP socket* managing multiple real sockets (each bound to a NIC queue) via `epoll`.

Atop the virtual AF_XDP socket, eTran provides two types of APIs to fulfill the needs of different applications: 1) POSIX APIs with standard socket operations like `read` and `write`, and 2) event-driven RPC APIs with continuation callbacks like eRPC [38]. Applications using standard socket APIs can switch to eTran by LD_PRELOAD without any code change.

However, using virtual AF_XDP sockets also introduces new challenges on scheduling and thread synchronization.

Scheduling challenge. Packet arrival at a virtual AF_XDP socket indicates that packets have arrived at one or more real AF_XDP sockets. eTran must take care of how to pull packets from these real sockets in a load-balanced manner, especially to avoid starvation. To this end, eTran employs Deficit Round Robin (DRR) [66] scheduling to achieve efficient and fair packet processing among multiple sockets. When sending packets to real AF_XDP sockets, eTran operates similarly by enhancing the kernel `xsk_tx_peek_desc` function.

Synchronization challenge. All AF_XDP rings (i.e., TX/RX and Fill/Comp rings) are SPSC (single-producer, single-consumer) lockless rings designed for high-performance, run-to-completion packet processing. The TX/RX rings are per-

AF_XDP socket, therefore application and NAPI operations on them are still lockless. The Fill/Comp rings, however, are per-NIC queue. Multiple application threads can access them concurrently. A user-space spinlock is used to protect each Fill/Comp ring pair for correct concurrent access among application threads. No locking is needed for NAPI operations, as each NIC queue is handled by a single NAPI context.

Performance advantages of eTran. Compared to existing kernel transports such as Linux TCP and Homa kernel module, eTran has the following performance advantages: 1) efficient buffer management and syscall-free IO batching, 2) streamlined implementation without complicated kernel socket and (thus) file system, and 3) decoupled kernel transport processing[1], which reduces context switch [81] and improves instruction execution with better cache efficiency [41].

3.4 Multi-Tenancy and Traffic Management

eTran also offers software-based multi-tenancy and traffic management as a complement to hardware mechanisms (e.g., SR-IOV [17]). For different applications using eTran, the control path daemon isolates kernel resources (e.g., eBPF maps) by using different namespaces and assigns different NIC queues to different applications. This design prevents malicious applications from accessing others' IO buffers and enables operators to schedule problematic NAPIs on other CPUs to mitigate interruptions. Allocating at least one NIC queue per application is feasible, as modern NICs support hundreds to thousands of queues [68]. For applications not using eTran, XDP/AF_XDP naturally co-exists with other kernel built-in transports. We will demonstrate that eTran can operate seamlessly with kernel built-in transports in §6.3.

eTran uniquely excels in flexible traffic management as all packets traverse eBPF hooks. It natively supports traffic management such as traffic monitoring, access control, and rate limiting. eTran abstracts these tasks into *modular* eBPF programs, linked via `bpf_tail_call` that allows chaining multiple programs on a single hook. Operators can flexibly manage eBPF programs through the control path daemon.

4 Case Studies

This section takes two representative transports as examples to demonstrate how eTran enables implementing and customizing transport protocols in eBPF with high performance. The first transport is the widely-used TCP with DCTCP congestion control, which is a sender-driven and connection-orient protocol (§4.1). The second transport is Homa, which is a receiver-driven and connection-less protocol for datacenter RPCs (§4.2). These two transports represent a wide range of transport designs in literature [14, 26, 30, 44, 45, 49, 82]. We discuss more transport implementations in Appendix A. While eTran currently targets datacenter protocols pursuing performance, we believe it can be extended to non-datacenter scenarios such as wide-area networks. This would require more designs to handle more complex packet reordering and security mechanisms such as TLS [34] in eBPF.

[1]For Linux, the sender-side transport processing is coupled to the application context in kernel space when calling `write` or `send`.

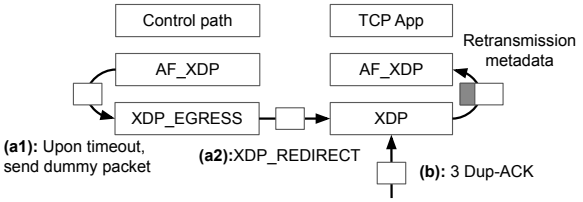


Figure 4: TCP retransmission in eTran. **(a)** is the retransmission triggered by the control path after a timeout, and **(b)** is the fast retransmission triggered by three duplicated ACKs.

4.1 TCP under eTran

TCP control path. As briefed in §3.1, the control path daemon manages TCP connections, congestion control (CC), and severe loss recovery when timeout happens. The daemon uses its own AF_XDP sockets to communicate with remote peers for the three-way handshake to establish TCP connections. It then installs TCP connection states and CC states into eBPF maps. Separating CC states into the control path is inspired by the CCP architecture [52] with two motivations: 1) eBPF does not have mature support for floating point computation that commonly appears in TCP CC algorithms, and 2) CCP has shown that separating CC from the datapath still gives similar CC behaviors and accuracy as in datapath. We defer the description of loss recovery to the end of this subsection, as it interacts closely with the data path.

eTran maintains the connection states in an eBPF hashmap while CC states are in an eBPF array. Storing CC states in an array allows eTran to `mmap` it into the daemon's address space through the `BPF_F_MMAPABLE` feature; this avoids the syscall overhead when the daemon updates CC states. To help eBPF programs locate the CC state, each connection state contains the index of the corresponding CC state in the array.

TCP data path. The eBPF program at the XDP hook receives both TCP control packets and data packets: it redirects control packets such as `SYN` and `SYN-ACK` to the control path AF_XDP; for data packets, it uses four-tuple as the key to look up TCP connection state in the aforementioned eBPF hashmap. The eBPF program then verifies the `ack` and `seq` values of data packets and updates TCP windows if valid. For common cases where `seq` is in order, packets are delivered to the virtual AF_XDP sockets in the user-space transport library (see §3.3). For out-of-order `seq`, eTran implements an out-of-order receiving mechanism similar to TAS [41].

Figure 4 shows eTran's workflow for loss recovery, including fast retransmission and timeout-triggered retransmission. To handle fast retransmission, once the eBPF program at the XDP hook detects three duplicate ACKs, it will rollback the transport state, piggyback the retransmission information (i.e., the rollback pointer) in the ACK packet, and deliver the ACK packet to user-space transport library. The library

will retransmit packets accordingly. The piggybacked retransmission information is stored in the headroom of the packet frame, enabling efficient coordination between the eBPF program and the user-space transport library. To handle timeout-triggered retransmission, once the control path daemon detects such timeout, it will send a dummy packet going through the `XDP_EGRESS` hook to trigger the fast retransmission logic of the eBPF program and then get dropped. This avoids locking operations between the data path and the control path.

The eBPF program at the `XDP_EGRESS` hook processes every packet transmitted by the AF_XDP sockets. It fills packets with the right TCP header and IP header based on the connection state maintained in eBPF maps and the right Ethernet header by looking up the kernel FIB table with `bpf_fib_lookup`. To reduce the cost of looking up the FIB table for every packet, eTran uses an eBPF array to cache results. When the flow control window and the CC window both permit, the `XDP_EGRESS` eBPF program will transmit the packet directly with the action of `XDP_TX`; otherwise, it will enqueue the packet into our `PKT_QUEUE` with the action of `XDP_REDIRECT`. Later, when both windows permit, the queued packets will be transmitted asynchronously (see §3.2.2). As briefed in §3.2.1, this eBPF program also prevents misbehaved applications from modifying connection states that do not belong to them. For example, it will check if the `umem_id` in the packet context matches the one maintained in the connection state and drop the packets if not matched.

4.2 Homa under eTran

Homa control path. Similar to TCP, Homa under eTran lets the control path daemon manage its socket creation, binding, and closing. Different from TCP, Homa runs CC in the data path via eBPF (rather than in the control path) with CC states maintained in eBPF hashmaps. There are two reasons for such a difference: 1) Homa CC or its receiver-driven credit generation is in the critical path of data packet transmission (in contract to TCP sender-driven CC that regulates rate), thus having huge performance penalty if running on a separate control path; and 2) Homa CC is relatively simple in terms of compute operations (e.g., no floating points), friendly to eBPF offloads. Since Homa is connection-less via short-lived RPCs, eTran also maintains its RPC states in the data path for efficient creation/destruction (with eBPF hashmaps).

The Homa control path detects RPC timeout and handles it by sending `RESEND` packets for retransmission. The daemon's timer thread periodically wakes up and scans RPC states in the kernel eBPF hashmap for detecting timeouts; it uses `bpf_map_lookup_batch` to reduce syscall overhead. Once the peer receives the `RESEND` in the XDP hook, XDP will piggyback the lost range and RPC buffer address (maintained in the kernel RPC state) on the packet and redirect it to the user-space library through AF_XDP for retransmission.

Other control-path events include maintaining infrequently changing priority cutoff values [55], freeing states in remote RPC servers when RPC clients do not explicitly confirm RPC completions, and aborting RPCs caused by machine crushes or long timeout by forcibly removing corresponding states.

Homa data path. Homa uses receiver-driven credit scheduling to achieve SRPT (Shortest Remaining Processing Time); eTran implements the credit scheduling at the XDP hook, while also enforcing sender-side SRPT at the `XDP_EGRESS` hook. When the XDP eBPF program receives the first *unscheduled packet* [50] of an RPC message, it creates a new CC state object, including the RPC ID and remaining bytes through `bpf_obj_new`, and enqueues it into a `bpf_rbtree` map, i.e., *credit list*. For subsequent packet arrivals, the eBPF program traverses the credit list, dequeues the corresponding state object, updates the state object with new remaining bytes, and re-enqueues it into the credit list.

The eBPF program at the `XDP_GEN` hook is responsible for selecting RPCs and sending credit packets. Conceptually, it has the following four steps: 1) selecting candidate RPCs from the credit list; 2) looking up the RPC state hashmap to find corresponding RPC states, then allocating credits; 3) removing finished RPCs from the credit list; 4) sending credits for the selected RPCs. However, we face two challenges due to eBPF constraints: lack of searching functionality in `bpf_rbtree` to implement RPC selections; instruction limit posed by the eBPF verifier disallows offloading complex logic.

eTran addresses these challenges by innovatively utilizing existing eBPF features. For the first challenge, we wrap a new eBPF kfunc called `bpf_rbtree_lower_bound` that returns the first node whose value is not less than the provided one; we then leverage this kfunc to select candidate RPCs ranked by their priorities while guaranteeing not selecting more than one RPC for each peer (required by Homa [55]). A common way to implement such RPC selections is through a 2-level rbtree as Homa does, but requires holding multiple locks simultaneously, which eBPF prohibits. Instead, we use `bpf_rbtree_lower_bound` to emulate such 2-level rbtree with a single rbtree; see Appendix B for more detail. For the second challenge, we use `bpf_tail_call` to break complex eBPF logic into multiple programs that go through the verifier separately. To pass states across different tail-called eBPF programs, we exploit per-CPU variables in the `bss` section.

On the sender side, the `XDP_EGRESS` eBPF program queues packets into the pacing engine (see §3.2.2) when no credit is available for this RPC. Upon credit replenishment, the callback enforces SRPT across RPCs using a `bpf_rbtree` of queued RPCs, sorted by the remaining bytes to send.

Due to space limitation, we elaborate on how eTran enables a flexible development model covering transport developers, application developers, and more in Appendix C.

5 eTran Implementation

Our eTran prototype contains 24K lines of C/C++, spreading across the Linux kernel, control path daemon, eBPF programs, and user-space transport library, as shown in Table 1. The

Category	Kernel	Control path daemon	eBPF programs: TCP/Homa	Transport libs: TCP/Homa
LoC	2597	8224	2173/5349	3661/2024

Table 1: eTran codebase breakdown with 24028 LoC in total.

current prototype runs on Linux kernel v6.6.0 and targets Mellanox mlx5 driver. Our Linux kernel change includes the two new hooks, the new eBPF map with associated kfuncs (for packet manipulation), `bpf_rbtree`'s new API as a kfunc, and AF_XDP's out-of-order completion with mlx5 driver support. We only make ~20 LoC change for the mlx5 driver, so that the driver can maintain the addresses of packet frames that get queued in the pacing engine, and only place them in the completion ring when they are eventually transmitted.

eBPF verifier. eTran does not change the verifier but just registers new eBPF hooks and kfuncs with the verifier for safety and security. In general, there are three types of checks done by the eBPF verifier: (a) eBPF program behaviors (e.g., bounded loops, NULL pointers, deadlock, etc.), (b) how eBPF programs access input contexts, and (c) how eBPF programs use eBPF helpers and kfuncs. Naturally, the eBPF verifier checks (a) for all eTran eBPF programs. It also checks (b) for `XDP_GEN` and `XDP_EGRESS` similarly to the existing XDP.

For (c), the principle is that eBPF programs must call these helpers and kfuncs safely and securely. For safety, we ensure the used kfuncs (either wrapped from the existing kernel functions or written by us) are non-preemptible and have finite execution steps. For example, our `XDP_EGRESS` hook uses the kfunc wrapped from the kernel `dev_xdp_enqueue()` function, which meets the safety requirement as it was designed to run in the softirq context. Our added kfunc for `bpf_rbtree` follows the implementation practice of existing eBPF helpers with extensive testing; nevertheless, we plan to verify it formally in the future. For security, we expose only necessary eBPF helpers to our new hooks and carefully reason through its security implications. The eBPF helpers exposed to the new hooks are only a `subset` of the ones exposed to XDP by kernel developers. For example, we exclude all socket-related helpers (e.g., `bpf_sk_lookup_tcp`) as the new hooks don't access these sockets; we exclude `bpf_redirect_map` for `XDP_GEN` to prevent redirecting packets arbitrarily.

6 Evaluation

This section aims to answer the following questions:

1. What are the latency and throughput of TCP and Homa under eTran compared to existing kernel ones and kernel-bypass ones (§6.1 and §6.2)?
2. How well does eTran support multi-tenancy and traffic management (through our new eBPF map) (§6.3)?
3. What is the performance of our new eBPF hooks (§6.4.1) and retransmission design (§6.4.3)?

Experiment setup. We use 10 `xl170` physical machines from CloudLab [19], each equipped with two 10-core Intel E5-2640v4 CPUs (2.4GHz), 64GB memory, and a Mellanox ConnectX–4 25 Gbps NIC. All machines are connected via a Mellanox 2410 switch under the same rack. For eTran (Homa/TCP) and Linux (Homa/TCP), all machines run Linux kernel v6.6.0. For TAS, we are unable to run it on Linux kernel v6.6.0 due to Mellanox driver issues, but we manage to run it on Linux kernel v5.15.0. Since TAS is built atop DPDK, we expect no performance differences between the two kernel versions. We only enable TCP Segmentation Offload (TSO) for Linux (Homa/TCP) as AF_XDP does not currently support it (see §7). There is no zero-copy optimization for eTran for a fair comparison (i.e., keep API semantics the same). Following prior work [80, 81], we disabled the NIC interrupt coalescing feature. All experiments use the default maximum packet size (MTU) of 1500 bytes. For DCTCP, we set the ECN marking threshold to 70KB, as done in prior work [55].

Comparison baselines. We compare eTran (Homa) to Linux/Homa (commit `8321cde`), a kernel module implementation of Homa that is highly optimized by the Homa authors. We refer to it as Linux (Homa) throughout this paper. We compare eTran (TCP) to TAS (commit `d3926ba`), a microkernel-style user-space TCP stack built on DPDK, and Linux (TCP) with DCTCP congestion control. By default, for eTran and TAS, we separate CPU cores for applications and interrupt processing/busy polling, and provision a dedicated core for control path/slow path. Linux (Homa) and Linux (TCP) use all provisioned cores to handle network load and applications.

6.1 Comparision with Linux (Homa)

Microbenchmarks. We first compare the basic latency and throughput between eTran (Homa) and Linux (Homa). We use a similar configuration described in the Homa paper [55]. The median latency of short messages and the throughput of large messages are measured by using a single client thread to send back-to-back requests (32B/1MB) to a single-threaded server, which responds with a 32-byte response. Next, we measure the throughput of a multi-threaded server receiving concurrent RPCs (500KB) from 7 clients, as well as the throughput of a multi-threaded client sending concurrent RPCs to 7 servers. Finally, we measure the RPC rate for small messages (32B), maintaining the same client-to-server ratio.

Table 2 summarizes the result. eTran (Homa) achieves lower median latency than Linux (Homa), i.e., 11.8μs vs. 15.6μs. eTran (Homa) outperforms Linux (Homa) in large message throughput, even with Linux (Homa) using TSO batching. In terms of RPC rate, eTran (Homa) achieves approximately 1.7-1.8$\times$ the message throughput of Linux (Homa). This gap is because eTran (Homa) uses AF_XDP for efficient packet IO while Linux (Homa) has a larger code path and uses costly structures like `sk_buff`.

Cluster benchmark. All the following experiments utilize the same cluster benchmark application from Linux (Homa) [55], conducted with 10 machines. In this experiment, each node serves as both a multi-thread client and a multi-thread server simultaneously. Clients randomly select servers to issue a batch of RPCs. Given that the data of modern RPCs primarily

	eTran (Homa)	Linux (Homa)
32B median latency (μs)	11.8	15.6
1MB throughput (Gbps)	17.7	14.5
Server throughput (Gbps)	23.0	23.1
Client throughput (Gbps)	22.7	22.9
Client RPC rate (Mops)	2.9	1.7
Server RPC rate (Mops)	3.3	1.8

Table 2: Comparison of basic latency and throughput between eTran (Homa) and Linux (Homa).

flows in one direction [64], we vary the request size up to 1MB while keeping the response size fixed as a small message. We use workloads W2-W5 in Homa paper [55] where W2 and W3 are dominated by short messages and W4 and W5 are dominated by large messages. Slowdown is the client-observed RTT divided by the ideal RTT for the same-length RPCs using eTran (Homa). For fairness, eTran (Homa) is configured with the same parameters as Linux (Homa).

Figure 5 shows the latency slowdown for workloads W2-W5. eTran (Homa) outperforms Linux (Homa) across all workloads. For the short message dominated workloads W2 and W3, eTran (Homa) achieves 3.9-7.5× lower P99 tail latency and 1.4-3.6× lower P50 latency compared to Linux (Homa). This is because: for W2 and W3 where the bottleneck is software overhead, eTran (Homa) leverages AF_XDP to achieve more efficient packet processing. For W4 and W5, which mainly consist of large messages, eTran (Homa) also slightly outperforms Linux (Homa). eTran (Homa) occasionally experiences higher tail latency than Linux (Homa) for large messages. This is because less optimized thread scheduling in eTran (Homa) causes interference between the pacing thread and applications; Linux (Homa) also faced this issue but mitigated it by carefully scheduling both threads to ensure pacing neither falls behind nor dominates CPUs. We plan to incorporate such transport-specific scheduling policies in the future, possibly through eBPF as well [63].

Figure 6 shows the RTT distributions for the shortest messages (10%) in W4 and W5, where eTran (Homa) achieves lower latencies across almost all percentiles. For example, eTran (Homa) achieves 4.1× lower P50 latency than Linux (Homa) in W4 and 3.9× lower in W5. The P99 latency is reduced by 4.3× in W4 and 2.9× in W5.

6.2 Comparision with Linux (TCP) and TAS

We built two POSIX API-based applications: a simple echo server and a key-value store derived from the TAS codebase (modeled after Memcached). The server handles requests using `epoll()`, and the client generates requests in a closed loop. All systems use the same binary but link to different POSIX implementations (e.g., Linux, TAS, and eTran).

Message throughput. Figure 7a shows the throughput of transferring a single back-to-back large message. eTran (TCP) performs comparably to Linux (TCP) but slightly lags behind TAS; for messages under 64KB, eTran (TCP) surpasses Linux (TCP). Figure 7b shows the throughput for small messages

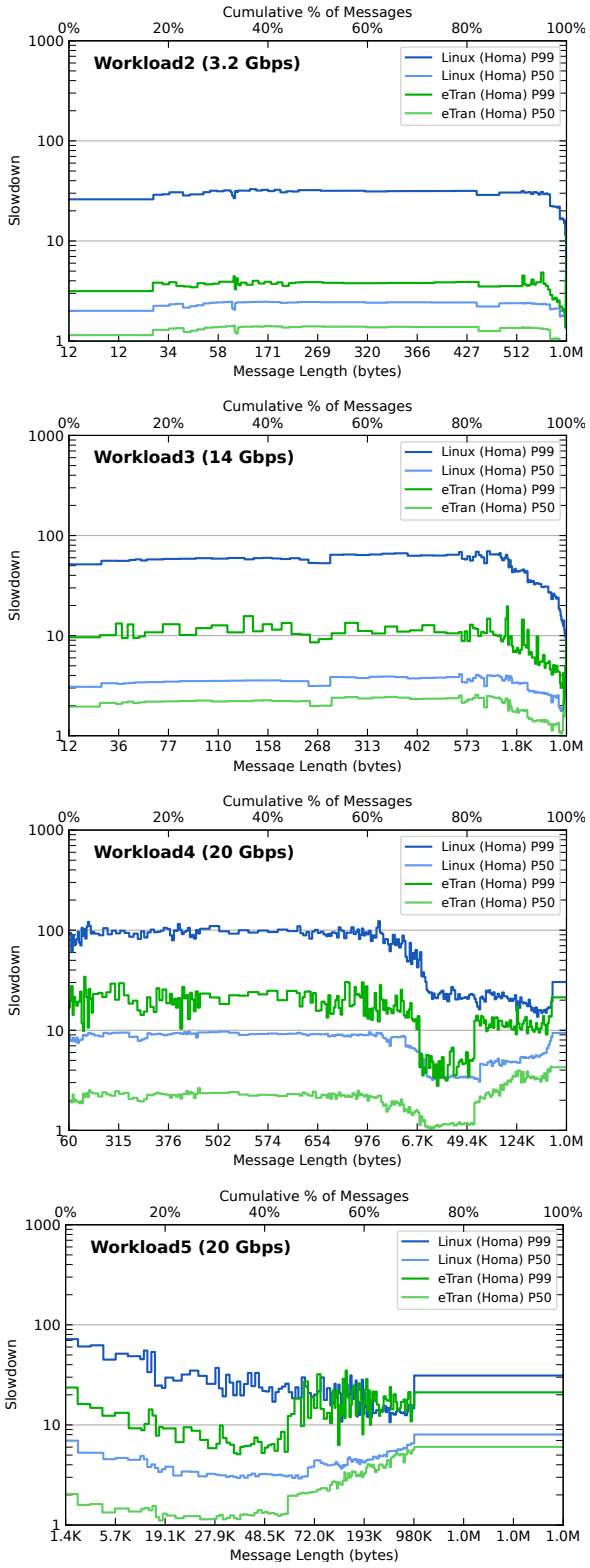


Figure 5: Median and 99th-percentile latency slowdown vs. message size for workloads W2–W5. The x-axis is scaled linearly to represent the number of RPCs, reflecting each workload's CDF of message lengths. W2 and W3 are dominated by short messages, while W4 and W5 feature larger messages.

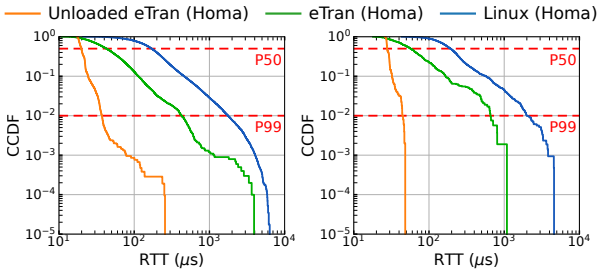
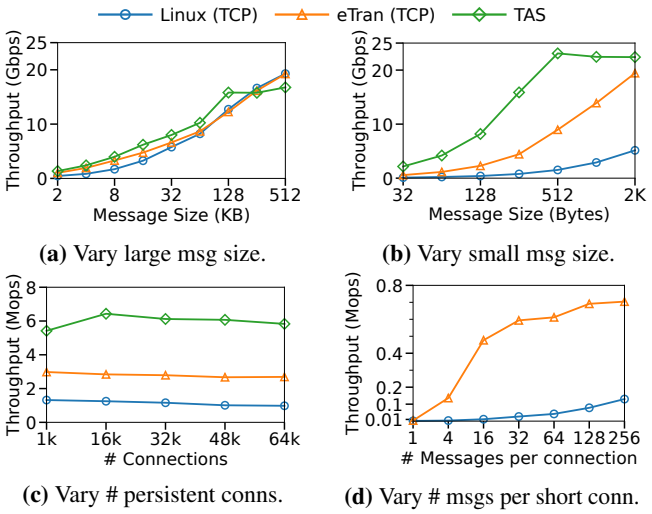


Figure 6: Complementary CDF of round-trip latency for the shortest 10% of RPCs in W4 (left) and W5 (right).



(a) Vary large msg size. **(b)** Vary small msg size.

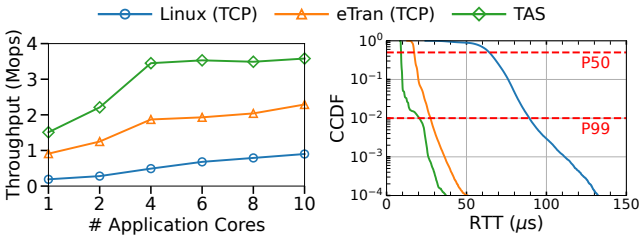
(c) Vary # persistent conns. **(d)** Vary # msgs per short conn.

Figure 7: Comparison of message throughput for the TCP echo. Figure (a) has a single outstanding message, while Figure (b) has 64. TAS fails to achieve stable throughput for Figure (d), thus omitted.

with 64 outstanding messages per connection. This is measured using a single-threaded server handling 100 connections, partitioned across five clients each with five threads. For example, with 1KB messages, eTran (TCP) and TAS achieve 4.8× and 7.7× higher throughput than Linux (TCP), respectively. The gap between eTran (TCP) and TAS stems from TAS's use of dedicated cores for DPDK-based busy polling, while eTran (TCP) is interrupt-driven. Nevertheless, eTran (TCP) consistently outperforms Linux (TCP) across all message sizes and achieves 87% of TAS throughput for 2KB messages.

Connection scalability. We use as many clients as possible to generate the load with persistent connections. Clients send a single 64B request per connection and wait for a response in a closed loop. Figure 7c shows the result. The throughput of eTran (TCP) falls between Linux (TCP) and TAS: e.g., with 1K connections, eTran (TCP) achieves 2.26× the throughput of Linux (TCP), while TAS achieves 4.1× the throughput of Linux (TCP). Linux suffers high syscall overhead with limited TSO/GRO benefits with many connections. eTran (TCP) batches processing but is still less efficient than TAS due to the performance gap between AF_XDP and DPDK [39].

Short-lived connections. In this experiment, the client uses four threads to establish a total of 1K short-lived concurrent flows with a single-threaded server. Upon receiving the server's response for each connection, the client immediately closes the connection and establishes a new one. The result for TAS is omitted as it fails to achieve stable throughput due to its implementation bugs. As shown in Figure 7d, with one message per connection, the throughput of eTran (TCP) and Linux (TCP) is similar. With four or more messages per connection, eTran (TCP) significantly outperforms Linux (TCP). For example, eTran (TCP) achieves 42.7× and 5.4× the throughput of Linux (TCP) with 16 and 256 messages per connection, respectively. This is because the eTran (TCP) control path uses AF_XDP to perform efficient packet IOs for connection handshake and teardown, while the data path utilizes efficient LRPC to communicate with the control path.



(a) KV tpt vs. #app cores. **(b)** CCDF of KV RTT.

Figure 8: Performance comparison of the TCP key-value store.

Key-Value Store. We use a workload that has 100K key-value pairs (32B keys, 64B values) following a Zipf (s=0.9) distribution, with a 9:1 GET to SET ratio. Each of the five client machines establishes 6K persistent connections, leaving 32 outstanding requests per connection. Figure 8a shows that eTran (TCP) achieves 2.4-4.8× the throughput of Linux (TCP), with lower performance than TAS (which achieves 3.9-7.9× of Linux). Figure 8b provides more detailed insights into the latency when the server is under-loaded: eTran (TCP) has slightly higher latency than TAS but is significantly lower than Linux (TCP). More specifically, eTran (TCP) achieves 3.7× (17.2 µs vs. 64.2 µs) lower P50 latency and 3.2× (27.5 µs vs. 89.3 µs) lower P99 latency compared to Linux (TCP).

6.3 More Features

6.3.1 Multi-Tenancy and Coexistence

eTran (TCP/Homa) supports multi-tenancy and can coexist with Linux (TCP/Homa), thanks to XDP/AF_XDP not taking over the entire NIC. Figure 9 shows the throughput timeline when using different protocols. Launching our control path slightly reduces Linux throughput due to the extra XDP processing. This overhead can be removed by loading XDP on isolated queues from Linux (TCP/Homa) [71] and using hardware mechanisms (e.g., Flow Director [16]) for isolation.

6.3.2 Traffic Management

As an example, we show that eTran enforces accurate rate limiting with low overhead using `PKT_QUEUE` and pacing engine (§3.2.2). eTran tail-calls a simple eBPF program after `XDP_EGRESS` to retrieve the target rate from a policy table and

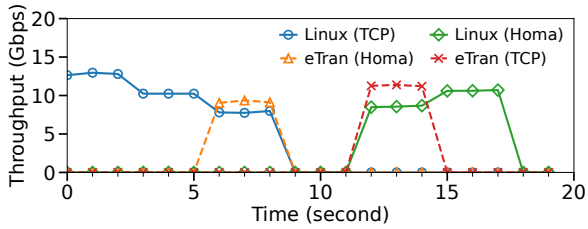
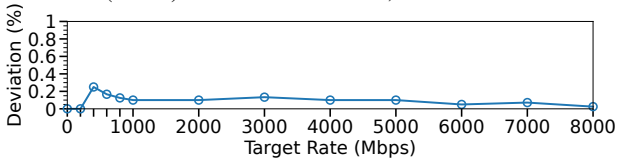
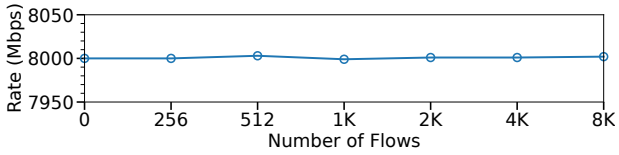


Figure 9: At 0 sec, APP1 starts TCP flows. At 3 sec, eTran launches the control path and loads eBPF. At 5 sec, APP2 sends RPCs with eTran (Homa). At 9 sec, both apps exit, and the control path switches protocols. At 12 sec, APP3 runs eTran (TCP), and APP4 sends RPCs with Linux (Homa). APP3 exits at 15 sec, and APP4 at 18 sec.



(a) The deviation from the target rate of eTran for a single flow.



(b) The actual rate achieved by eTran with a target rate of 8Gbps.

Figure 10: Rate conformance of eTran.

calculate the timestamp; it then uses this timestamp in the pacing engine to regulate the traffic rate. Figures 10a and 10b show rate conformance for single and multiple flows, with deviations under 0.4% as the rate and flow count increase.

6.4 Design Drill-Down

6.4.1 eBPF Hook Performance

We show that our new hooks offer powerful functionalities with minimal overhead. We employ three benchmarks from `bpf-examples` [9]: `tx-only` for small packet transmission, `l2fwd` for layer 2 forwarding, and `rx-drop` for packet discarding. 64B packets are used to stress test the performance.

XDP_EGRESS overhead. Table 3 shows that an empty hook reduces throughput by 6.6%. Out-of-order completion further lowers it to 86.1%. The primary overhead in `XDP_EGRESS` comes from constructing new packet buffers, which is mitigated by pre-allocations (§3.2.1). The overhead of out-of-order completion comes from higher cache miss rate incurred by increased memory footprint. As a reference, the table also notes the cost of map lookup operations in `XDP_EGRESS`.

XDP_GEN performance. We run `rx-drop`+XDP_GEN on two cores: one core generates ACK/credit packets using XDP_GEN, and another uses AF_XDP to drop received packets. We compare it to `l2fwd` on two cores: one core in the kernel redirects packets to AF_XDP, and another uses AF_XDP to forward packets back as ACK/credit packets. Table 4 shows that our overall throughput is slightly lower than `l2fwd` but with higher per-core throughput. This is because we avoid cross-core communication for ACK/credit packet transmission, as it is done inside the kernel. We conclude that `XDP_GEN` is efficient enough in transmitting ACK/credit packets.

Action	Tpt (Mpps)	Tpt loss (%)
AF_XDP `tx-only`	11.55	-
+ Empty XDP_EGRESS	10.79	6.6
+ OOO comp	9.95	13.9
+ ARRAY lookup	9.71	15.9
+ HASHMAP lookup	9.10	21.2

Table 3: Performance of XDP_EGRESS (single core).

	Tpt (Mpps)	# cores	Per-core tpt (Mpps)
`l2fwd`	6.73	1.74	3.87
`rx-drop` + XDP_GEN	6.03	1.35	4.47

Table 4: `l2fwd` vs. `rx-drop` + XDP_GEN on receiving data packets and responding with ACK/credit packets.

6.4.2 CPU Overhead Breakdown

Table 5 presents the CPU overhead across the networking stack for a multi-threaded server. We stress-test transport processing with a single NAPI, isolating it from applications explicitly for all systems. We made great efforts to tune the Linux stacks to be even better than the one reported in [55] under the same hardware. Note that this configuration is not general for all applications. Per-function cycles are obtained using Perf [29] and categorized with tools extended from [12]. Homa incurs higher overheads than TCP in both implementations due to frequent allocation/destruction for small RPCs. eTran significantly reduces primary overhead (*Socket/RPC*, *TCP/Homa+IP*) through streamlined implementation, with optimizations to *Sk_buff*, *Memory Mgmt* via lightweight xdpbuff and pre-allocated UMEM. *Scheduling* is optimized through batched IO and decoupled kernel transport processing. For Linux, per-request context switching incurs higher instruction counts, pipeline stalls, memory footprint, and poor cache locality [41]. *Other* overheads (e.g., security checks, kernel/user mode switching) are reduced due to file system bypassing and AF_XDP's syscall-free asynchronous IO.

6.4.3 Impact of Retransmission Design

We show that retransmission handling in the control path matches Linux (Homa) and Linux (TCP) performance. By launching 100 flows between two machines and measuring throughput under varying loss rates, Figure 11 shows the throughput penalty of different protocols. eTran (TCP) incurs a larger throughput penalty than TCP due to limited out-of-order optimization. Homa's retransmission is simple as packet loss is rare; both eTran (Homa) and Linux (Homa) show similar throughput decline. Due to space constraints, we show additional experiments on control path CPU usage, multi-queue impact, and interrupt interference in Appendix D.

7 Discussion and Future Work

Emerging kernel techniques. Many emerging kernel techniques inspire eTran design or might help improve eTran per-

Component	eTran (T)	Linux (T)	eTran (H)	Linux (H)
Application	0.48	0.53	0.95	1.04
Socket/RPC	0.63	**3.5**	0.98	**3.38**
Data copy	0.19	0.57	0.32	1.30
Sk_buff	0.15	0.47	0.08	0.39
TCP/Homa+IP	1.06)	**2.12**	1.47	**3.36**
Lock/Unlock	0.18	0.45	0.24	2.68
NIC Driver	1.17	1.54	0.83	1.81
Memory Mgmt	0.05	0.32	0.06	1.04
Scheduling	0.25	1.19	0.18	1.02
Other	0.21	1.82	0.38	1.41
Total (kcycles)	**4.37**	**12.51**	**5.48**	**17.43**

Table 5: Breakdown of CPU Cycles per Request. **T**: TCP; **H**: Homa.

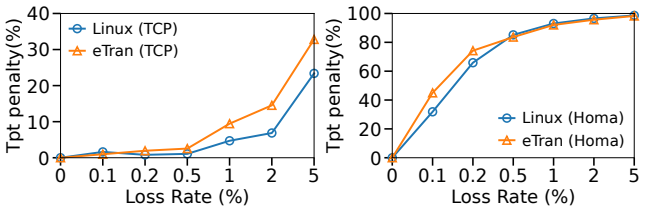
formance. For example, the Linux kernel community made efforts to introduce queueing for XDP when redirecting among different network interfaces [33]; instead, our pacing engine is to provide a general queueing mechanism for packet transmission. We also note that the existing Linux qdisc (queueing discipline) cannot sustain the high packet rate as our pacing engine [62]. From Linux kernel v6.6, some NIC drivers support AF_XDP multi-buffer [22], enabling features like TSO and scatter/gather. Unfortunately, the mlx5 driver we are using does not support them yet; we expect it could help improve eTran's performance for large messages/long flows.

Improving eBPF programming. eTran suffers from limited synchronization support in eBPF programming. For example, eBPF prohibits holding two locks simultaneously, posing significant challenges for implementing Homa's credit list (§4.2). Another example is that eBPF lacks blocking synchronization primitives like Mutex, necessitating careful spinlock use. We understand this limitation is rooted in the fact that the eBPF execution is non-sleepable and non-preemptable to simplify safety verification, but any relaxation of it would hugely simplify eBPF programming for complex applications.

eTran limitations. eTran relies on the eBPF subsystem to customize transports, suffering from similar limitations as eBPF. For example, eBPF does not support floating point arithmetic, motivating eTran to adopt the CCP architecture to separate CC into the control path daemon for sender-driven transports. Currently, eTran depends on the kernel for CPU scheduling, which can lead to tail latency issues, as noted in prior works [24, 54]. In addition, since AF_XDP bypasses kernel layers, eTran lacks the integration of other kernel subsystems such as file system. We leave them for future work.

8 Related Work

High-performance network transports. There has been a line of work on high-performance transports, e.g., mTCP [36], IX [4], Arrakis [57], TAS [41], eRPC [38], Snap [47], Demikernel [78] and Junction [23]. Most rely on kernel-bypass, busy polling to achieve high performance and co-locate with applications in the same process context (*library OS*). In contrast, eTran transports operate within the kernel interrupt-driven NAPI context, with protection ensured by eBPF. This trades performance for CPU efficiency and prevents applications



(a) Comparison of TCP. **(b)** Comparison of Homa.

Figure 11: Throughput penalty at varying packet loss rates.

from interfering with transport processing and states. Both IX and Arrakis provide transport protection for *library OS*. However, IX repurposes virtualization hardware non-standardly, while Arrakis relies on NIC virtualization, which is more limited and inflexible than software-based approaches. TAS and Snap separate the kernel-bypass transport on isolated cores from applications (*Microkernel*). Although retaining flexibility, it must reimplement many functionalities from scratch. Conversely, eTran still uses some kernel networking infrastructures, and new transports can co-exist with existing kernel ones. eTran also differs from *kernel* transports: eTran transports are split to the control path (user-space) and the data path (kernel eBPF); the untrusted library uses AF_XDP socket for efficient IO and metadata exchange (rather than relying on syscalls), while being restricted from directly accessing kernel transport states maintained in eBPF. This design enables safely customizing high-performance kernel transports.

eBPF verification. eTran relies on verification techniques to guarantee the safety of running application-supplied eBPF programs inside the kernel. Therefore, a bunch of eBPF verification work in this space such as [8, 40, 46, 69, 73, 76] would benefit eTran and enhance our kernel safety and security.

Kernel offloads. Offloading application functions to the kernel via eBPF has recently emerged to avoid kernel overheads and improve performance [11, 27, 37, 56, 59, 79–81]. Different from offloading applications via eBPF, eTran targets making kernel transport extensible by extending eBPF subsystems.

Extensible kernels. eTran adopts a similar approach that is reminiscent of the extensible kernel research in the 1990s such as SPIN [7], Exokernel [21], and VINO [65]. The key difference is that eTran relies on existing kernel eBPF subsystem for safety and targets performance-hungry network transports in modern datacenters. There is a line of recent work on making kernel TCP more extensive via eBPF [10, 31, 35, 70, 77]. However, they only focus on the TCP transport and allow customizing a small portion of TCP behaviors such as initial congestion window, CC schemes, and path management.

9 Conclusion

eTran is an extensible kernel transport system, achieving agile customization, kernel safety, strong protection, and high performance simultaneously. It achieves these goals by extending the kernel-safe eBPF, hiding transport states inside the kernel, and absorbing techniques from user-space transports. eTran code is available at https://github.com/eTran-NSDI25/eTran.

Acknowledgments

We thank our shepherd Boris Pismenny and the anonymous reviewers for their helpful comments. We thank Cloudlab [19] for the development and evaluation infrastructure. Zhongjie Chen, Yifan Liu, and Fengyuan Ren are supported in part by the National Key Research and Development Program of China (No. 2022YFB2901404), and by the National Natural Science Foundation of China (NSFC) under Grant No. 62132007 and No. 62221003. ChonLam Lao, Minlan Yu, and Yang Zhou are supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou is also supported by a Google PhD Fellowship and the UC Berkeley Sky Computing Lab.

References

[1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *Proceedings of USENIX NSDI*, pages 51–70, 2022.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, page 63–74, 2010.

[3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *Proceedings of USENIX NSDI*, pages 93–109, 2020.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[5] Steven M Bellovin. Security Problems in the TCP/IP Protocol Suite. *ACM SIGCOMM CCR*, 19(2):32–48, 1989.

[6] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Call. *ACM SIGOPS Operating Systems Review*, 23(5):102–113, 1989.

[7] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of ACM SOSP*, pages 267–283, 1995.

[8] Sanjit Bhat and Hovav Shacham. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis, 2022.

[9] The bpf examples authors. bpf-examples. https://github.com/xdp-project/bpf-examples.

[10] Lawrence Brakmo. TCP-BPF: Programmatically Tuning TCP Behavior through BPF. *Proc. NetDev*, 2:1–5, 2017.

[11] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A Database Proxy That Bounces with User-Bypass. *Proceedings of the VLDB Endowment*, 16(11):3335–3348, 2023.

[12] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of ACM SIGCOMM*, pages 65–77, 2021.

[13] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *Proceedings of ACM SIGCOMM*, pages 767–779, 2022.

[14] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of ACM SIGCOMM*, page 239–252, 2017.

[15] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *Proceedings of USENIX OSDI*, pages 299–314, 2020.

[16] Intel Corporation. Introduction to Intel Ethernet Flow Director and Memcached Performance. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

[17] Intel Corporation. Single Root I/O Virtualization (SR-IOV) for Intel NICs. https://www.intel.com/content/www/us/en/support/articles/000005722/ethernet-products.html.

[18] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, et al. A Large-Scale Deployment of DCTCP. In *Proceedings of USENIX NSDI*, pages 239–252, 2024.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[20] Kumar Kartikeya Dwivedi. Allocated objects, BPF linked lists. https://lwn.net/Articles/915404/.

[21] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[22] Maciej Fijalkowski. xsk: multi-buffer support. https://lwn.net/Articles/937525/.

[23] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *Proceedings of USENIX NSDI*, pages 55–73, 2024.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[25] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of ACM SIGCOMM 2024 Conference*, page 57–70, 2024.

[26] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the ACM CoNEXT*, 2015.

[27] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[28] Google. Falcon transport protocol. https://github.com/opencomputeproject/OCP-NET-Falcon.

[29] Brendan Gregg. Perf Examples. https://www.brendangregg.com/perf.html.

[30] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of ACM SIGCOMM*, page 29–42, 2017.

[31] Jörn-Thorben Hinz, Vamsi Addanki, Csaba Györgyi, Theo Jepsen, and Stefan Schmid. TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 1–7, 2023.

[32] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of ACM SIGCOMM*, page 422–434, 2020.

[33] Toke Høiland-Jørgensen. Adding packet queueing to XDP. https://lpc.events/event/16/contributions/1351/attachments/1049/2037/xdp-queueing.pdf.

[34] IETF TLS Working Group. Transport layer security. https://datatracker.ietf.org/wg/tls/about/.

[35] Mathieu Jadin, Quentin De Coninck, Louis Navarre, Michael Schapira, and Olivier Bonaventure. Leveraging eBPF to Make TCP Path-Aware. *IEEE Transactions on Network and Service Management*, 19(3):2827–2838, 2022.

[36] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[37] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[38] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[39] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, volume 37, page 38, 2018.

[40] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, Flexible, and Practical Kernel Extensions. In *Proceedings of ACM SOSP*, 2024.

[41] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[42] The Linux kernel development community. BPF Kernel Functions (kfuncs). https://docs.kernel.org/bpf/kfuncs.html.

[43] The Linux kernel development community. XDP RX Metadata. https://docs.kernel.org/networking/xdp-rx-metadata.html.

[44] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of ACM SIGCOMM*, pages 514–528, 2020.

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

[46] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. Towards Functional Verification of eBPF Programs. In *Proceedings of ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 37–43, 2024.

[47] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[48] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From Luna to Solar: the Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of ACM SIGCOMM 2022 Conference*, pages 753–766, 2022.

[49] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[50] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of ACM SIGCOMM*, pages 221–235, 2018.

[51] Multipath TCP community. Multipath TCP. https://www.multipath-tcp.org/.

[52] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[53] Atsuya Osaki, Manuel Poisson, Seiki Makino, Ryusei Shiiba, Kensuke Fukuda, Tadashi Okoshi, and Jin Nakazawa. Dynamic Fixed-point Values in eBPF: a Case for Fully In-kernel Anomaly Detection. In *Proceedings of the Asian Internet Engineering Conference 2024*, pages 46–54, 2024.

[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[55] John Ousterhout. A Linux Kernel Implementation of the Homa Transport Protocol. In *Proceedings of USENIX ATC*, pages 99–115, 2021.

[56] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

[57] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[58] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[59] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[60] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of ACM SIGCOMM*, page 691–706, 2024.

[61] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of ACM HotOS*, pages 152–158, 2021.

[62] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of ACM SIGCOMM*, pages 404–417, 2017.

[63] The sched_ext authors. Sched_ext Schedulers and Tools. https://github.com/sched-ext/scx.

[64] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of ACM SOSP*, pages 498–514, 2023.

[65] Margo I Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. *ACM SIGOPS Operating Systems Review*, 30(213-228):10–1145, 1996.

[66] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.

[67] Alexei Starovoitov. bpf: Introduce BPF timers. https://lwn.net/Articles/862136/.

[68] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *Proceedings of USENIX ATC*, pages 431–444, 2017.

[69] Hao Sun and Zhendong Su. Validating the eBPF Verifier via State Embedding. In *Proceedings of USENIX OSDI*, pages 615–628, 2024.

[70] Viet-Hoang Tran and Olivier Bonaventure. Making the Linux TCP Stack more Extensible with eBPF. In *Proc. of the Netdev 0x13, Technical Conference on Linux Networking*, 2019.

[71] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Proceedings of ACM SIGCOMM*, pages 245–257, 2021.

[72] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). *ACM SIGCOMM CCR*, 42(4):115–126, 2012.

[73] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the Verifier: eBPF Range Analysis Verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023.

[74] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific Network Transport for Distributed DNN Training. In *Proceedings of USENIX NSDI*, pages 1421–1443, 2024.

[75] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. *ACM SIGCOMM CCR*, 41(4):50–61, 2011.

[76] Qiongwen Xu, Michael D Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. In *Proceedings of ACM SIGCOMM*, pages 50–64, 2021.

[77] Sepehr Abbasi Zadeh, Ali Munir, Mahmoud Mohamed Bahnasy, Shiva Ketabi, and Yashar Ganjali. On Augmenting TCP/IP Stack via eBPF. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 15–20, 2023.

[78] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[79] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

[80] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of USENIX NSDI*, pages 1391–1407, 2023.

[81] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Proceedings of USENIX NSDI*, 2024.

[82] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of ACM SIGCOMM*, page 523–536, 2015.

APPENDIX

A Implementing More Transports

In this section, we extend our case studies beyond Homa and TCP, exploring the implementation of other transport protocols (specifically, their host logic) under eTran. We focus on congestion control (CC) while loss recovery and segmentation/reassembly are omitted as they are less important.

Swift [44] is a representative RTT-based transport protocol. It adjusts the congestion window using RTT measurements and distinguishes fabric congestion from host congestion. Swift's CC algorithm can be implemented entirely in XDP. Swift typically uses window control and switches to rate-based pacing when the congestion window falls below 1. This can be implemented with two maps of `BPF_MAP_TYPE_PKT_QUEUE`. Swift requires NIC hardware timestamp, which can be provided via XDP RX metadata [43].

HPCC [45] is a transport protocol that utilizes in-network telemetry (INT) to gather precise load information for CC. The CC algorithm on the sender side of HPCC can be implemented using XDP. Additionally, HPCC employs rate-based pacing with window boundaries, which can be implemented similarly. Specifically, HPCC incorporates a flow scheduler that scans all flows to assign credits. Only flows that have accumulated sufficient credits and are within the window limit can be transmitted. This flow scheduler can be implemented using the dedicated kthread of the BPF timer.

DCQCN [82] is another ECN-based transport protocol that integrates with QCN. On the sender side, DCQCN controls the sending rate using a per-flow timer and a byte counter. The byte counter is updated in XDP, while the per-flow timer can be implemented using an array of BPF timers. Each connection tracks the index of its corresponding BPF timer, similar to how the CC state index is managed in DCTCP. DCQCN enforces rate control entirely based on the rate, which can be implemented similarly to DCTCP.

NDP [30] is a receiver-driven transport protocol that incorporates packet trimming. The receiver-driven logic and credit-based pacing can be implemented similarly to Homa. The main difference is that NDP does not prioritize short messages like Homa. Instead, it lets all flows share bandwidth fairly by default.

B RBtree Operations

The fundamental approach involves utilizing a single red-black tree to manage two distinct trees. We assume each machine is assigned a unique *peerid*, and each RPC is identified by a unique *rpcid*, which consists of a five-tuple. The first tree, termed the *RPC Tree*, maintains the states of all RPCs. The second tree, referred to as the *Peer Tree*, tracks the highest priority RPC from each peer. In the *RPC Tree*, the key for lookups is defined as (treeid = 0, peerid, bytes_remaining, rpcid). In the *Peer Tree*, the key for lookups is defined as (treeid = 1, bytes_remaining, peerid). Keys are compared from left to right. All operations are performed using three APIs: `bpf_rbtree_add()`, `bpf_rbtree_remove()`, and `bpf_rbtree_lower_bound()`, which have the time complexity of $O(\log N)$.

Operations in XDP. When XDP receives a new RPC that needs to be scheduled, it creates an object and inserts it into the *RPC Tree* and takes a snapshot of the value of bytes_remaining in the RPC state for future search. Subsequently, XDP tries to adjust the *Peer Tree*. It starts by searching the *RPC Tree* using the key (treeid = 0, peerid = our_peerid, bytes_remaining = 0, rpcid = 0).

1. If the search results in (treeid $\neq$ 0 or peerid $\neq$ our_peer or rpcid $\neq$ our_rpcid), it indicates that we are not the highest priority RPC for this peer, and no further action is taken.
2. If the search results in (treeid = 0 and peerid = our_peer and rpcid = our_rpcid), it indicates that we are the highest priority RPC for this peer. In this case, we search the *RPC Tree* with the key (treeid = 0, peerid = our_peer, bytes_remaining = our_remaining, rpcid = our_rpcid + 1) to determine if a previous highest priority RPC exists. If such an RPC is found, it is removed from the *Peer Tree*. The new RPC is then inserted into the *Peer Tree*, and its status is marked as present in the *RPC Tree*.

When a new batch of packets arrives for this RPC, a search is performed in the *RPC Tree* using the key (treeid = 0, peerid = our_peer, bytes_remaining = snapshot_bytes_remaining, rpcid = our_rpcid). If no matching object is found, it indicates that the RPC has fully used granted, and no action is needed. Otherwise, it is removed from the *RPC Tree*, its bytes_remaining value is updated, and it is reinserted into the *RPC Tree*. Then, we adjust the *Peer Tree*:

1. If the RPC is already present in the *Peer Tree*, indicating it is still the highest priority RPC from this peer, the key (treeid = 1, peerid = our_peer, bytes_remaining = snapshot_bytes_remaining, rpcid = our_rpcid) is used to search for it in the *Peer Tree*. Then, the object in the *Peer Tree* is removed and reinserted.
2. If the RPC is not present in the *Peer Tree*, the case is treated as a new RPC state.

Operations in XDP_GEN. In XDP_GEN, the key (treeid = 1, bytes_remaining = last_min, peerid = last_peerid + 1) is used to search the *Peer Tree*, where *last_min* and *last_peerid* represent the last minimum bytes_remaining and the last peerid used, respectively. They both start from zero. If the RPC has completed its grant, it is removed from the *Peer Tree*, and the resulting information is then used to search and remove the RPC from the *RPC Tree*.

```
/* Congestion Control */
/* Called at XDP_GEN */
void *bpf_dequeue_ack(struct bpf_map *map);
struct bpf_rb_node *bpf_dequeue_candidate(struct bpf_rb_root *root, bool (less)(struct
    bpf_rb_node *a, const struct bpf_rb_node *b));
/* Called at XDP */
int bpf_update_wnd(struct bpf_map *map, void *key, void *value);
int bpf_update_rate(struct bpf_map *map, void *key, void *value);
int bpf_enqueue_ack(struct bpf_map *map, void *ack_info);
int bpf_enqueue_candidate(struct bpf_rb_root *root, struct bpf_rb_node *n, bool
    (less)(struct bpf_rb_node *a, const struct bpf_rb_node *b);
/* Called at control path */
int cp_read_cc(int map_fd, void *key, void *value);
int cp_update_cc(int map_fd, void *key, void *value);

/* Pacing */
/* Called at XDP_EGRESS */
int bpf_tx_pacing_wnd(struct bpf_map *map, void *key);
int bpf_tx_pacing_rate(struct bpf_map *map, void *key);
struct xdp_frame *bpf_fetch_pkt(struct bpf_map *map, void *key);
int bpf_send_pkt(struct bpf_map *map, int ifindex, __u64 flags);
int bpf_flush_pkt(void);

/* Reliability */
/* Called at XDP */
int bpf_detect_loss(int (cb)(struct xdp_md *ctx, void *value));
/* Called at control path */
int cp_detect_loss(int (cb)(void *cc_ebpf, void *cc_cp);
```

Listing 1: Transport component hooks and corresponding APIs.

C Development Model

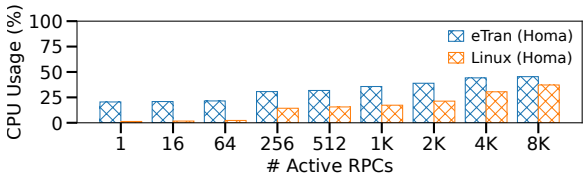
eTran envisions a flexible development model targeting different levels of developers. For transport developers familiar with eBPF programming and transport implementation such as timers and loss recovery, eTran provides a set of low-level APIs for them, as shown in Listing 1. The transport developers can quickly prototype customized transports by directly leveraging these APIs or based on our implementation of Homa and TCP (§4) and then safely deploy them in the Linux kernel. For application developers who focus on user-space application development, they can specify which transport (among eTran-supported ones) to use for their applications. For site reliability engineers and networking administrators, eTran further enables them to conveniently fine-tune the parameters or implementation (thus the performance) of these transports.

We note that our development model is conditioned on the new eBPF hooks and maps in eTran being upstreamed into the Linux kernel. Although this is partially out of the scope of our eTran research, we have tried to minimize changes to the existing kernel eBPF subsystems, thus being friendly to existing eBPF applications as far as we know. Another important factor that determines whether eTran can be upstreamed into the kernel is security, which we elaborate on in §5.

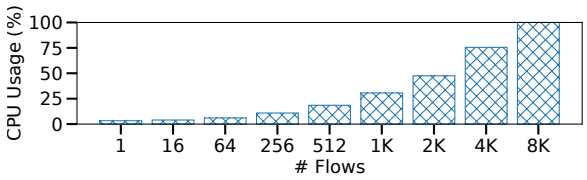
D Additional Design Drill-Down

D.1 CPU utilization of Control Path

We now examine the control path CPU utilization for Homa and TCP. In eTran (Homa), the control path uses a timer thread to check RPC states from the kernel. In eTran (TCP), it runs the CC algorithm and timeout detection. In this experiment, eTran (Homa) performs lookups with a batch size of 16, and eTran (TCP) employs a CC interval of 200μs. We vary the number of RPCs/flows and measure the CPU usage of the control path, the result is shown in Figure 12a and Figure 12b. Both eTran (Homa) and eTran (TCP) exhibit increased CPU usage as the number of RPCs/flows grows. Compared with Linux (Homa), eTran (Homa) incurs higher CPU overhead because it relies on system calls to look up transport states. For eTran (TCP), 8K connections can saturate one CPU core. The control path of eTran (TCP) can be scaled out by partitioning connections across multiple threads.

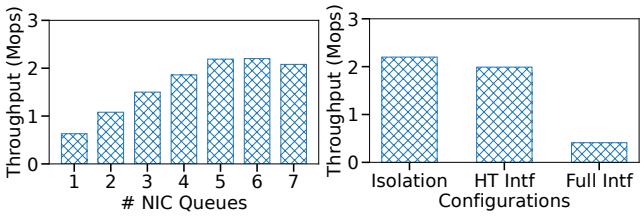


(a) eTran (Homa) vs. Linux (Homa)



(b) eTran (TCP)

Figure 12: CPU usage of eTran (Homa), Linux (Homa), and eTran (TCP), under different numbers of RPCs/flows.



(a) Impact of multi-queues. **(b)** Interrupt interference.

Figure 13: Throughput comparison under different configurations.

D.2 Impact of Multiple Queues and Interrupt Interference

We now examine whether eTran can scale with the NIC queues. We take TCP as an example, as Homa doesn't work well with RSS. We replicate the connection scalability experiment described in §6.2, fixing the number of CPU cores for the application while varying the number of NIC queues. 32K connections are separated across five clients. Figure 13a shows that eTran scales with NIC queues from 1 to 5, but beyond that, throughput slightly declines. This is due to multiple application threads requiring locked access to the Fill ring and Comp ring.

We are also interested in the performance of eTran when the application cores are collocated with the NAPI cores. We conduct the same experiment above with three configurations: 1) separate to different physical CPU cores, 2) separate to different logical CPU cores, and 3) combine on same logical CPU cores. Figure 13b shows the result. The performance of the second configuration is lower than the first mode due to Hyper-Threading (HT) interference. The third configuration exhibits poor performance due to the serve interference between application threads and NAPI [81]. Considering eTran offloads all transport states in eBPF, how to utilize eBPF for CPU scheduling [63] to optimize eTran is an interesting direction. We leave it as future work.

White-Boxing RDMA with Packet-Granular Software Control

Chenxingyu Zhao[1], Jaehong Min[1], Ming Liu[2], Arvind Krishnamurthy[1]
[1]University of Washington, [2]University of Wisconsin-Madison

Abstract

Driven by diverse workloads and deployments, numerous innovations emerge to customize RDMA transport, spanning congestion control, multi-tenant isolation, routing, and more. However, RDMA's hardware-offloading nature poses significant rigidity when landing these innovations. Prior workflows to deliver customizations have either waited for lengthy hardware iterations, developed bespoke hardware, or applied coarse-grained control over the black-box RDMA NIC. Despite considerable efforts, current customization workflows still lack flexibility, raw performance, and broad availability.

In this work, we advocate for *White-Boxing* RDMA, which provides control of the hardware transport to general-purpose software while preserving raw data path performance. To facilitate the white-boxing methodology, we design and implement *Software-Controlled RDMA (SCR)*, a framework enabling packet-granular software control over the hardware transport. To address challenges stemming from granular control over high-speed line rates, SCR employs effective control models, boosts the efficiency of subsystems within the framework, and leverages emerging hardware capabilities. We implement SCR on the latest Nvidia BlueField-3 equipped with Datapath Accelerators, delivering a spectrum of new customizations not present in legacy RDMA transport, such as Multi-Tenant Fair Scheduler, User-Defined Congestion Control, Receiver-Driven Flow Control, and Multi-Path Routing Selection. Furthermore, we demonstrate SCR's applicability for GPU-Direct and NVMe-oF RDMA with zero modifications to machine learning or storage code.

1 Introduction

Remote Direct Memory Access (RDMA) empowers data center workloads with high-performance networking thanks to hardware-offloaded transport. However, as deployments shift from single-tenant lossless fabric to multi-tenant lossy Ethernet and applications expand from HPC to storage and GPU communication, a one-size-fits-all approach to RDMA hardware transport is inadequate. Consequently, extensive innovations have emerged to customize RDMA transport. From the industry, Microsoft Hyperscale Deployment [1–3], Google Falcon [4], AWS SRD [5], Meta [6], Alibaba HPN [7], and the Ultra Ethernet Consortium (UEC) [8], among others, are driving RDMA transport customizations at scale. This is supplemented by academic contributions spanning topics such as congestion control [9–16], multi-tenant isolation [17–19], routing [20, 21], reliability [22–24], and more.

Although the innovations are in full swing, RDMA's hardware nature imposes significant rigidity on integrating these customizations. Fundamentally, ASIC hardware iterations are expensive both in terms of time and monetary cost. For instance, DCQCN congestion control [2] baked into ASICs by Microsoft and vendors of RDMA NIC (or RNIC) has persisted for nearly a decade. Despite subsequent congestion control innovations, the hardware rigidity has hindered the integration of new ideas. Some cloud providers have explored bespoke hardware or FPGA RNICs [25, 26], but these solutions still require substantial efforts and lack widespread availability compared to commercial RNICs. Alternatively, software-based approaches [18, 24, 27, 28] treat RNIC hardware transport as a black box, enforcing customizations through upper-layer mediators. However, these software overlay mediators are coarse-grained at the message/verb granularity (up to GB in size) instead of the transport-layer packet granularity (typically 1024B) and introduce additional processing overhead, leading to latency penalties and security concerns [17].

Inspired by the successful practice of white-boxing switches [29–34], we propose *white-boxing* RDMA, a new methodology to customize RDMA hardware transport. Historically, switch vendors tightly integrated control logic with hardware data planes, limiting flexibility for user-defined control. With white-box switches, control planes run in customizable software while preserving the data plane in high-performance hardware, sparking a series of SDN innovations over the last decades. Similarly, we advocate the same methodology for the current RNIC; we expect to take transport control from the hardware into software for flexible customization while keeping the data path in performant hardware.

Besides the need for customization, another key enabler for white-boxing RDMA is the advancements of SmartNICs [35–40]. SmartNICs provide two building blocks for white-boxing RDMA: computational power and control interfaces. Taking the latest Nvidia BlueField-3 (BF3) as one example, it interposes general-purpose Datapath Accerlators (DPA) in line with the data path, potentially supporting flexible control logic without incurring host-side CPU overhead. Moreover, DPA features heavy multi-threading capabilities to keep pace with line-rate processing. DPA also leverages low-profile RISC-V designs to achieve energy and cost efficiency. These advancements pave the way for white-boxing RDMA.

Realizing the vision of white-boxing RDMA poses two main challenges: flexibility and efficiency. In terms of flexibility, we aim to accommodate diverse control functionalities that rely on various control signals, versatile control laws, and control actions. As for efficiency, fine-grained control over RDMA transport encounters challenges in keeping pace with high-speed line rates. For instance, at 100 GbE with a

1500B MTU, achieving 8.3 Mpps generates packet-granular signals (such as ACK and TX) at the sub-μs level (120 ns!). Processing signals at this granularity demands high efficiency, as even single access to DDR memory (hundreds of ns) entails a relatively significant time cost. Therefore, boosting processing efficiency is essential.

In this work, we introduce a control framework called *Software-Controlled RDMA (SCR)*. Drawing from classical control theory [41–43], SCR centers on two fundamental questions: *What to Control* and *How to Control*. Addressing the first question, we propose the *Dequeue Rate* control model, which regulates the rate at which the RNIC fetches messages from memory regions associated with queue pairs. This control knob serves diverse purposes across local-host, fabric, and peer-receiver domains (detailed in §4.2). As for *How to Control*, the SCR framework comprises three subsystems: signal/event collector, event queues, and event processor. These subsystems are carefully designed to tackle the challenges of flexibility and efficiency. The event collector aims to gather comprehensive signals from various domains, while event queues enhance efficiency through coalescing signals. For event processors, we build a Deterministic Multi-Threading (DMT) system to achieve high-efficiency and general-purpose processing (detailed in §4.3).

We prototype SCR on the latest Nvidia BlueField-3 SmartNIC. Specifically, we leverage BF3 DPA's *central* architecture position: DPA can interact with RNIC hardware, communicate with the host CPU, and inject traffic into the fabric. Moreover, implementing SCR using the DPA minimizes the host CPU's computational burden and operates in a *non-intrusive* manner for host-side applications and libraries. We successfully deliver a spectrum of new functionalities not present in legacy RDMA transport: 1) Achieving fair multi-tenant QP scheduling (up to 1.78x improvement in fairness index compared to the default scheduler; up to 52.8% latency reduction); 2) Enabling user-defined congestion control (Swift [10]); 3) Enabling multi-path routing selection (MP-RDMA [20]; PLB [44]); 4) Enabling receiver-driven control (Homa [15]). Importantly, we demonstrate that storage scenarios using NVMe-oF RDMA and machine learning using GPU-Direct RDMA can benefit from these customizations without any code changes to applications and libraries.

2 Background and Motivation

2.1 Why Customize RDMA

We first comprehensively summarize prior efforts in customizing RDMA transport and then identify the implications (*I*) for an RDMA customization framework.

• Better Congestion Control: To achieve lossless fabric, RoCEv2 employs PFC, which, despite its effectiveness, faces issues like PFC storms, HoL blocking, and deadlocks [3,45,46]. Mitigating these drawbacks, numerous congestion control algorithms have been developed, including Microsoft's ECN-based DCQCN [2], Alibaba's INT-based HPCC [9], Google's RTT-based Timely [47] and Swift [10], and ACK-Based ACC [11], showcasing how to customize RDMA congestion control, especially for hyper-scale lossy Ethernet. *I1: New types of signals from the transport layer enable improved forms of congestion control laws.*

• Stronger Multi-Tenancy Performance Isolation: RDMA transport bypasses in-kernel multi-tenant isolation mechanisms, resulting in inadequate multi-tenancy isolation within cloud environments. Studies such as Husky [19,48], and Harmonic [17] have demonstrated a pressing need for improved performance isolation among multi-tenants sharing RNIC. Solutions like Flor [24], X-RDMA [49], and Justitia [18] introduce a software mediation layer between the tenant and the RNIC driver, aiming to improve multi-tenant isolation. *I2: New measurements reveal the need to control new domains like RNIC micro-architecture and multi-tenant host.*

• Receiver-driven Flow Control: The many-to-one traffic pattern, common in GPU collective communications [50] and storage IO [1], often leads to incast congestion at the responder. This challenge is intensified by RDMA's one-sided operations, which are agnostic to responder-side controls. RCC [51] proposes receiver-driven congestion control for RDMA, with EQDS [16], Homa [15], NDP [52], and pHost [53] advocating for a receiver-driven control loop. *I3: Emerging workload patterns motivate new types of flow controls, such as receiver-driven flow control.*

• Per-Packet Multi-Path Routing: RoCEv2's default single-path routing ensures in-order packet delivery but fails to utilize the fabric's multiple paths. MP-RDMA [20] introduces mechanisms for multi-path routing. AWS SRD [5] emphasizes multi-path routing benefits for RDMA in the production environment. And switches such as Nvidia Spectrum [54] and Broadcom Tomahawk [55] support adaptive routing (per-packet ECMP). *I4: New types of routing controls like multi-pathing are beneficial for RDMA transport.*

RDMA transport customization is vast in the literature, and we present additional studies in Appendix Table 6.

2.2 How Prior Works Deliver Customization

RNIC offloads control functions onto rigid ASIC hardware, making reprogramming most customizations difficult. We compare prior efforts to deliver customizations as follows:

1) One approach is collaborating with RNIC vendors to integrate customizations into next-gen ASICs, such as Microsoft/Mellanox incorporating DCQCN into ConnectX. However, this method relies on hardware generation cycles, which typically span years.

2) Developing a bespoke RNIC from scratch using FPGA or ASIC. FPGA-based solution [17, 20, 26, 56, 57], despite significant engineering efforts, still generally exhibit discrepancies compared to native commercial RNIC ASICs, which have evolved over almost a decade to become a commodity. Cloud providers like AWS SRD [5], Google 1RMA [25], and

Falcon [4] can tape out custom ASICs, yet requiring substantial costs and expertise, and they are primarily utilized for internal operations rather than as broad commodities.

3) Software overlay solutions [16, 18, 24, 28, 58] provide userspace libraries over RNIC drivers. However, host software is limited to operating at Verb-level (*i.e.,* message level) and still treats the hardware transport as a black box to work around. Userspace libraries are not transparent to the host software stack because they customize drivers or insert mediators between the RNIC and applications. This added software mediator inevitably incurs a latency penalty [18, 24, 28], consuming host CPU cycles, and could raise security concerns for cloud tenants [17].

4) Fully software emulation like SoftRoCE [59] lacks the crucial property of hardware-accelerated data paths.

2.3 Our Goals

Flexible and Packet-Granular Control: In delivering RDMA customization, software agility surpasses hardware rigidity. Thus, we aim to leverage flexible software to enable diverse control functionalities. We aim to enforce packet-granular control at the hardware transport, a lower layer than userspace message-granular coarse-grained control.

Raw High-Performance Data Plane: High performance is crucial for RDMA; otherwise, people could opt for fully programmable transport like in-kernel TCP/IP. Thus, we adhere to the principle of *Zero-Intrusion* (or *Zero-Touch*) for the hardware data plane to maintain its original high performance.

Deployment Ready: We utilize commodity hardware to provide ready-to-deploy solutions and adhere to the principle of *smart-edge-dumb-core*, placing complex functionalities on end-host devices while keeping in-network switches simple. Next, we explore the capabilities of commodity hardware.

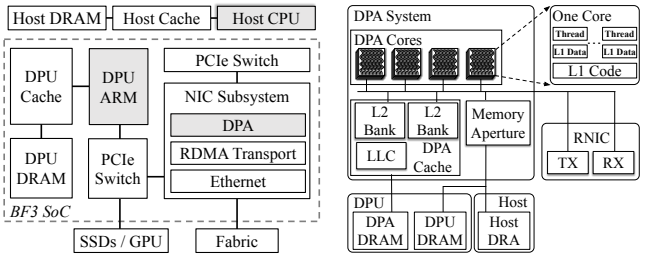
3 Characterizing Commodity Hardware

3.1 Requirements of Control System

Before characterizing existing hardware, we first examine the capabilities required for an ideal control system. Drawing from classic control theory's signal-driven feedback-loop [41–43], we identify three key components for an ideal control system: *signal collectors*, *processors*, and *action executor*. To achieve our design goals, these three components necessitate specific hardware capabilities. These requirements drive our following characterization.

3.2 Existing Hardware Model

As shown in Figure 1a, with the advancement of SmartNICs [35, 37, 38, 62], several processors within one server can be potentially utilized to implement the control system. Taking the latest BlueField-3 as an example, we have three options: Host CPU, DPU-embedded ARM CPU (off-path cores), or Datapath Accelerators (on-path cores). Next, we explore their potential capabilities and identify a suitable candidate.



(a) System Model (b) DPA Architecture

Figure 1: **System Model and DPA Architecture. Referring to Public Information Provided by the Vendor [60, 61].**

	Host CPU	DPU ARM	DPA
ISA	x86 Xeon	Arm v8.2+	RISC-V
Cores	64 Cores	16 Cores	16 Cores
Threads	128 Threads	16 Threads	256 Threads
Memory	384GB DDR4	32GB DDR5	Local 1GB DDR5 + Load/Store Access to Host/SoC DRAM
L1d Cache	1.5MB*32	64KB*16	1KB*256
L1i Cache	1MB*32	64KB*16	1KB*16
L2 Cache	40MB*32	0.5MB*16	1.5MB*1
L3 Cache	72MB*2	16MB*1	3MB*1
Network	General	General	Raw Ethernet
Interaction with RNIC	Verb-Granular	Verb-Granular	Packet-Granular
OS	Linux	Linux	Real-Time OS
Requirements Satisfied	**R1/R3/R4**	**R1*/R3**	**R1*/R2/R3/R4**

(* denotes partial satisfying)

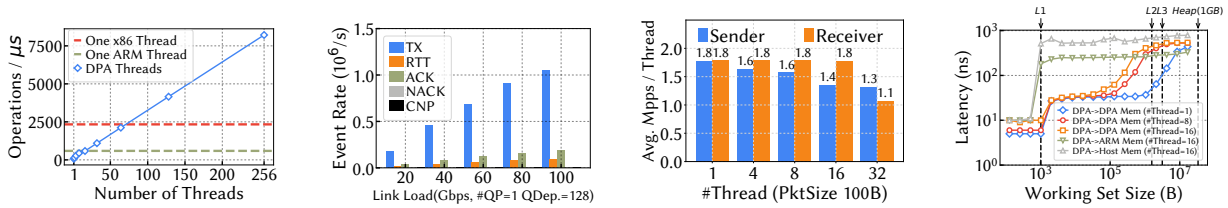
Table 1: **Comparing Heterogeneous Processors.**

Testbed: Our testbed (used throughout the paper) consists of two servers connected to a 100GbE switch. Each server has dual Intel Xeon Gold 6346 processors, 384GB DDR4, PCIe Gen4, and an Nvidia BlueField-3 DPU with 100GbE cables. Each server is equipped with one Nvidia A30 GPU (24GB HBM2, 933GB/s memory bandwidth) with driver-550 and CUDA v12.4. The host machines run Ubuntu 22.04 with kernel 6.5.0, DOCA SDK v2.6, and SPDK v24.05. We configure SR-IOV VFs for multi-tenant support.

3.3 Characterizing Datapath Accelerators and Others

As Table 1 shows, we characterize the Datapath Accelerator (DPA) and compare it with Host CPUs and DPU ARM cores. We focus on micro-benchmarking the DPA because extensive research has already been conducted on x86/ARM cores [39, 63–66]. Chen et al. [63] characterizes the DPA from the architectural perspective. We investigate DPA functionalities relevant to the control requirements mentioned, for example, the fine-grained interaction between RNIC and DPA (*e.g.,* Programmable Congestion Control), not covered in [63].

DPA Overview: DPA is a general-purpose on-datapath co-processor. Figure 1a illustrates the conceptual position of the DPA within the BF3 overall architecture, and Figure 1b provides a detailed view of the DPA's micro-architecture. As Table 1 lists, DPA comprises 256 hardware threads and possesses its own cache and memory system. DPA can access the host memory and DPU on-SoC memory with load/store



(a) Multi-Thread Execution (b) PCC Events (WRITE Verb) (c) L2 Send/Receive (d) Memory Access Latency

Figure 2: **Characterizing Hardware Capabilities of Datapath Accelerators (DPA).**

instructions (*i.e.,* via memory aperture [60]). DPA can operate the RNIC, which is co-packaged on the same PCIe card.

For R1, Multi-threading Performance: DPA features heavily multi-threaded computation with 256 hardware threads (even more than the host's beefy x86 CPU). In Figure 2a, we conduct General Matrix Multiplications (GEMM) to assess scalability while increasing the number of threads (Each thread computes the tiled outer product of GEMM [67]). Each operation represents one multiplication. The scalability trend closely approaches a linear relationship.

Notably, DPA cores in low-profile and low-cost RISC-V are much less powerful than x86/ARM cores. As shown in Figure 2a, it may require up to 16 DPA threads to achieve equivalent processing capability as one ARM thread. Therefore, tailoring the functionality over a single DPA thread, which involves a low amount of code and lightweight operations, becomes necessary. Although DPA offers 256 threads, given these low-code constraints, we believe that the wimpy cores partially satisfy the R1 requirement.

For R2, Interaction with RNIC: The RNIC offers the Programmable Congestion Control module [68] for the DPA, facilitating fine-grained interactions between the RNIC and DPA. These include the DPA receiving per-packet signals (TX/ACK/NACK/CNP, detailed definitions in §4.3.1), measuring RTT by injecting the RTT request packet into the RNIC, and the RNIC providing APIs for the DPA to set per-flow rates. In Figure 2b, we measure the event rates while varying the link load. It is evident that the DPA can receive packet-granular events. Note that NACK (packet loss) and CNP (congestion) events are rare; hence, the bars representing them are small.

For R3, Raw Ethernet Capability: Since we need to send and receive small-sized control signals over fabric, we test DPA's communication APIs for transmitting data. These APIs operate at the raw L2 Ethernet level, allowing for flexible customization of L3/L4 protocols such as TCP/IP or UDP/IP. Figure 2c illustrates the performance of L2 sending and receiving. For a single thread, it achieves packet rates of about 1.8 Mpps, meeting the requirements for μs-scale communication. As the number of threads increases, single-thread performance degrades due to contention, but the total Mpps increases. Since control signal packets are generally small, we focus on Mpps for small packets, such as 100B.

For R4, Memory Access Latency: Signal collectors interact with the host ideally via memory access. In Figure 2d, we measure the latency for DPA accessing the local cache, local memory, DPU on-SoC DRAM, and host-side DRAM. On workloads, we use the memory bandwidth test from *lmbench* [69], a classical micro-benchmark suite. We highlight several takeaways: DPA accessing L1 is significantly faster than accessing the DDR memory; DPA accessing local memory is faster than accessing the DPU memory and host memory; DPA L1 and L2 are precious and limited resources shared by multiple threads. Thus, during the design and implementation, we should keep the stateful memory (*e.g.,* per-flow states) compact and reduce memory access times.

We discuss the host CPU's and DPU ARM's limitations in meeting the above four requirements in Appendix §A.1. Compared to DPA, the host CPU and DPU ARM lack a control interface for requirement R2 and face performance issues for the other requirements.

Overall, we conclude that DPA is a suitable but not perfect candidate among three. We must address challenges like wimpy core constraints on low-code execution and limit states maintained to improve cache locality. Importantly, for generality, we loosely couple our design with any specific hardware. If DPA's APIs are exposed to host CPU and ARM cores, we can adapt our design to be compatible with them.

4 Software Control RDMA Framework Design

4.1 Overview

The key questions for Software-Controlled RDMA (SCR) are *What to Control* and *How to Control*. Firstly, we introduce a control model that forms the foundation of SCR to address the first question (§4.2). Next, addressing the second question relies on the event-driven rate computation framework (§4.3).

4.2 Dequeue Rate Control Model

QP Scheduler Model: We first examine RNIC's QP scheduler, which serves as the infrastructure for controlling network traffic at the sender side. Figure 3 illustrates the behavior of the RNIC QP scheduler, also functioning as the rate limiter: When tenants post WQEs to the SQ, the host driver notifies the RNIC via doorbell to update QP context (QPC), and the RNIC fetches several WQEs to aid in scheduling decisions. The scheduler leverages per-QP metadata, such as QPC and WQE, along with per-QP rates, to determine the execution order of WQEs. Typically, these per-QP rates are allocated by the congestion control module, such as DCQCN. For simplic-

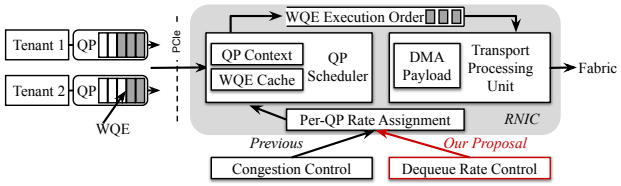


Figure 3: **RDMA NIC Queue Pair Scheduling Model.**

ity, we focus solely on the scheduler's input (metadata and rate assignment) and output (WQE execution order), while the internal structures of schedulers fall outside the scope of our work. Rich literature investigates the design of schedulers and rate limiters [70–72] (Tassel [70] specifically for RNIC). After the scheduler generates the WQE execution orders, the Processing Units of the Transport Logic (*e.g.,* RoCEv2) dequeue the WQE, DMA the payload from host memory, and complete packetization before sending it out over the wire.

	Standalone Metric	Competing with BW-Hungry	BW-Hungry QP (64KB Msg.)
Th.-Sensitive (4KB Msg.)	88.30 Gbps	33.25 Gbps	57.64 Gbps
Lat-Sensitive (256B Msg.)	2.44 us	5.31 us	90.57 Gbps

Table 2: **Limitations of DCQCN-Only Rate Assignment.**

Although the current RNIC QP scheduler delegates the per-QP rate assignment to fabric congestion control like DCQCN, unfortunately, relying solely on fabric congestion to control the sending rate has limitations. For instance, as shown in Table 2, consider a scenario where tenants compete with a bandwidth-hungry QP. We observe that the throughput-sensitive tenant experiences significant degradation and unfairness in throughput while the latency-sensitive QP suffers from increased latency. A desired behavior is that the throughput-sensitive QP can fairly share bandwidth with the bandwidth-hungry QP, while the latency-sensitive QP experiences no latency increase when competing with the bandwidth-hungry QP. It is worth noting that their aggregate bandwidth is at line rate, indicating no congestion in the fabric, so DCQCN does not react accordingly. Many prior works also report similar inadequacies in RDMA congestion control for multi-tenant scenarios [17, 18, 24, 28, 73].

Dequeue Rate: We observe that the input of the QP scheduler is a powerful control knob that can go beyond the congestion control's sending rate. As Figure 3 shows, we leverage this input knob as the per-QP *Dequeue Rate*, reflecting the rate at which the RNIC retrieves messages from memory regions. For instance, in WRITE verbs, the WQEs dequeued from the SQ indicate the message size per request, and then the dequeue rate is determined by the total message sizes over a given time period. The choice of the dequeue rate originates from its beneficial *multiplicity* as follows.

Multiplicity of Dequeue Rate: As Figure 4 shows, the dequeue rate influences multiple aspects of RDMA transport: 1) Primarily, it impacts the sending rate, as messages can only

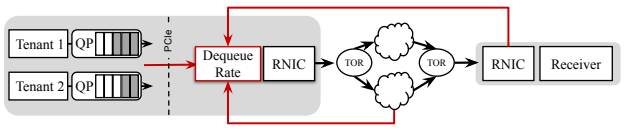


Figure 4: **Dequeue Rate Affects Multiple Domains.**

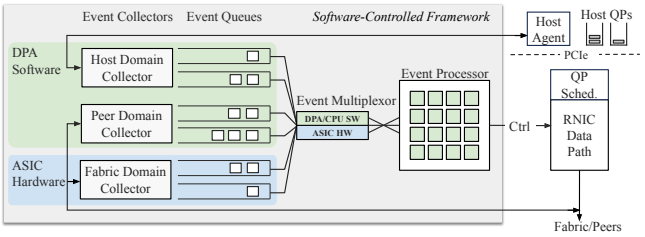


Figure 5: **Event-Driven Rate Computation Framework.**

be sent out by the RNIC hardware once dequeued from the host; 2) It affects PCIe utilization because a higher dequeue rate leads to increased PCIe utilization for DMA transfers; 3) The dequeue rate determines resource allocation within RNIC hardware, such as ASIC Processing Units, since dequeue operations initiate subsequent hardware processes like packetization; 4) The dequeue rate can respond to back pressure from the receiver side, as slowing down the sender's dequeue rate can alleviate pressure on the receiver side. In Appendix §A.2, we empirically evaluate how the dequeue rate influences various aspects mentioned above. Overall, the dequeue rate offers a comprehensive control, making it a suitable control knob for SCR.

4.3 Event-Driven Rate Computation Framework

With the dequeue rate control model, we next design an event-driven framework to collect multi-domain signals, compute the rate, and output the control actions. As Figure 5 shows the workflow, we next describe the subsystems of the framework.

4.3.1 Event Collectors Subsystem

Design Goals: As prior literature shows, the key enabler for flexible control is collecting accurate and diverse signals from multiple domains (*i.e.,* signal sources). Therefore, the primary objective of event collector systems is to be *all inclusive*, capable of providing a wide range of events. Furthermore, while we strive to collect more signal types, we prioritize *deployment-friendly* solutions that utilize existing features of NICs and switches rather than requiring specialized devices. Next, we outline how to achieve these design goals:

Multiple Domains: To control the dequeue rate, we gather signals from three primary domains: *Fabric Domain*, *Host Domain*, and *Peer Domain*. Table 3 lists the specific signals collected from each domain. These domains employ distinct mechanisms to gather events as follows:

Fabric Domain: We employ two types of collecting mechanisms: 1) In-Band Mechanisms: In-band packets carry signals such as CNP/ACK/NACK. Additionally, the TX/RX pipe of the RNIC can generate TX/RX events during network trans-

Domain	Signal	Definition
Fabric Domain	TX	Completion of transmitting a burst of packets, including timestamp and total byte count
	RX	Completion of receiving a burst of packets, including timestamp and total byte count
	CNP	Congestion Notification Packet (*e.g.,* ECN)
	ACK	Acknowledgements
	NACK	Negative Acknowledgements
	RTT	Measurements of Round Trip Time
Host Domain	App-Hints	Application-specific policies (*e.g.,* Fair bandwidth sharing among tenants)
	PCIe Util.	PCIe real-time bandwidth and latency
	RNIC Util.	RNIC hardware resource utilization
Peer Domain	Receiver Credits	Rate at which peer can handle incoming traffic
	Peer PCIe Util.	Peer node's PCIe bandwidth and latency
	Peer RNIC Util.	Peer RNIC's resource utilization

Table 3: **Signals From Multiple Domains. This list includes typical signals but can be expanded as needed.**

fer. RTT can also be obtained by timestamping the in-band packets. Some commodity RNICs, such as the BF3 used in this project, already support the collection of in-band signals such as CNP/ACK/NACK/TX/RTT. To our knowledge, the current BF3 does not support RX signal collection. As an alternative, we enable RX signal collection through host domain collectors, which will be discussed later. 2) Out-of-Band Mechanisms: Out-of-band packets probe network status in a telemetry manner. For probing packets, we implement the sender and receiver using the BF3 DPA, which supports communication over the fabric as shown in §3.3. We design a UDP-based protocol for transferring out-of-band messages over raw Ethernet APIs. Several reasons support the choice of UDP-based protocols: RoCEv2 is based on UDP, allowing one to track the header field of the RDMA in-band connection and reuse the meta info for associated out-of-band transfers. Additionally, UDP alleviates the burden of maintaining connections and handling reliability. An example of out-of-band probing is RTT measurements. The sender initiates out-of-band RTT request packets that are IP-routable. Upon the receiver NIC's ping-pong response to the RTT request packets, the sender can calculate the RTT. Unlike in-band RTT measurements with static per-ACK granularity, out-of-band RTT offers flexibility in the frequency of sending out RTT request packets. Also, out-of-band RTT carried as payloads is more flexible than in-band RTT, avoiding packet header format modifications.

Host Domain: We design several methods to collect host-domain signals: 1) For application hints, the event collector provides a shareable memory region that applications can access using load/store instructions, allowing them to write signals with sub-μs latency. Specifically, we implement the collector using two components: a lightweight agent on the host CPU and a process on the DPA. The host agent collects application hints and writes them to a shared memory region, from which the DPA process reads the signals. 2) For PCIe utilization measurement, we employ two methods: The first method involves the host-side agent utilizing tools like Intel PCM [74] to query real-time stats, followed by writing these stats to the event collector via the shared memory region. The second method entails implementing a proactive approach using PCIe endpoints like NIC to detect PCIe utilization. Specifically, the DPA process initiates a ping, to which the host agent responds with a pong, allowing the DPA to analyze PCIe latency. Importantly, both the host agent and the DPA process are configured per node rather than per flow, ensuring there are no scalability issues. This implementation is efficient because all flows within the same server typically share the same RNIC, host, and PCIe interconnect. 3) For RNIC utilization, we can utilize the RNIC hardware's query API to obtain real-time status. Also, we analyze the fabric domain signals to infer current RNIC utilization, such as the number of active connections.

Peer Domain: Signals of this domain must traverse the network fabric between the requester/responder. Similar to the fabric domain's out-of-band mechanism, we operate raw Ethernet packets to send/receive signals between local/peer nodes. We employ a UDP-based protocol to ensure the signal is routable while eliminating unnecessary heavy-weight mechanisms like connection setup and reliability. This approach achieves low latency and low processing overhead. Peers initially gather node information, including application hints, PCIe, and RNIC utilization, using the method outlined above for the host domain. These signals can serve as credits from the receiver to throttle the sender's dequeue rate. Once peers have gathered their information, they can transfer it using the lightweight UDP-based protocol. In our practice, using raw UDP with 128 Bytes payloads is sufficient to encapsulate essential information like flow ID and credits. We implement the sender and receiver using the BF3 DPA on both local and peer nodes. The sender injects packets into the network in several modes: periodically, sampling, and based on anomaly triggers. In the periodic approach, we set the packet interval to one RTT, balancing timely information delivery with moderate network usage. Considering that one DPA thread can handle 1.8+ Mpps, as detailed in §3.3, we can consolidate senders for multiple QPs into a single thread. With a typical RTT setting of 20 μs [75], a single thread can support up to 36 flows. In the sampling approach, the sender transmits out-of-band packets only after a specified number of in-band bytes have been sent. The anomaly-triggered method conserves network bandwidth by activating only when significant changes are detected, such as changes in the PCIe or port utilization.

4.3.2 Event Queues and Multiplexer Subsystem

Motivating Queuing and Coalescing: Event queues temporally store events[1] from the collectors before the multiplexer dequeues events and invokes the processor to handle them. The event queue subsystem is to facilitate *event coalescing*, a crucial mechanism for managing high-speed packet-granular signals. In Figure 6, we motivate the coalescing mechanism using the TX events example. Three factors contribute to

[1]Events encapsulate signals and their associated metadata such as flowID. We use the terms of events and signals interchangeably in the paper.

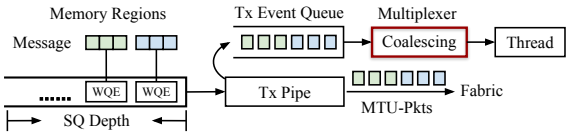


Figure 6: **Coalescing Mechanism (TX Events Example).**

the rapid accumulation of per-packet TX events: Firstly, in RDMA operations such as WRITE, each WQE can indicate a large message size (up to GB), while the wire has a typically small and fixed MTU size (*e.g.,* 1024B). Therefore, a message associated with a single WQE could be segmented into a large number of MTU-sized packets. Secondly, for SQ, users often configure a large queue depth to enhance utilization through batching, further speeding up the accumulation of per-packet signals. Thirdly, depending on the complexity of control laws, the processor may operate slowly. This speed may be outpaced by the event arrival rate, resulting in event queue buildup. If we process each per-packet event sequentially, the event queue will quickly fill up. Thus, coalescing multiple consecutive events into one event is a promising solution to improve processing efficiency. We micro-benchmark the effectiveness of coalescing in Appendix §A.3. Importantly, in addition to the TX signals from the fabric domain, signals from other domains can also benefit from coalescing, driven by similar motivations.

Design Goals: According to the above observations, the queuing system aims at 1) Supporting efficient and flexible coalescing strategies; 2) Applying to various types of signals. To achieve these two goals, we not only utilize existing functionalities in ASIC hardware but also introduce new functionalities in DPA software, as detailed below.

Hardware Coalescing/Multiplexing: As described in §4.3.1, event collectors for in-band signals, such as TX, ACK, and CNP events, are ASIC hardware modules positioned with the RNIC TX/RX pipelines. In addition to the event collectors, the current BF3 ASIC hardware also supports coalescing these in-band signals and subsequently invoking DPA threads for processing. However, to our knowledge, the current hardware coalescing and multiplexing approach has two main limitations: 1) Applicability: It is limited to in-band signals from the fabric domain, excluding other signal types from domains such as Host or Peer Domains, which cannot leverage these hardware circuits; 2) Flexibility: The hardware supports only fixed strategies for coalescing and multiplexing. Specifically, for coalescing, it only supports an *Accumulation* strategy, which accumulates values indicated by each event and outputs a single summary event. For multiplexing, it periodically invokes a thread based on hashing results of the flow ID associated with the events. Critically, it does not provide a programming interface for users to define more versatile coalescing and scheduling strategies. To address these shortcomings, we enhance the applicability and flexibility of the queuing system with software implementations using DPA.

Software Coalescing: As we previously described in §4.3.1, for out-of-band signals and host domain signals, we employ software-based collectors over the DPA cores (Figure 5 denoting DPA Software). These types of signals can not benefit from hardware coalescing. Consequently, the event coalescing and multiplexing for these out-of-band signals are software-based implementations over DPA. Specifically, we first allocate a queue for each type of event for each flow, simplifying the logic without considering cross-flow and cross-type dependencies. We adopt the single-producer and single-consumer model for these queues to ensure lock-free operation. These queues are allocated using the memory resources accessible to the DPA process. Unlike hardware coalescing, which is limited to the *Accumulation* strategy, software coalescing offers the flexibility to implement additional strategies through software modifications. In addition to basic strategies such as *Accumulation*, *Keep-Latest-Event*, and *Keep-Oldest-Event*, we support more complex strategies. For instance, when flow rates stabilize (*e.g.,* converge after a slow start), continuing to process certain fabric events like TX could be wasteful for processor cycles. Instead, we coalesce multiple TX events into one processing event, saving unnecessary processing cycles. Specifically, when the hardware multiplexer hands over TX events to the DPA software processing thread, we early terminate processing and enqueue events. Full processing executes only if we accumulate enough signals in the event queue. Such complex strategies are not supported by current hardware coalescing.

Software Multiplexing: The multiplexer is responsible for dequeuing events and invoking the DPA thread to process. Ideally, we aim to consolidate all functions—such as enqueuing, coalescing, dequeuing, and invoking—onto the DPA cores to free processing cycles of the host CPU or DPU ARM. However, to our knowledge, DPA operates a specialized real-time OS (RTOS) with limited capabilities for thread and process management, leading to two main issues:

1) The DPA process cannot create threads during runtime (*e.g.,* utilizing POSIX thread system calls). Consequently, we rely on the host CPU or DPU ARM to pre-launch a process with a pool of threads (we discuss the pool size parameter in §4.3.3). Within this pool, specific threads are dedicated for collecting events (*e.g.,* sending/receiving out-of-band packets), while others are dedicated to processing. Event queues are stored in process-global memory accessible to both collectors for enqueuing and processor threads for dequeuing.

2) DPA's inter-process communication (IPC) capabilities are limited. The current BF3 DPA allows only a specialized process (*i.e.,* doca_pcc) to set flow rates. Therefore, other processes handling host-domain and peer-domain events need to communicate with the doca_pcc process to adjust flow rates. However, the DPA lacks typical IPC mechanisms like Linux pipes or domain sockets. Instead, we use the host CPU or DPU ARM as an agent, facilitating memory sharing with processes handling host-domain and peer-domain events. Furthermore, the host CPU (or DPU ARM) utilizes a Mailbox Mechanism

[68] to activate doca_pcc for setting the flow rate. Importantly, most functionalities still operate on the DPA, with the host CPU merely handling lightweight operations to compensate for the current limitations of the DPA RTOS.

4.3.3 Event Processor Subsystem

Design Goals: The event processor subsystem is to execute user-defined control laws. The primary goal for the event processor is computational efficiency, given that the per-packet signal interval could be as short as a sub-μs. Inspired by classical OS and architecture techniques [76–78], we aim at building a *Deterministic Multi-Threading* (DMT) model. Next, we propose several mechanisms to realize such a model on a multi-core processor of the BF3 DPA.

Run-to-Completion and Shared-Nothing: To ensure deterministic execution, computation adheres to the run-to-completion manner without context switching or preemptive scheduling. Given the packet-granular signals at the sub-μs level, context switching becomes prohibitively expensive. By default, we handle event processing fairly for all event types without prioritization. Thus, when the multiplexer schedules an event, it waits until the target thread is idle before invoking it rather than preempting. Furthermore, processors operate based on the shared-nothing principle, avoiding inter-thread locking or synchronization. Specifically, the multiplexer maps each event to only one thread and assigns the same type of event from the same flow to the same thread. Each thread holds its partition of global stateful records, ensuring the thread remains independent of others.

Scalable Thread Pool: We allocate hardware threads into separate pools corresponding to different purposes, such as handling fabric events or peer-domain events. Under the run-to-completion model , where threads are not dynamically created during runtime, the pre-determined size of the thread pool is a critical parameter. Considering scalability, we carefully determine the appropriate thread pool sizes for two primary types of event processing:

1) For per-node processing, the number of processing threads does not need to scale with the number of QPs (flows). Some types of signals, such as those from host domains, can be processed per node without the need for per-flow processing. As described in §4.3.1, host domain signals like PCIe utilization are shareable to all QPs over the same RNIC, as they share the same PCIe interconnect. Consequently, there is no need to establish per-QP threads to process these signals. Instead, we configure a limited number of threads (generally fewer than 10) dedicated to each host signal such as PCIe Utilization, Application Hint, and RNIC utilization;

2) For per-flow processing, it intuitively seems that the number of processing threads should scale with the number of flows. However, we introduce an observation indicating that such scaling is *not* always necessary for our thread steering policy as follows:

Steering Policy: For signals requiring per-flow processing such as in-band fabric signals TX/ACK, out-of-band fabric signals RTT, and peer signals, we pre-determine the thread pool size before launching the DPA process. For a small number of QPs (*e.g.,* fewer than thread pool size), the subscription ratio of threads to QPs is fixed (*e.g.,* 1:1). For a large number of QPs exceeding the thread pool size, signal processing for multiple QPs is consolidated into a single thread.

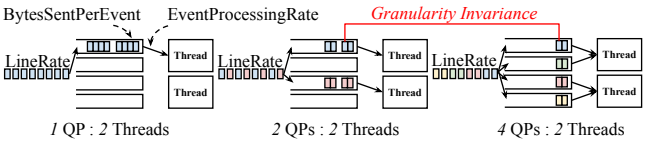


Figure 7: **Example of the Granularity Invariance Principle. While fixing two threads, the coalescence granularity remains invariant while increasing the number of flows from two to four.**

Processing Fidelity: Surprisingly, consolidating signal processing for multiple QPs does not affect processing fidelity. To measure fidelity, we introduce the metric *coalescing granularity*, defined as the number of bytes sent out per event. Recalling §4.3.2, when the event queue builds up, multiple per-packet signals are coalesced into a single event. Consequently, a single event representing more bytes indicates more events awaiting processing. Coalescing granularity, therefore, serves as an effective metric for assessing the waiting time due to contention faced by the processing thread. A smaller granularity indicates finer control and improved fidelity. Next, we introduce a theoretical principle suggesting that consolidating the processing of multiple QPs into a single thread will not compromise fidelity:

Granularity Invariance Principle: For N flows saturating a fixed line rate ($LineRate$), when N exceeds the number of processing threads T, the coalescing granularity for each flow remains *invariant* even though increasing the number of flows (Figure 7 illustrates an example).

Proof: The principle holds because the line rate is invariant to the number of flows. Consider a processing system capable of handling packet-granular events without dropping events for any sent bytes. Let $BytesSentPerEvent_i$ be the number of bytes each flow i sends per event, and $EventRate_i$ the rate at which events are processed. Assuming a fixed number of processing threads T, uniform coalescing strategy for all flows ($BytesSentPerEvent_i = Bspe$ for all i), and equal sharing of processing capability among the flows:

$$LineRate = \sum_{i=1}^{N} BytesSentPerEvent_i \times EventRate_i$$
$$= Bspe \times \sum_{i=1}^{N} EventRate_i$$

Since $\sum_{i=1}^{N} EventRate_i$ is a constant C (total event processing capacity shared among flows), we have: $LineRate = Bspe \times C$. Given C and $LineRate$ are invariant with respect to N (fixed T), $Bspe$ remains constant even as N increases. Hence, the coalescing granularity for each flow does not change.

The *Granularity Invariance Principle* provides scalability guarantees for per-flow processing utilizing the fixed number of threads, which applies to in-band fabric signals, out-of-band fabric signals, and peer signals. To achieve a reasonable coalescing granularity (*Bspe*), we typically launch 64 threads for processing in-band fabric signals. We empirically demonstrate its effectiveness and scalability in §6.1. For out-of-band fabric signals and peer signals, each flow's sender and receiver transmit out-of-band signals periodically (*e.g.,* per RTT), by sampling (*e.g.,* per BDP), or respond to anomaly triggers as discussed in §4.3.1. Compared with per-packet in-band signals, out-of-band signals are relatively coarse-grained, indicating more bytes sent per event. Thus, a smaller thread pool (16 in our setting) is sufficient for processing.

Strike Time-Space Trade-off: Improving both time/space efficiency is essential for DPA, which features a low-profile design with relatively wimpy cores and memory subsystems. Facing the fundamental time-space tradeoff, we employ distinct strategies for memory-intensive stateful operations and computation-intensive stateless operations. Details follow:

For stateful operations, we prioritize space efficiency by improving cache locality. Transport control algorithms often rely on stateful operations like membership queries, key-value queries, and counting. Stateful memory typically scales with the number of flows or the value range of flow information (*e.g.,* flow ID), but DPA memory resources are limited. To optimize stateful memory usage, we offer a computation library powered by compact data structures and algorithms. For membership queries, such as identifying the first packet of a flow, we utilize BloomFilter [79]. For key-value queries, such as recording flow statistics, we provide a HashMap. For counting tasks, we employ HyperLogLog [80] to estimate the cardinality and use Count-Min sketch [81] to estimate cumulative values.

For stateless operations, we prioritize time efficiency by lowering the instruction count. In addition to memory access costs, the instruction count directly translates into the execution time. For example, to improve time efficiency, we opt for fixed-point numbers (FXP-16) instead of floating-point numbers for arithmetic operations. Additionally, we employ bitwise operations for multiplication, square root calculations, and other arithmetic computations.

5 Spectrum of New Customizations

In this section, we demonstrate how to enable new functionality over SCR. To comprehensively cover a wide range of innovations, we categorize them into various system domains, as illustrated in Figure 8. For each domain, we explore candidate policies. Considering the rich customizations (Appendix Table 6), one policy never fits all, even for a single domain like congestion control. Thus, these policies showcase SCR's expressiveness (without claiming to be the best) and stimulate other innovations for diverse workloads and deployments.

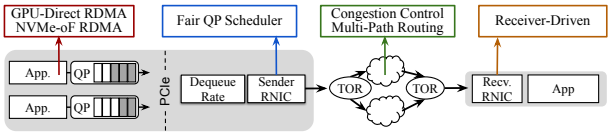


Figure 8: **New Customizations from Various Domains.**

5.1 Host Domain: Fair QP Scheduler

In this domain, we investigate the case of customizing the RNIC QP Scheduler to improve fairness and performance isolation across multiple tenants. Several prior works [17, 18, 24, 28, 73] investigate the importance of addressing multi-tenancy issues of QP schedulers.

Customized Policy: We specify one simple yet practical policy: Achieve fair bandwidth allocation for QPs with varying message sizes. SCR can support other policies, such as different performance metrics and different definitions of fairness. Here, we take a typical bandwidth fairness example. Before delving into the mechanisms, we first explain why the legacy RNIC is not fair for some scenarios. As described in §4.2, the QP scheduler dequeues WQEs from tenants' QPs, prompting RNIC to initiate DMA transfers to fetch messages across the PCIe. Consequently, the QP scheduler's decisions can significantly impact host-to-RNIC PCIe resources and RNIC hardware utilization. By default, RNIC adopts a greedy approach, prioritizing large message sizes to achieve high utilization, a phenomenon widely observed [18,24,56]. However, this greedy strategy may result in unfair resource allocation in multi-tenant environments with varying message sizes, potentially violating cloud providers' bandwidth commitments.

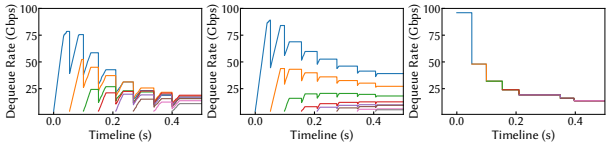
Control Laws: Firstly, we need to measure the real-time bandwidth for each flow, which is a metric to evaluate fairness. We utilize the TX events from SCR event colletor, which provide per-QP flow ID, event timestamps, and the amount of bytes sent out. For each flow, SCR event processors record the timestamp of the last observed TX event, calculate the time duration since then until the current event, and derive the real-time bandwidth utilization as the bytes sent out divided by the time duration. Note that such bandwidth calculation applies to events coalesced using the accumulation strategy. We utilize Exponentially Weighted Moving Average (EWMA) to smooth the real-time bandwidth. To enhance efficiency, BloomFilter performs the new flow identification, flow records are stored using HashMap, and floating-point calculations are approximated using fixed-point arithmetic.

Next, we present three algorithms to achieve fairness. Note that it is desirable to maintain high utilization while achieving fairness. SCR event processors support user-customized algorithms, here demonstrating three typical cases: 1) Default RNIC QP scheduler mechanism; 2) Static Allocation Algorithm: We statically control the dequeue rate for each flow as $\frac{Link_Capacity}{Num_Of_Flow}$. 3) Water Filling (Progressive-Filling) Algorithm: Water Filling is a classical strategy to achieve max-min fairness in bandwidth allocation problems [82, 83]. Specifically, in our dequeue rate control model, we initially set the dequeue rates at starting values (discussed later) and

Case Studies	Packet-Granular Signals/Events	Benefits and Performance Gains
Fair QP Scheduler	TX Events	Improving Fairness with Varying Message Sizes; Isolating Performance between Lat./BW. Tenants
Congestion Control (§A.6)	Fabric RTT Events; Host/Peer Endpoint Latency	Enable RTT-based CC in addition to ECN-based CC
Multipath Routing (§A.7)	Fabric-Domain RTT Probing	Enable Sender-Side Multi-Path Monitoring and Selection
Receiver-Drivern Control (§A.8)	Peer Credits; Peer Congestion Notifications	Handle Incast; Enable NIC-Initiated ECN-like Backpressure
ML and Storage (§A.9)	The above signals are applicable.	Enhancing Fairness with Zero-Intrusion for GPU-Direct RDMA and NVMe-oF RDMA Applications and Libraries

Table 4: **Summary of Case Studies on Utilized Packet-Granular Signals and Achieved Benefits.**

then progressively increase rates. If a tenant QP can saturate the dequeue rate (*i.e., BW_EWMA* $\approx$ *DequeueRate*) and the link capacity permits, we continue increasing it. However, if the tenant QP does not fully utilize the dequeue rate (*e.g.,* for non-bandwidth-hungry QPs), we stop further increases. Additionally, if a tenant's bandwidth degrades (*i.e., BW_EWMA* $\ll$ *DequeueRate*), we decrease the dequeue rate and reclaim the credits. Event processors based on the DMT model ensure the rapid completion of the filling process and simultaneous progress among multiple flows.



(a) Scale down to half; Start from 0 (b) Scale down to $\frac{n-1}{n}$; Start from 0 (c) Scale down to $\frac{n-1}{n}$; Start from $\frac{1}{n}$

Figure 9: **Various Water Filling Strategies for Handling Flows Dynamicity (8 Bandwidth-Hungry QPs).**

To handle the dynamics of flows coming and going, we adjust the bandwidth allocations as follows: when a new flow joins, we scale the allocations for each existing flow i down to $\frac{n-1}{n} \times DequeueRate_i$ and start the new flow at $\frac{1}{n} \times LineRate$ (where n is the number of flows); when a flow terminates, we reclaim its allocated *DequeueRate*, making it available for other flows. In Figure 9, we visualize three Water Filling strategies with time-series dequeue rates for 8 flows, initiated one by one at 50 ms intervals (More experiment setup in §A.4). Our strategy (Figure 9c) demonstrates efficiency in both fairness and convergence time. Water Filling offers a broad design space, including strategies for handling the flows dynamicity and several tunable parameters, such as step size. In Appendix §A.4, we present more traces with varying parameters. In this work, we focus on the expressiveness of SCR (without claiming algorithms to be the best) and defer the algorithm optimality to future research.

5.2 More Case Studies

Table 4 summarizes key enablers and performance gains for more case studies. Due to space constraints, we provide their descriptions and experiments in the appendix (§A.6, §A.7, §A.8, and §A.9). Given the rich literature on RDMA, we provide a feasibility analysis for other works in §A.13.

6 Evaluation

Implementation: We implement SCR over the DPA in over 7.8K lines of C/C++ code. We describe specific SDK dependencies in A.10. We describe our testbed in §3.2.

6.1 Micro-Benchmarking SCR

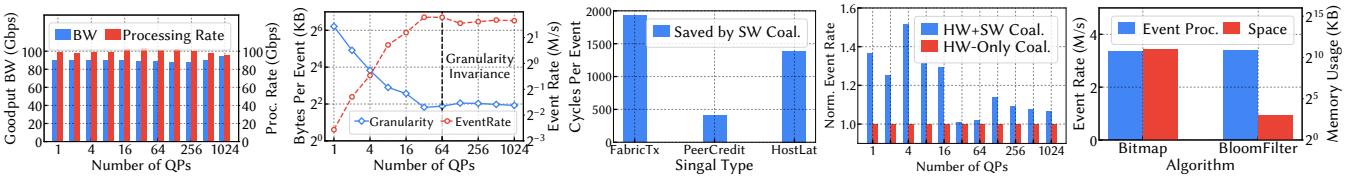
Setup: In Figure 10, we micro-benchmark the SCR framework using Perftest *ib_write_bw* [84] to generate traffic. We vary the number of QPs (flows) up to 1024 per node. The message size is set to 64KB with a queue depth of 4, allowing full 100Gbps line rate utilization. We launch up to 64 DPA threads. We primarily use the following metrics:

- *Event Rate:* Number of events handled (millions/second);
- *Coalescing Granularity:* Bytes sent per event;
- *Processing Rate:* Event Rate $\times$ Coalescing Granularity.

Processing Capacity: In Figure 10a, we demonstrate that the processing rate of the SCR can match the line rate traffic. In this experiment, SCR's processing task is to enforce fairness across multiple tenants. We utilize up to 1024 concurrent QPs and dump out the processing rate for the fabric TX signal, which is the most frequent signal among all signal types from all domains. The figure illustrates that SCR can handle these signals even during line-rate traffic. The processing rate exceeds the application goodput, as the bytes indicated by the TX event include both transport headers and payloads.

Scalability: In Figure 10b, we investigate the scalability using the same setup as the previous test. As we increase the number of QPs from 1 to 64, each flow exclusively utilizes one thread for event processing as designed in §4.3.3. With varying the number of flows and corresponding threads from 1 to 64, the line rate remains saturated and unchanged, reducing the processing load on individual threads. Thus, as the number of QPs increases to 64, we observe that the SCR can process events with the finest granularity and peak event rate. Beyond 64 QPs, we do not increase the thread count; instead, multiple flows are consolidated into a single thread. Excitingly, as analyzed by the *Granularity Invariance Principle* in §4.3.3, the granularity remains invariant even with up to 1024 QPs. This indicates that the SCR can handle more flows without additional threads, maintaining the same processing fidelity.

Software Coalescing Applicability: In Figure 10c, we show the applicability of software coalescing. Unlike hardware coalescing, which is limited to in-band signals from the fabric domain, software coalescing in SCR extends to signals across all domains. In this test, we apply software coalescing to fabric TX signals, peer credit signals, and host PCIe latency signals, showing the reduction in processing cycles per event. We use cycles as the metric—rather than execution times—because the DPA lacks a sub-μs timer (for reference, the DPA system clock runs at 505 MHz). Also, the figure highlights the flexibility of user-defined coalescing strategies. For TX events, we implement coalescing based on the cover-



(a) Proc. Matches LineRate (b) Granularity Invariance (c) SW Coal. Applicability (d) SW Coal. Efficiency (e) Space Efficiency

Figure 10: **Micro-Benchmarking Key Properties of the SCR Framework.**

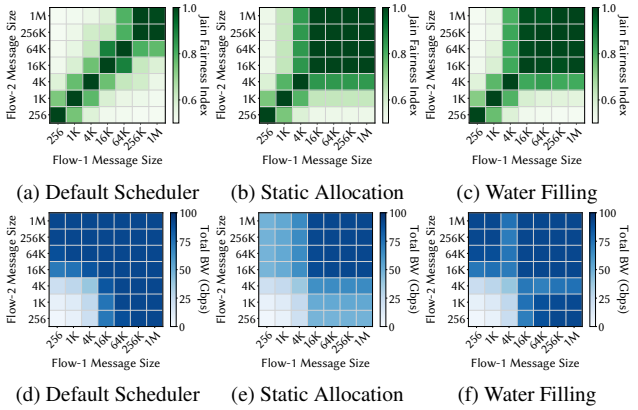
age status of flow rates (detailed in §4.3.2). For peer and host signals, we employ the *Keep-Latest-Event* strategy.

Software Coalescing Efficiency: In Figure 10d, we present the enhanced efficiency of software coalescing over typical hardware-only coalescing. As in previous experiments, we report the event rate for TX events with multi-tenant isolation processing task. To demonstrate enhanced efficiency, we normalized the combined hardware and software (HW+SW) coalescing against hardware-only (HW-Only) coalescing. The results show that software coalescing improves efficiency, particularly when the number of threads is limited, thereby increasing the load per thread. For instance, with only four threads, the HW+SW coalescing achieves a 51.3% higher event rate compared to the HW-Only coalescing.

Space Efficiency: In Figure 10e, we show the space efficiency for stateful operations, using the example of identifying the first signal of new flows—an essential operation for initiating processing. The Precise Bitmap Algorithm pre-allocates one bit for each potential flow ID. Upon signal arrival, it indexes the bit according to the flow ID to verify existence. With Flow IDs generated by BF3 hardware being 24 bits long, the Bitmap requires about 2 MB of memory (2^{24} bits). The Bloom Filter is an approximate algorithm for membership queries, with memory usage scaling only with the number of flows, not the value range. To achieve a false positive rate of 0.03 with three hash functions for up to 8192 flows, BloomFilter requires just 8 bits per flow (totaling 8KB). The Bloom Filter's space requirement is smaller than that of the Bitmap algorithm. Additionally, both methods can achieve similar event processing rates as they execute O(1) operations.

6.2 Achieving Fairness with the QP Scheduler

In Figure 11, we report the bandwidth as the utilization metric and Jain's Fairness Index [85] as the fairness metric. We present the evaluation results for the three aforementioned algorithms (§5.1). We set up two competing tenants using Perftest [84] WRITE verbs and vary the message size. The total link capacity is 100 Gbps. Figure 11a and 11d depict the performance of the default scheduler. It can only ensure fair allocation when both tenants use the same message size, as indicated by the diagonal line in Figure 11a. With the default scheduler, large messages can achieve high utilization, as shown in Figure 11d. Notably, for small messages (*e.g.,* 256 B), which are not bandwidth-hungry, they cannot saturate the link capacity. Figure 11b illustrates the fairness results when using the static allocation algorithm, which is the fairest



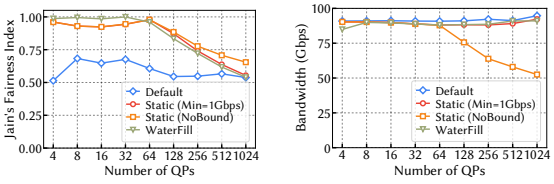
(a) Default Scheduler (b) Static Allocation (c) Water Filling

(d) Default Scheduler (e) Static Allocation (f) Water Filling

Figure 11: **Enhancing Fairness and Utilization in QP Scheduling. Deeper Color is Better for Both Metrics.**

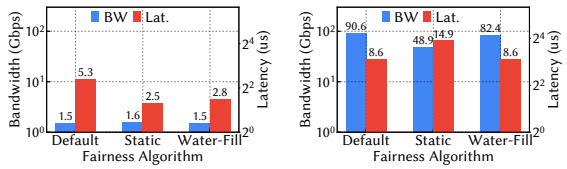
as it allocates exactly $\frac{1}{n}$ bandwidth to each tenant. However, static allocation can lead to bandwidth over-provisioning for smaller messages and starvation for larger messages. For instance, while bandwidth-hungry tenants with large message sizes (*e.g.,* 1 MB) can starve due to the 50 Gbps budget, small messages like 256B fail to fully utilize 50 Gbps. As a result, overall utilization suffers. The Water Filling algorithm strikes a balance between fairness and utilization by progressively filling the bandwidth demands for all tenants. As depicted in Figures 11c and 11f, the Water Filling algorithm can achieve comparable fairness to static allocation while enhancing utilization. Unlike static allocation, the Water Filling prevents over-provisioning for small messages and starvation for large messages. In Figure 11, the Water Filling algorithm improves the fairness index of up to 1.78x compared to the default scheduler (*e.g.,* 4KB-msg. QP *vs.* 1MB-msg. QP).

In Figure 12, we scale the number of QPs to evaluate fairness algorithms. Each server manages up to 1024 QPs, with each QP's message size uniformly sampled from the range [4KB, 1MB], resulting in varied message sizes across QPs. In this experiment, we use WRITE operations, while experiments for READ are in Appendix §A.5. The queue depth is 4 for all QPs. In Figure 12a, the Static and Water Filling allocations can improve fairness by up to 1.88x than the default scheduler (for #QPs=4, increasing from 0.51 to 0.96). Figure 12b shows that all three algorithms can almost saturate the line rate. Notably, when the number of flows exceeds 100, each flow is allocated less than 1 Gbps. Achieving accurate hardware rate limiting at the granularity of Mbps for RNICs is still challenging [86, 87]. In Figure 12b (Static-NoBound),



(a) Fairness Index (b) Bandwidth Utilization

Figure 12: **Fairness Across Multiple Tenants (WRITE).**



(a) Latency-Sensitive QP (b) Bandwidth-Hungry QP

Figure 13: **Performance Isolation (Lat QP *vs.* BW QP).**

setting the flow rate below 1 Gbps reduces total utilization and the inability to saturate the line rate, likely due to excessive arbitration within the rate limiter. Thus, to avoid low utilization, we set a minimum bandwidth bound of 1 Gbps per flow for both Static and Water Filling approaches. Setting the bandwidth roughly at 1 Gbps rather than a more precise Mbps level explains the decline in the fairness metric when the number of flows exceeds 100. Beyond 100 flows, the significance of achieving fairness becomes minimal since each flow is allocated only a small amount of bandwidth, which is fairly similar across flows.

6.3 Enhancing Tenants Isolation with the QP Scheduler

In Figure 13, we illustrate enhancing fairness and isolation can decrease latency for latency-sensitive tenants when competing with bandwidth-hungry tenants. We use Perftest WRITE [84] to set up two competing tenants: one bandwidth-hungry, utilizing large-message sizes (64 KB) and large batch sizes (128), and the other latency-sensitive, using small-message sizes (256B) and small batch sizes (4). Figure 13a displays the latency results for the latency-sensitive QP, while Figure 13b shows the bandwidth for the bandwidth-hungry QP. Under the default scheduler, the latency-sensitive tenant experiences significant latency spikes. With static allocation, where the bandwidth-hungry QP is limited to half the line rate, interference with the latency-sensitive tenant is reduced, but throughput suffers. Water Filling algorithms maintain high utilization while simultaneously reducing latency. In Figure 13, the Water Filling algorithm reduces latency by up to 52.8% compared to the default one while achieving high utilization. For the latency-sensitive tenant, all three algorithms deliver the same bandwidth. For the bandwidth-sensitive tenant, Water-Filling matches the default scheduler's latency. However, the Static algorithm increases latency due to restricting the bandwidth-hungry QP to only half the line rate.

Moreover, as depicted in Figure 13, we can draw a straightforward yet significant observation: SCR enables customization without compromising (even improving!) RDMA data plane performance, particularly critical latency metrics. This observation holds true across all case studies.

6.4 Discussion

More Cases and Guidelines: As referenced in Table 4, more cases are detailed in the appendix. We report some developer guidelines in A.11 and suggestions for vendors in A.12.

Relationship to DOCA PCC: We reference PCC [68] for APIs of setting rates for QPs and obtaining fabric-domain signals. In SCR, we propose the Dequeue Rate Control Model (§4.2 and §4.3.1), Software Coalescing and Multiplexing to address DPA RTOS limitations (§4.3.2), and strategies to efficiently utilize DPA (§4.3.3). We anticipate that these contributions will also benefit PCC.

Limitations of SCR: In §4.3.1, we describe the lack of RX signals. In §4.3.2, we discuss the inability to create threads during runtime and the IPC limitation. In Figure 12, achieving precise rate limiting at the Mbps granularity remains a challenge. In §A.7, we describe the limitation on how to specify routing paths. In Appendix A.11 and A.12, we report some latency issues such as setting rates. We expect future software and hardware advancements to address these issues.

Generality: Besides the BF3 DPA, we explore generalizing the SCR design to other platforms such as hardware devices for Google Falcon [4] and Broadcom's Inband Flow Analyzer (IFA) [88, 89]. Google Falcon, according to the spec [4], features a Rate Update Engine (RUE), which occupies a similar architectural position as the BF3 DPA. Thus, our dequeue rate control and coalescing/multiplexing strategies could be adapted for use with the RUE. Similarly, according to Broadcom IFA's draft, the Initiating Function Node can potentially fulfill a role akin to the BF3 DPA in SCR.

Resource Costs: We discuss computational and network resources costs of SCR. In practice with BF3, we use minimal computational resources from the host CPU or DPU ARM, typically for lightweight tasks such as launching DPA threads or responding to host probes; a single CPU core suffices in our experiments. Moreover, DPA, based on RISC-V, is designed for efficiency in power usage, chip area, and monetary cost [60, 90]. Besides BF3 DPU, even the low-power version of the BF3 (SuperNIC [91]) incorporates DPA. In terms of network resources, both out-of-band probing and in-band telemetry consume bandwidth. It is a common challenge [9, 92, 93].

7 Conclusion

In this work, we introduce Software-Controlled RDMA (SCR) to realize the vision of white-boxing RDMA. SCR applies the dequeue rate control model and facilitates an event-driven rate computation framework. We enable a spectrum of new functionalities not present in legacy transport. We envision SCR to help land past and future RDMA transport innovations.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Masoud Moshref Javadi. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was supported in part by NSF grants CNS-2212193, CNS-2213387, CNS-2106199, CNS-2212192, CNS-2339755, and Cisco Research Grant.

References

[1] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[2] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Google Falcon. https://github.com/opencomputeproject/OCP-NET-Falcon, 2024.

[5] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

[6] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[8] Ultra Ethernet Consortium. https://ultraethernet.org/, 2024.

[9] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58. Association for Computing Machinery, New York, NY, USA, 2019.

[10] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Yiran Zhang, Qingkai Meng, Chaolei Hu, and Fengyuan Ren. Revisiting congestion control for lossless ethernet. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 131–148, Santa Clara, CA, April 2024. USENIX Association.

[12] Yiran Zhang, Yifan Liu, Qingkai Meng, and Fengyuan Ren. Congestion detection in lossless networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 370–383, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX symposium on networked systems design and implementation (NSDI 22)*, pages 51–70, 2022.

[14] Prateesh Goyal, Preey Shah, Naveen Kr Sharma, Mohammad Alizadeh, and Thomas E Anderson. Backpressure flow control. In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 1–3, 2019.

[15] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Vladimir Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 761–777, 2022.

[17] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. Harmonic: Hardware-assisted RDMA performance isolation for public clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, April 2024. USENIX Association.

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

[19] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 31–48, 2023.

[20] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path transport for RDMA in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 357–371, 2018.

[21] Sugi Lee, Yusung Kim, Honguk Woo, and Ikjun Yeom. Efficient user-level multi-path utilization in rdma networks. *IEEE Access*, 9:127619–127629, 2021.

[22] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 313–326, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific network transport for distributed DNN training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, April 2024. USENIX Association.

[24] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, et al. Flor: An open high performance RDMA framework over heterogeneous RNICs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 931–948, 2023.

[25] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest*

Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 708–721, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. SRNIC: A scalable architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, 2023.

[27] Zhiqiang He, Yuxin Chen, and Bei Hua. Roud: Scalable rdma over ud in lossy data center networks. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 36–46. IEEE, 2023.

[28] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. Rogue: Rdma over generic unconverged ethernet. In *Proceedings of the ACM symposium on cloud computing*, pages 225–236, 2018.

[29] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 1–12, New York, NY, USA, 2007. Association for Computing Machinery.

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

[31] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery.

[32] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, 2020.

[33] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[34] Everything You Should Know About White Box Switch. https://cloudswit.ch/blogs/what-is-white-box-switch/#what-is-white-box-switch, 2021.

[35] NVIDIA BLUEFIELD-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2024.

[36] Broadcom Ethernet Network Adapters. https://www.broadcom.com/products/ethernet-connectivity/network-adapters, 2024.

[37] AMD Pensando. https://www.amd.com/en/accelerators/pensandoe, 2023.

[38] AWS Nitro System. https://aws.amazon.com/ec2/nitro/, 2023.

[39] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA, July 2023. USENIX Association.

[40] Next-Generation Networking for the Next Wave of AI White Paper. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/next-generation-netw, 2024.

[41] Chris V Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. On designing improved controllers for aqm routers supporting tcp flows. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1726–1734. IEEE, 2001.

[42] Steven H Low, Fernando Paganini, Jiantao Wang, Sachin Adlakha, and John C Doyle. Dynamics of tcp/red and a scalable control. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 239–248. IEEE, 2002.

[43] Laurent Massoulie. Stability of distributed congestion control with heterogeneous feedback delays. *IEEE Transactions on Automatic Control*, 47(6):895–902, 2002.

[44] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. Plb: congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 207–218, New York, NY, USA, 2022. Association for Computing Machinery.

[45] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Abdul Kabbani, et al. Datacenter ethernet and rdma: Issues at hyperscale. *arXiv preprint arXiv:2302.03337*, 2023.

[46] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[47] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery.

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

[49] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[50] NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl, 2024.

[51] Jiao Zhang, Jiaming Shi, Xiaolong Zhong, Zirui Wan, Yu Tian, Tian Pan, and Tao Huang. Receiver-driven rdma congestion control by differentiating congestion types in datacenter networks. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2021.

[52] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[53] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[54] NVIDIA Spectrum-X Ethernet switch. https://www.nvidia.com/en-us/networking/spectrumx/, 2024.

[55] Broadcom Tomahawk Ethernet Switches. https://www.broadcom.com/products/ethernet-connectivity/switchin, 2024.

[56] Yunkun Liao, Jingya Wu, Wenyan Lu, Xiaowei Li, and Guihai Yan. Optimize the tx architecture of rdma nic for performance isolation in the cloud environment. In *Proceedings of the Great Lakes Symposium on VLSI 2023*, pages 29–35, 2023.

[57] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. Star: Breaking the scalability limit for rdma. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

[58] Xizheng Wang, Shuai Wang, and Dan Li. srdma: A general and low-overhead scheduler for rdma. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, pages 21–27, 2023.

[59] Soft-RoCE. https://enterprise-support.nvidia.com/s/article/howto-configure-soft-roce, 2024.

[60] NVIDIA DPA Subsystem Programming Guide . https://docs.nvidia.com/doca/sdk/doca+dpa/index.html, 2024.

[61] Idan Burstein. Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20. IEEE, 2021.

[62] Intel Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html, 2023.

[63] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Ming Liu, and Zeke Wang. Demystifying datapath accelerator enhanced off-path smartnic. *arXiv preprint arXiv:2402.03041*, 2024.

[64] Benjamin Michalowicz, Kaushik Kandadi Suresh, Hari Subramoni, Dhabaleswar Panda, and Steve Poole. Dpu-bench: A micro-benchmark suite to measure offload efficiency of smartnics. In *Practice and Experience in Advanced Research Computing*, pages 94–101, 2023.

[65] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333. Association for Computing Machinery, New York, NY, USA, 2019.

[66] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 740–755, 2021.

[67] Matrix Multiplication. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, 2024.

[68] NVIDIA DOCA Programmable Congestion Control (PCC) API. https://docs.nvidia.com/doca/sdk/doca+pcc/index.html, 2024.

[69] Larry W McVoy, Carl Staelin, et al. Lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[70] Zilong Wang, Xinchen Wan, Chaoliang Zeng, and Kai Chen. Accurate and scalable rate limiter for rdma nics. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, pages 15–20, 2023.

[71] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 404–417, New York, NY, USA, 2017. Association for Computing Machinery.

[72] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 475–488, 2014.

[73] Yiwen Zhang, Juncheng Gu, Youngmoon Lee, Mosharaf Chowdhury, and Kang G Shin. Performance isolation anomalies in rdma. In *Proceedings of the Workshop on Kernel-Bypass Networks*, pages 43–48, 2017.

[74] Intel Performance Counter Monitor (Intel PCM). https://github.com/intel/pcm, 2024.

[75] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefler. Noise in the clouds: Influence of network performance variability on application scalability. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(3):1–27, 2022.

[76] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 53–64, 2010.

[77] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 337–351, New York, NY, USA, 2011. Association for Computing Machinery.

[78] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. Deterministic process groups in dOS. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[79] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[80] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science*, 2007.

[81] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[82] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.

[83] Pooria Namyar, Behnaz Arzani, Srikanth Kandula, Santiago Segarra, Daniel Crankshaw, Umesh Krishnaswamy, Ramesh Govindan, and Himanshu Raj. Solving Max-Min fair resource allocations quickly on large graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, April 2024. USENIX Association.

[84] Linux-rdma perftest . https://github.com/linux-rdma/perftest, 2024.

[85] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 21:1, 1984.

[86] Zilong Wang, Xinchen Wan, Luyang Li, Yijun Sun, Peng Xie, Xin Wei, Qingsong Ning, Junxue Zhang, and Kai Chen. Fast, scalable, and accurate rate limiter for rdma nics. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 568–580, New York, NY, USA, 2024. Association for Computing Machinery.

[87] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM special interest group on data communication*, pages 367–379, 2019.

[88] Network Performance Anomaly detection with In-band Flow Analyzer (IFA). https://www.broadcom.com/blog/network-performance-anomaly-detection-with-in-band-flow-analyzer, 2024.

[89] Jai Kumar, Surendra Anubolu, John Lemon, Rajeev Manur, Hugh Holbrook, Anoop Ghanwani, Dezhong Cai, Heidi Ou, Yizhou Li, and Xiaojun Wang. Inband Flow Analyzer. Internet-Draft draft-kumar-ippm-ifa-08, Internet Engineering Task Force, April 2024. Work in Progress.

[90] Mikhail Khalilov, Salvatore Di Girolamo, Marcin Chrapek, Rami Nudelman, Gil Bloch, and Torsten Hoefler. Network-offloaded bandwidth-optimal broadcast and allgather for distributed ai, 2024.

[91] What Is a SuperNIC? https://blogs.nvidia.com/blog/what-is-a-supernic/, 2024.

[92] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[93] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, Dai Zhang, Binzhang Fu, Frank Kelly, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. Predictable vfabric on informative data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 615–632, New York, NY, USA, 2022. Association for Computing Machinery.

[94] NVIDIA ConnectX NICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/, 2024.

[95] Intel Ethernet Network Adapters. https://www.intel.com/content/www/us/en/support/articles/000031905/ethernet-products/700-series-controllers-up-to-40gbe.html, 2024.

[96] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt: Sub-RTT congestion control for Ultra-Low latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 219–236, 2023.

[97] RDMA Core Userspace Libraries and Daemons. https://www.mankier.com/3/mlx5dv_modify_qp_udp_sport, 2024.

[98] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 275–287, New York, NY, USA, 2023. Association for Computing Machinery.

[99] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 753–766, New York, NY, USA, 2022. Association for Computing Machinery.

[100] Building Meta's GenAI Infrastructure. https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/, 2024.

[101] https://spdk.io/. https://spdk.io/, 2024.

[102] RDMA Core Userspace Libraries and Daemons. https://developer.nvidia.com/nccl, 2024.

[103] NVIDIA DOCA FlexIO SDK Programming Guide. https://docs.nvidia.com/doca/archive/doca-v1.5.0/flexio-sdk-programming-guide/index.html, 2024.

[104] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 30–43, New York, NY, USA, 2018. Association for Computing Machinery.

[105] Srinivas Narayana. Making decisions at data plane speeds. *SIGMETRICS Perform. Eval. Rev.*, 51(2):88–90, oct 2023.

[106] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: a building block for internet congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 158–176, New York, NY, USA, 2022. Association for Computing Machinery.

[107] Mellanox Neo-Host. https://support.mellanox.com/s/login/?ec=302&startURL=%2Fs%2Fproductdetails%2Fa2v50000000N2OlAAK%2Fmellanox-neohost, 2024.

[108] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

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

[110] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. High-throughput and flexible host networking via control and data path physical separation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[111] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, April 2024. USENIX Association.

[112] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. Congestion control in machine learning clusters. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 235–242, 2022.

[113] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. Network load balancing with in-network reordering support for rdma. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 816–831, New York, NY, USA, 2023. Association for Computing Machinery.

[114] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 15–29, Boston, MA, April 2023. USENIX Association.

[115] Yuzhen Su, Jiao Zhang, Zirui Wan, Pingping Lin, Yunpeng Zhang, Tian Pan, and Tao Huang. Hermes: An efficient building block for rdma incast in datacenters. In *2023 9th International Conference on Computer and Communications (ICCC)*, pages 2306–2311. IEEE, 2023.

[116] Jung-Hwan Cha, Shinhyeok Kang, Yewon Kang, Hansaem Seo, Jungeun Lee, Jongsung Kim, and Minsung Jang. Corn: Cloud-optimized rdma networking. In *2023 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 52–59, 2023.

[117] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-SpeedNICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, 2020.

[118] Wenxue Cheng, Kun Qian, Wanchun Jiang, Tong Zhang, and Fengyuan Ren. Re-architecting congestion management in lossless ethernet. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 19–36, 2020.

[119] Haonan Qiu, Xiaoliang Wang, Tianchen Jin, Zhuzhong Qian, Baoliu Ye, Bin Tang, Wenzhong Li, and Sanglu Lu. Toward effective and fair rdma resource sharing. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, pages 8–14, 2018.

[120] Dian Shen, Junzhou Luo, Fang Dong, Xiaolin Guo, Kai Wang, and John CS Lui. Distributed and optimal rdma resource scheduling in shared data center networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 606–615. IEEE, 2020.

[121] Yanfang Le, Mojtaba Malekpourshahraki, Brent Stephens, Aditya Akella, and Michael M Swift. On the impact of cluster configuration on roce application design. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking*, pages 64–70, 2019.

[122] Huichen Dai, Binzhang Fu, and Kun Tan. PFC-Free Low Delay Control Protocol. Internet-draft, Internet Engineering Task Force, July 2020.

[123] Benjamin Fuhrer, Yuval Shpigelman, Chen Tessler, Shie Mannor, Gal Chechik, Eitan Zahavi, and Gal Dalal. Implementing reinforcement learning datacenter congestion control in nvidia nics. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 331–343. IEEE, 2023.

A Appendix

A.1 Limitations of CPU/DPU-ARM in Meeting Control System Requirements

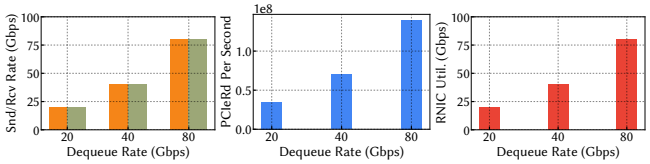
R1 Multi-threading Performance: For CPU and DPU ARM, while the host CPU has many cores, these cores should primarily be allocated to tenants for revenue-generating workloads. In contrast, the ARM CPU possesses only 16 cores without hyper-threading, partially satisfying the R1 requirement.

R2 Interaction with RNIC: For CPU and DPU ARM, several issues arise. Firstly, they currently lack APIs to acquire packet-granular signals, particularly in-band per-packet signals like ACK/CNP. Also, as far as we know, these cores lack APIs to enforce control actions, such as changing the per-flow rate during runtime. While the lack of APIs can be addressed in the short term, another long-term concern remains. As Figure 1a shows, DPA is co-packaged with the RNIC, with Network-on-Chip direct access to RNIC TX/RX pipes, resulting in lower access latency. However, host CPUs or ARM cores must cross the host PCIe or on-SoC PCIe to interact with RNICs. Furthermore, the host CPU and ARM cores access RNIC signals with the Performance Monitoring Unit (PMU) counters, which work in a sampling manner instead of timely monitoring.

R3 Raw Ethernet Capability: CPU and DPU ARM satisfy the requirements of the general network, including the in-kernel TCP/IP stack and RDMA transport. However, DPA is more tightly integrated with RNIC than CPU and ARM cores.

R4 Memory Access Latency: Note that it is trivial for the host CPU to monitor host-domain resource usage and record it with host memory, but this is not the case for the ARM CPU. Although the ARM CPU can access host memory using RDMA (μs scale), this incurs longer latency than the load/store accessing capabilities of the DPA (sub-μs scale).

A.2 Evaluation of Dequeue Rate's Impact Across Multiple Domains



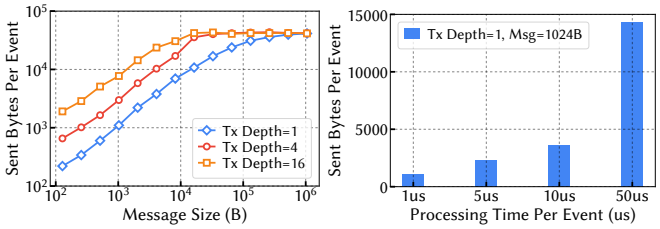
(a) Send/Receive BW (b) PCIe Utilization (c) RNIC Processing

Figure 14: **Dequeue Rate's Impact on RDMA Transport.**

In Figure 14, we explore the impact of dequeue rate on various domains: Fabric, Host, and Peer. Figure 14a illustrates how varying dequeue rates can regulate traffic sent into the fabric and received by peers. In Figure 14b, we monitor PCIe utilization using Intel PCM [74] tools across different dequeue rates. To understand how the dequeue rate influences RNIC

hardware utilization, we track DMA jobs initiated by the RNIC to fetch messages from host memory. Specifically, we use Intel PCM to monitor the IIO (Integrated Input/Output) controller for In-Bound read activity, representing the number of bytes requested by RNIC for reading from main memory via DMA. Figure 14c demonstrates the effect of the dequeue rate on RNIC DMA job execution.

A.3 Micro Benchmark of Coalescing TX Events

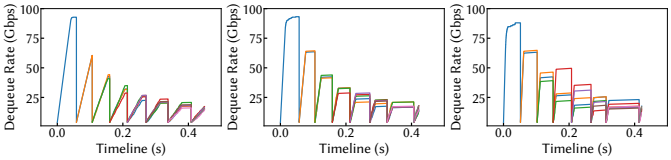


(a) Batching Triggers Coalescing (b) Slow-Proc. Triggers Coalescing

Figure 15: **Coalescing Experiments (TX Events Example).**

In Figure 15, we conduct a micro-benchmark on the coalescing of multiple TX events by accumulating the *sent_bytes* indicated by each TX event into a single sum value. Figure 15a demonstrates that larger message sizes per WQE and deeper SQ depths can trigger the coalescing of multiple TX events. It's worth noting the presence of a ceiling for *sent_bytes_per_event* due to the PCIe-related configuration on maximum batching size. Additionally, Figure 15b illustrates how slower event processing can also induce TX event coalescing. Overall, coalescing proves to be a promising mechanism for absorbing the high burstiness of packet-granular events.

A.4 Traces of Water Filling Procedure



(a) Step=0.02 GBps (b) Step=0.05 GBps (c) Step=0.1 GBps

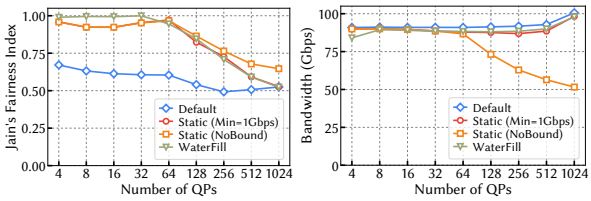
Figure 16: **Traces of Water Filling Procedure (#QPs=8). Scale down to 0; Start from 0.**

In Figure 16, we present time-series dequeue rates for eight flows during the Water Filling procedure. The minimum message QP utilizes 4KB, while subsequent QPs double this size successively, resulting in different message sizes per QP. All QPs have a queue depth of 4 and can saturate a 100 Gbps line rate individually. We initiate the dequeue rate at 0.5 Gbps, with step sizes varying from 0.03 Gbps, 0.05 Gbps to 0.1 Gbps. Flows are started one by one at 50 milliseconds intervals. To clearly demonstrate the impact of step size, we employ a strategy to handle flow dynamics: when a new flow joins, we simply reset all allocations and restart the Water Filling procedure. As indicated by the traces, smaller step sizes aid in converging to fairer rate allocations but require more time to achieve convergence. Table 5 shows the total time required for all flows to converge to a stable state when eight QPs begin filling concurrently. Note that to generate the trace for Figure 16, we need to output logs, which introduces some delay in the time series. For the measurements in Table 5, we disabled log printing to eliminate such delays.

Filling Step Size	Convergence Time
0.02 GBps	294 μs
0.05 GBps	122 μs
0.10 GBps	49 μs

Table 5: **Time Cost for All Flows to Converge to Stable Rates.**

A.5 Support RDMA READ Operation



(a) Fairness Index (b) Bandwidth Utilization

Figure 17: **Achieving Fairness Across Multiple Tenants (RDMA READ).**

In Figure 17, we demonstrate that SCR is applicable to various RDMA operations, including READ and WRITE. The experimental setup is identical to that shown in Figure 12, except that this experiment uses RDMA READ operations instead of the RDMA WRITE operations used previously. The results are similar: Static and Water-Filling allocations enhance fairness while maintaining high utilization and scalability, as analyzed in §6.2. SCR supports multiple RDMA operations, differing primarily in whether the requester's or responder's BF3 DPA takes effect. For WRITE operations, the Fair QP scheduler takes effect at the requester's BF3 DPA, as the requester sends out data and collects TX signals. Conversely, for READ operations, the Fair QP scheduler takes effect at the responder's BF3 DPA, since the responder sends out data and collects TX signals.

A.6 Fabric Domain: Congestion Control

In the fabric domain, we investigate congestion control (CC), a classical and everlasting problem. Most RNIC vendors [36, 94,95] baked DCQCN into the ASIC. With the introduction of

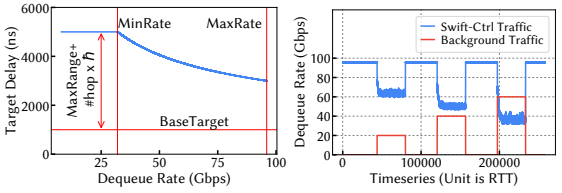
SCR, there is a promising opportunity for commercial RNICs to reap the benefits of other innovative CC [10, 11, 13, 47, 96].

Customized Policy: With ECN-based algorithms already demonstrated by DCQCN, we now explore implementing RTT-based CC using SCR. Delay-based CC is desired for several large-scale RDMA deployments [5, 46].

Swift Control Laws: We investigate Swift, a typical delay-based CC algorithm proposed by Google [10]. Swift's core logic is as follows: Additively increase the congestion window when RTT is less than the target delay; multiplicatively decrease the congestion window when RTT exceeds the target delay. With SCR, we examine RTT measurement and target delay computation, two building blocks for Swift's control.

Measuring RTT Delays: Swift control loops decompose the end-to-end delay into three components: Fabric Delay, Local NIC TX/RX Delay, and Remote NIC RX/TX Delay. Swift measures fabric delay by utilizing ACK packets to convey in-band timestamp information. As described in §4.3.1, SCR supports out-of-band per-flow RTT measurements, offering greater flexibility compared to in-band RTT measurements. This method allows for control over the frequency of sending RTT requests and eliminates the need to customize the ACK packet format. Pingmesh [92] shows such proactive RTT measurement is feasible in large-scale environments. For local NIC TX/RX delay indicating local host congestion, Swift originally utilizes timestamps from the local host and NIC to calculate per-packet delays, necessitating precise time synchronization between the NIC and host. Alternatively, SCR utilizes the host-domain signal of PCIe latency to reflect host congestion. For the remote NIC TX/RX delay, SCR utilizes signals from the peer domain, such as peer PCIe latency, to represent this delay.

Computing Target Delay: Swift encodes most of the complexity into the computation of the target delay, enabling it to scale to multiple competing flows and multiple-hop network typologies. The target delay calculations require floating-point and complex arithmetic operations like square root. To enhance efficiency, SCR employs fixed-point and bitwise operations. Given that Swift is a window-based algorithm and the current BF3 RNIC employs a rate-based mechanism, we utilize the Bandwidth-Delay Product (BDP) approach to translate the per-flow rate into the per-flow window.



(a) Target Delay Curve (b) Swift-Control Traffic

Figure 18: **Statistics Dump for Swift Congestion Control.**

Experiments of RTT-Based Congestion Control: As shown in Figure 18, we dump out the runtime statistics to verify the functionality of Swift. In Figure 18a, we present a trace demonstrating how the dequeue rate affects the target delay. According to our testbed, we configured Swift's parameters as follows: base target as 1 μs, #hop as 1, per-hop scaling factor as 2 μs, max scaling range as 2 μs, Min Rate as 32 Gbps, and Max Rate as 96 Gbps. We observe that our implementation accurately follows Swift's control law and reproduces the curve behavior as in the original design [10]. In Figure 18b, we generate background traffic varying from 20 Gbps, 40 Gbps to 60 Gbps and monitor the Swift-Controlled traffic. We can observe that SCR-enabled Swift can converge to the appropriate bandwidth to avoid fabric congestion.

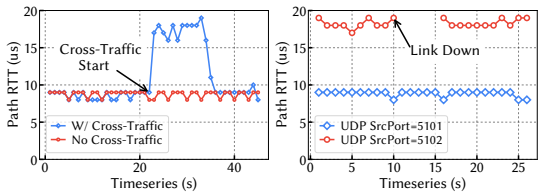
A.7 Fabric Domain: Multi-Path Routing Selection

We investigate the second case of the fabric domain, Multi-Path Routing, which is advocated by several industrial-level RDMA customizations such as Google PLB [44] in Falcon, AWS SRD [5], and UEC Transport [8].

Customized Policy: We monitor multi-path status and repath according to congestion status and link failure at the sender side. We revisit one prior policy, MP-RDMA [20], which repurposes the *Src Port* field of the RoCEv2 transport to indicate the *Virtual Path ID*, enabling switch ECMP to execute multipathing accordingly. MP-RDMA relies on the in-band ACK clocking to detect congestion status for each *Virtual Path*. The end-host logic of MP-RDMA is prototyped via FPGAs, not commercial RNICs. Next, we examine the feasibility of multi-path routing using SCR.

Control Laws: SCR offers two key functionalities: Multipath Monitoring and Path Selection. To monitor the status of multiple paths, we employ out-of-band telemetry to probe status proactively. In SCR, the local collector pings the peer, which responds with a pong, allowing the local collector to obtain the RTT. By varying the header field of the request, we can enforce RTT requests through different paths in the fabric. For header field assignments, we adopt a random generation approach, similar to the practice used in MP-RDMA [20] and PLB [44]. After detecting the RTTs of multiple paths, we select the path with the lowest latency. For path selection enforcement, we use the source port field of RoCEv2 to indicate the path, as MP-RDMA advocates. Currently, it is feasible to specify the source port field of RoCEv2 for each QP using APIs like *mlx5dv_modify_qp_udp_sport()* [97]. However, this cannot be done at the granularity of individual packets. Thus, we rely on the BF3 hardware flow engines within the NIC-embedded switch to rewrite the packet header [60].

Experiments of Monitoring Multi-Path Status: As Figure 19 shows, we demonstrate two scenarios for monitoring multi-path RTT. In Figure 19a, we present the measurements that reveal the congestion status of network paths effectively. We introduce 100 Gbps background cross-traffic between the sender and receiver, observing an increase in RTT when cross-traffic begins and a decrease when it ceases. In Figure 19b, we



(a) Path RTT with Cross-Traffic (b) Multi-Path RTT

Figure 19: **Latency Monitoring Across Multiple Paths.**

present another scenario monitoring multiple path statuses by varying the source port to control routing paths. By manually adding a 10 μs latency for traffic with UDP SrcPort 5102, we compare it with traffic on UDP SrcPort 5101, revealing accurate latency representation for different paths. We also simulate a link failure scenario for the path of UDP SrcPort 5102. We configured a timeout value (100 μs) to detect the link failure, causing the loss of RTT probing packets. Figure 19b illustrates RTT probing's ability to detect link failures and recovery. This information aids in SCR selecting the optimal path for RDMA traffic, even at the per-packet level, via RoCEv2 header manipulation.

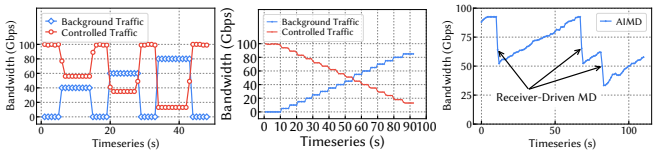
A.8 Peer Domain: Receiver-Driven Control Loop

In this domain, we investigate the receiver-driven control loop, which is not present for legacy RDMA transport. RDMA operations, especially one-sided READ/WRITE, are designed to be sender-driven, allowing transfers to bypass the involvement of the peer entirely. However, sender-side controls may struggle to handle traffic patterns that cause congestion at the receiver side, such as many-to-one incast. The receiver-driven paradigm can help handle the incast [15, 16, 52, 53].

Customized Policy: We refer to one typical receiver-driven protocol, Homa [15]. In Homa's control loop, the sender is only allowed to send bytes after receiving a *grant* from the receiver, enabling efficient handling of incast scenarios by allowing a few senders to transmit simultaneously. Next, we leverage SCR to transfer such receiver-initiated signals to benefit RDMA one-sided operations WRITE/READ.

Control Laws: SCR provides the peer-domain signals as described in §4.3.1. Peer-domain signals, transmitted out-of-band as packet payloads, offer flexibility for carrying information, distinct from modifying in-band packet headers. The in-band manner requires addressing size limitations and ensuring packet routability across the fabric and middlebox. To support Homa-like grant mechanisms, we utilize peer domain signals to convey the grant information. Originally, Homa utilized a window-based control mechanism where credits indicated the allowable byte counts. Given that the current BF3 supports only rate-based control, we have made two modifications: First, we use credits to represent the available bandwidth (in Gbps) for the sender. Second, the senders operate *optimistically*, continuing to transmit at the current rate unless new credits are received to adjust the traffic.

In addition to grant mechanisms, we also support a second receiver-driven mechanism where the receiver issues ECN-like backpressure notifications to the sender. Commodity RNICs currently lack mechanisms to mark ECN bits at the receiver NIC. This functionality is beneficial for related work [17, 98] because receivers can mark the ECN bits to backpressure the senders. While SCR doesn't directly mark ECN bits on in-band packets, we utilize out-of-band peer-domain signal packets to convey ECN-like information to backpressure senders.



(a) Recv.-Granted Credit (b) 16-to-1 Incast (c) Recv.-Init. ECN

Figure 20: **Receiver-Driven Control Mechanisms.**

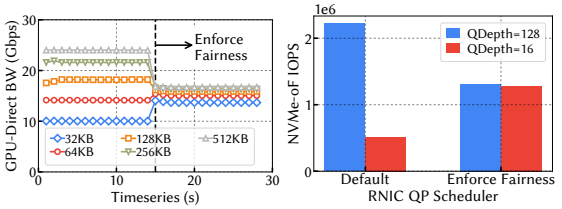
Experiments of Receiver-Driven Control: In Figure 20a, we illustrate the mechanism of receiver-granting credits to handle the 2-to-1 incast scenario. We use Perftest to configure two clients (one BF2 and one BF3) sending data to the same destination server (one BF3), with BF2-to-BF3 traffic acting as the background cross-traffic. The BF3-to-BF3 connection demonstrates the receiver-granting credits mechanism. Specifically, the receiver-side SCR monitors the real-time bandwidth utilization of the given port and calculates the credit as the remaining bandwidth headroom. These credits are then transferred to the sender as peer-domain signals, allowing the sender-side SCR to set the dequeue rate accordingly. From Figure 20a, we observe that the sender appropriately utilizes the remaining bandwidth. In Figure 20b, we dynamically scale the number of background senders up to 16, adding one sender every 5 seconds, with each sender generating traffic at 5 Gbps, achieving an incast ratio of up to 16-to-1. The figure shows that the BF3-to-BF3 connection, equipped with the receiver-granting credits mechanism, effectively captures the remaining bandwidth across all settings.

In Figure 20c, we demonstrate a scenario of receiver-initiated ECN-like backpressure flow control. We utilize Perftest [84] to establish a QP connection between the sender and receiver, generating WRITE traffic at the line rate (100 Gbps). The receiver can trigger receiver-driven backpressure signals while congestion is happening at its side. Upon receiving peer-domain signals, the sender multiplicatively decreases (MD) the dequeue rate and then performs additive increases (AI) without further backpressure signals. To ensure visibility, we use a small AI factor to extend the AI process. As shown in this experiment, while WRITE is inherently sender-driven, it currently can benefit from the receiver-driven control loop.

A.9 Application Domain: ML and Storage

We highlight the feasibility of harnessing SCR framework to NVMe-oF RDMA and GPU-Direct RDMA because storage [1, 99] and machine learning workloads [46, 100] take the significant portion of current data center traffic. They can seamlessly leverage our framework with minimal overhead, as SCR operates at the hardware transport level without changing the host software stack. Specifically, it leaves essential data path functionalities like peer-to-peer DMA transfer unchanged (*i.e.,* GPU-to-NIC and SSD-to-NIC). Also, SCR requires no alterations to the application code, host drivers like CUDA, and libraries like SPDK [101] and NCCL [102]. Moreover, SCR ensures optimal raw RDMA performance.

Customized Policy: We ensure application-driven bandwidth allocation for multi-tenant storage or ML workloads without changing any application code. Specifically, multiple NVMe-oF initiators on the same host share NVMe-oF RDMA bandwidth according to the policy, regardless of their individual IO depth settings. Similarly, multiple tenants on the same GPU share GPU-Direct bandwidth according to the policy, even though the size of messages sent from HBM varies among them. The control law is the same as in §5.1, but the policy in §5.1 manages traffic from main memory to PCIe endpoint (EP); this policy controls EP-to-EP traffic.



(a) GPU-Direct RDMA (b) NVMe-oF RDMA

Figure 21: **Ensuring Fairness in GPU-Direct and NVMe-oF RDMA Traffic.**

Experiments of Achieving Fairness for ML/Storage Workloads: In Figure 21a, we demonstrate how SCR effectively manages GPU-to-GPU RDMA communication. Using Perftest, we set up five concurrent RDMA connections between two GPUs, utilizing GPUDirect RDMA to directly transfer messages between HBMs without involving the host DDR as an intermediary. Here, we demonstrate the impact of an application-driven QP scheduler. Initially, with five connections having different message sizes and using the default scheduler, bandwidth allocation favors large message tenants. When SCR is used to ensure even sharing, all tenants benefit from an equitable share of the bandwidth. Importantly, no modifications were made to any of the host software stack.

In Figure 21b, we showcase the efficacy of SCR for storage workloads utilizing NVMe-oF RDMA. At the NVMe-oF target side, we employ SPDK to configure one *nvmf_tgt* with NULL devices, rather than read SSDs, to maximize network traffic without encountering bottlenecks on the storage medium. On the initiator side, we set up two tenants. Both tenants issue 4KB write requests, while varying the IO queue depth for different tenants. It is evident that with the default QP scheduler, smaller IO depths struggle to obtain the bandwidth. However, when we employ SCR to enforce an application-specific policy of even sharing, these tenants can equitably share the bandwidth without compromising utilization. Importantly, no modifications were required for the SPDK NVMe-oF implementation and the applications.

A.10 SDK Dependency and Implementation

We implement SCR over the DPA in over 7.8K lines of C/C++ code, covering both DPA and host management functionalities. Fabric-domain signal collection and dequeue rate updates are facilitated using the DOCA PCC SDK [68]. For host-domain signal collection, we utilize the RPC and Command Queue mechanisms in the DOCA DPA SDK [60]. Peer-domain signals are managed using the FlexIO SDK [103] for out-of-band communications. Additionally, all analytic, approximation, and arithmetic libraries are implemented in C code, compatible with the DPA RISC-V toolchain.

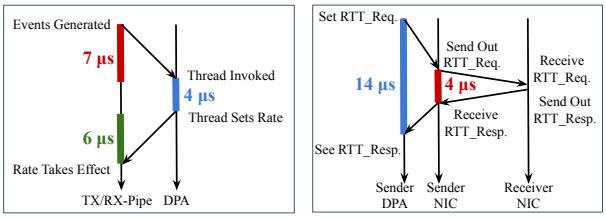
We operate most SCR functionalities on the DPA, only managing very lightweight tasks on the host x86 CPU and none on the DPU ARM cores. By default, we set up 25 DPA threads to handle signals from the fabric domain, including TX, ACK, CNP, NACK, and RTT. One DPA thread establishes communication with the host, and two manage telemetry tasks: one for ping and another for pong. Additionally, two threads exchange signals between peer domains—one for receiving and another for sending. The host CPU is required only to initialize the DPA process and relay the host domain signals to the DPA, for which a single x86 CPU core is sufficient.

A.11 Guidelines for Implementing Other Functionalities

In addition to the aforementioned functionalities, SCR has the potential to support various other customizations. Here, we provide general implementation guidelines based on our analysis and experience.

Timescale of Control Feedback Matters. While SCR can produce per-packet signals at data plane speed, event processing might lag behind such high data rates. We identify three main components contributing to the overall latency of making control decisions: 1) Event triggering, 2) The computation procedure itself, and 3) The time it takes for control feedback to take effect.

We conducted a microbenchmark to quantify the delay breakdown, as shown in Figure 22. For in-band events like TX/ACK events (Figure 22a), event triggering, from the hardware TX/RX pipe generating the events to the DPA invoking the thread to process, could take up to 7 μs. This latency is due to event queuing, event multiplexer operation, and loading the program onto the DPA thread. To accelerate event triggering, we can keep the DPA thread busy waiting rather than relying



(a) In-Band Events (b) Out-of-Band RTT Events

Figure 22: **Event Processing Latency Breakdown.**

on interrupt-like triggering, which can hide the latency. The second component of latency is the processing itself, which is heavily influenced by the complexity of the control laws. Here, we measure the Swift control law with some additional bookkeeping tasks, like maintaining per-flow dequeue rate, which takes about 4 μs. After making the decision for the new rate, the DPA thread writes new values to hardware registers. We measure how long it takes for the changes to take effect, which may require up to 6 μs to start affecting the data planes. This is longer than expected, and we anticipate that next-gen DPA hardware can optimize this rate-installing procedure. For out-of-band events like RTT (Figure 22b), besides the fabric delay, there are two components: 1) The latency between the DPA setting the request bit and the RNIC actually sending out the RTT request. 2) The delay between the RNIC receiving the RTT response and invoking the DPA thread to process. These two delays can take up to 10 μs, which can be optimized by the DPA thread busy waiting for the RTT response and vendors optimizing the delay between the request set and the request sent out. Overall, we observe that event processing could lag behind the data plane by a few microseconds even after optimization. Control lagging behind data is widely observed for high-speed line rates, not unique to our setting but also in traditional in-kernel congestion control for TCP/IP [104–106]. We propose two guidelines to address the lagging issues: 1) Congestion control algorithms would be better designed to work at the RTT granularity, as more fine-grained control could lead to inaccuracies. 2) If algorithms still aim for sub-RTT control, they should be designed to tolerate signaling coalescing, that is, folding consecutive signals into one event. This approach can help hide latency by combining multiple processing steps into one. Control laws should consider coalesced signals, similar to how Swift considers coalesced ACKs.

Keep Simple yet Effective Control Law. We evaluate simplicity from several aspects: 1) the computation of the control logic, 2) the memory access for per-flow states, and 3) the types of required signals. As we measured above, the latency for computing is positively proportional to the instruction counts of the control logic. Although SCR provides general-purpose computation to implement diverse logic, we still need to accelerate the control logic with approximation techniques to ensure faster reactions. Besides the instruction count, memory access is also expensive in terms of time cost. Reducing the memory footprint, such as fitting into the L1 cache, can reduce the time overhead associated with memory access. On the type of signals, we propose leveraging the signals supported by SCR, such as in-band and out-of-band signals. Although some complex signals like switch-aided in-network telemetry can enhance precision, they come with the cost of end-host processing cycles and switch customizations.

A.12 Suggestions for Next-Gen Datapath Accelerator

Based on our experience implementing SCR over DPA, we suggest improvements for the next-gen hardware in the following areas:

- Decrease event-triggering and rate-installing latency: As we described in the §A.11, the latency of event invocation and rate update is crucial for the timely response of the control loop. Next-gen DPA should optimize program loading time and reduce latency for user-invoked DPA threads via RPC mechanisms (*e.g.,* Mailbox RPC [68]).
- Expand the set of signals/events accessible from the RNIC. Diverse signals are essential to enable flexible control laws. For instance, at present, only in-band TX events are accessible. Rx events indicating the timestamp and bytes received would provide valuable additional signals. Currently, the RX pipe does not provide the API for the DPA to access INT fields. Also, expanding the set of NIC counters accessible for DPA would be beneficial. Currently, only TX/RX bytes counters are available. For instance, exposing utilization counters for RNIC micro-architecture resources accessible to DPA, as Neo-Host [107] enables, is promising.
- Expand the set of control actions. Currently, only the rate and RTT requests are considered control actions. As previously mentioned, incorporating additional control actions such as modifying the RoCEv2 header, like adjusting the source port or setting the ECN bits, would be beneficial.
- Streamline and improve the flexibility of DPA process management. The DOCA PCC process is currently the only one capable of adjusting the rate. However, at present, it cannot be easily extended to support other functionalities such as accessing host memory or operating raw Ethernet APIs. As a result, we must set up other DPA processes to access host memory and operate raw Ethernet APIs, which introduces some overhead in inter-process communication (IPC).

A.13 Extended Analysis for Prior Work

Table 6 is an extended list of related works, and we analyze the feasibility of supporting these customizations using SCR.

Prior Work	Why Customization	Prior Customization	Feasibility and Limitations using SCR
Microsoft: DCQCN [2]; Large-Scale Deployment [3]; Storage [1]	CC / FC	ASIC/ Tuning	DCQCN and PFC are SCR default strategies.
AWS: SRD [5]	CC / Re. / MP	Bespoke HW	SCR explores delay-based CC and multi-path routing similar to SRD.
Google: Falcon [4]; Swift [10]; PLB [44]; Carousel [71]; SNAP [108]	CC / Re. / MP	Bespoke HW	SCR showcases the implementation of Swift.
Ultra Ethernet Consortium [8]	CC / Re. / MP	Bespoke HW	Case studies of SCR follow design proposals by UEC.
Hyperscale Issues [45]	CC / FC / More	-	SCR demonstrates some solutions for issues reported.
Alibaba: Solar [99]; Pangu [109]	CC / Re. / MP	Bespoke HW	Solar utilizes UDP Src Port for multi-pathing and per-ACK CC requiring several path conditions (*e.g.,* RTT), which are supported by SCR.
ZeroNIC [110]	Data/Ctrl	FPGA	SCR supports data/ctrl path separation for RDMA.
MLT [23]	Re.	Userspace	SCR can monitor packet loss explored by MLT, but current BF3 has no APIs for SCR to program reliability logic.
ACC [11]	CC	Simulation	SCR supports per-ACK CC like ACC.
CASSINI [111, 112]	CC	Tuning	SCR supports encoding hints from ML workloads to the CC logic.
MegaScale [46]	CC	ASIC	MegaScale proposes ECN-RTT hybrid CC, which is supported by SCR.
Harmonic [17]	Iso. / FC	FPGA	SCR can support rate limiter logic required by Harmonic.
Host CC [98]	CC	In-Kernel	Host memory contention signal is one of SCR host-domain signals.
ConWeave [113]	MP	Switch	SCR focuses on endhost, which is orthogonal to switch innovations.
Hostping [114]	Iso.	SW	SCR applies the manner of Hostping to probe host PCIe utilization.
SRNIC [26]	Scalability	FPGA	SRNIC focus on RNIC connection scalability, which is orthogonal to the SCR scope.
Flor [24]; X-RDMA [49]	CC / Re.	Userspace	Flor enforces the customization at the message level above the transport layer while SCR directly customizes the transport. Flor enforces message-granular control while SCR enforces packet-granular control.
Hermes [115]	CC	Switch	SCR is compatible with Hermes mechanisms of incast control on switches.
CORN [116]	CC	Userspace	CORN enforces CC logic above RoCEv2 in userspace while SCR directly customizes RNIC CC.
Justitia [18]	ISo.	Userspace	Justitia investigates the RDMA multi-tenancy issues. SCR enforces multi-tenant isolation via customizing the QP scheduler. Justitia enforces message-granular control while SCR enforces packet-granular control.
EQDS [16]	CC / Re. / MP	Tunneling	SCR explores receiver-driven credit schemes similar to EQDS.
BFC [14]	FC	Switch	SCR can benefit from switch-side fine-grained flow control.
TCD [12]	CC	Switch	SCR can benefit from accurate congestion detection on the switch side.
Tonic [117]	CC / Re.	FPGA	Tonic explores FPGA-based programmable CC. SCR is based on general-purpose cores to re-program CC.
PCN [118]	CC	Simulation	SCR provides ECN signals which are required by PCN.
1RMA [25]	CC / Re.	Bespoke HW	1RMA requires bespoke HW.
HPCC [9]	CC	FPGA	Currently, BF3 does not provide INT signals for SCR. Alternatively, SCR utilizes out-of-band packets to carry in-network information.
IRN [22]	Re.	FPGA	Currently, retransmission logic is handled by the RNIC data path. NACK signals are available for SCR.
MP-RDMA [20]	MP	FPGA	SCR demonstrates multi-path selection at the sender side.
Tassel [70]	CC	FPGA	SCR can benefit from a better rate limiter.
TX-Arch [56]	Iso.	FPGA	TX-Arch investigates the RNIC multi-tenant contention. SCR enforces the fairness of the QP scheduler.
RCC [51]	CC	Simulation	SCR demonstrates receiver-driven control loop.
Avatar [119]	CC	Userspace	SCR can enforce more fair CC without additional userspace processing.
Shen et al., [120]	Iso.	Userspace	SCR enforces fairness for multi-tenant RDMA without additional userspace processing.
UL-MPRDMA [21]	MP	Userspace	SCR supports probing multi-path real-time conditions.
sRDMA [58]	Re.	Userspace	SCR supports customizing the QP scheduler without additional userspace overhead.
RoGUE [28]	CC / Re.	Userspace	SCR supports customizing CC.
Le et al., [121]	Iso.	Tunning	SCR supports tuning DCQCN.
LDCP [122]	CC	PCC	PCC is submodule of SCR.
RL-CC [123]	CC	PCC	PCC is submodule of SCR.
Restructuring CC [104, 105]	CC	In-Kernel	SCR explores restructuring CC in the context of RDMA transport.
pHost [53]; NDP [52]; Homa [15]	CC / FC	In-Kernel	SCR explores receiver-driven control loop.

(*CC* denotes Congestion Control. *FC* denotes Flow Control. *Iso.* denotes Isolation. *Re.* denotes Reliability. *MP* denotes Multi-Pathing.)

Table 6: **Prior Innovations in RDMA Transport Customization and Analyzing Feasibility with the SCR Framework.**

SIRD: A Sender-Informed, Receiver-Driven Datacenter Transport Protocol

Konstantinos Prasopoulos
EPFL

Ryan Kosta
UCSD *

Edouard Bugnion
EPFL

Marios Kogias
Imperial College London

Abstract

Datacenter congestion control protocols are challenged to navigate the throughput-buffering trade-off while relative packet buffer capacity is trending lower year-over-year. In this context, receiver-driven protocols — which schedule packet transmissions instead of reacting to congestion — excel when the bottleneck lies at the ToR-to-receiver link. However, when multiple receivers must use a shared link (*e.g.,* ToR to Spine), their independent schedules can conflict.

We present SIRD, a receiver-driven congestion control protocol designed around the simple insight that single-owner links should be scheduled, while shared links should be managed with reactive control algorithms. The approach allows receivers to both precisely schedule their downlinks *and* to coordinate over shared bottlenecks. Critically, SIRD also treats sender uplinks as shared links, enabling the flow of congestion feedback from senders to receivers, which then adapt their scheduling to each sender's real-time capacity. This results in tight scheduling, enabling high bandwidth utilization with little contention, and thus minimal latency-inducing buffering in the fabric.

We implement SIRD on top of the Caladan stack and show that SIRD's asymmetric design can deliver 100Gbps in software while keeping network queuing minimal. We further compare SIRD to state-of-the-art receiver-driven protocols (Homa, dcPIM, and ExpressPass) and production-grade reactive protocols (Swift and DCTCP) and show that SIRD is uniquely able to simultaneously maximize link utilization, minimize queuing, and obtain near-optimal latency.

1 Introduction

Datacenter workloads like ML training [32, 62] and disaggregated resource management [30] increasingly demand high-throughput and low-latency networking. While datacenter hardware continues to offer higher link speeds, switch packet buffer capacity is failing to keep up, and SRAM density trends show that this is unlikely to change [17, 24]. The combination imposes a challenge for congestion control, which generally relies on buffering to deliver high throughput. The challenge is compounded by other important concerns such as hardware heterogeneity and cost of operations [23, 35].

The most established approach to managing congestion is by reactively slowing down sender transmission rates after detecting a problem. This is the approach of sender-driven (SD) protocols [4, 8, 9, 12, 44, 47, 50, 54, 67, 69, 73, 74] like DCTCP [8] and Swift [44] in which senders make decisions based on network feedback. The reactive nature of SD protocols often requires several roundtrips to address congestion; a limitation for workloads dominated by small flows [16, 33, 56]. Further, the flow abstraction makes it difficult to reduce latency through message-centric scheduling [10, 13].

To address these limitations, a recent line of work [16, 20, 31, 36, 39, 56, 61] proactively schedules packet transmissions instead of reacting to congestion buildup. Scheduling is typically implemented by having receivers send *credit* packets to senders, which the latter consume to send back data. This enables tight control over ToR-to-Host ports which are usually the most congested [33, 65, 72]. The receiver-driven (RD) approach enables high throughput with limited buffering in dcPIM [16] and near-optimal latency in Homa [56].

The fundamental tension in the design of receiver-driven schemes is how to schedule packets over shared links. Each receiver has exclusive control of its downlink but must share the bandwidth of the network's core and that of sender uplinks with other receivers. In a distributed protocol, this creates scheduling conflicts if receivers do not coordinate. Proposed solutions for this problem include explicit pre-matching of senders and receivers (dcPIM [16]), in-network credit throttling (ExpressPass [20]), and the overcommitment of receiver downlinks (Homa [56]). Despite impressive results, each of these approaches makes a sacrifice in either message latency, protocol complexity, or packet buffer utilization.

We propose SIRD, a receiver-driven design based on the following simple insight: exclusive links (receiver downlinks) should be managed proactively, and shared links (sender up-

*Work done while at EPFL.

links and switch-to-switch links) should be managed reactively. In practice, this means that receivers gather congestion feedback from shared links which gives them the means to coordinate, while still explicitly controlling their downlinks. By deducing the bandwidth availability of shared links, and especially that of senders, receivers can be more precise in their credit allocation. This allows them to drive high link utilization without inducing contention and the corresponding queuing in the network. Low queuing, in turn, enables fast message delivery as the network does not introduce head-of-line blocking.

SIRD implements receiver-driven scheduling that reacts to shared-link congestion feedback using 1) end-host signals to detect congestion on sender uplinks, and 2) ECN [63] to detect congestion in the network's core. Congestion information flows via data packets to receivers and is fed into two independent control loops, with the most congested determining how much credit can be allocated to any given sender. SIRD's design is end-to-end and does not require the use of in-network priority queues or unconventional switch configuration.

We implement SIRD in 4300 SLOC within Caladan [29], a state-of-the-art dataplane operating system. We show that SIRD can efficiently allocate credit, minimize queuing, and implement round-robin and SRPT policies at 100Gbps.

We use network simulation to systematically compare SIRD to three state-of-the-art proactive protocols (Homa [56], dcPIM [16], and ExpressPass [20]) and two production-grade reactive protocols (DCTCP [8] and Swift [44]). Our evaluation shows that SIRD can simultaneously maximize link utilization, minimize queuing, and obtain near-optimal latency. Specifically: (1) SIRD causes 12× less peak top-of-rack (ToR) buffering than Homa yet achieves competitive latency and utilization. (2) Compared to dcPIM, SIRD has no message exchange rounds before sending and outperforms it in link utilization, peak ToR buffering, and tail latency by 9%, 43%, and 46%. (3) SIRD achieves 10× lower tail latency and 26% higher utilization than ExpressPass. (4) SIRD outperforms DCTCP and Swift across the board, especially in incast-heavy scenarios.

2 Background

The main goal of congestion control (CC) is to allocate network resources in a work-conserving manner while avoiding packet loss. There are also other, at times conflicting, CC goals, *e.g.,* fairness, flow completion time or deadlines, and multi-tenancy.

Existing datacenter CC schemes for Ethernet networks can be broadly split into two categories: reactive and proactive. In reactive (also sender-driven (SD)) schemes [8–10, 44, 47, 50, 54, 67, 74], senders control congestion by participating in a distributed coordination process to determine the appropriate transmission rates. Senders obtain congestion information through *congestion signals* like ECN [8, 63, 67], delay [44, 54], and in-network telemetry [47, 69].

In proactive schemes [5, 13, 16, 20, 31, 36, 46, 56, 61] bandwidth is allocated explicitly, either globally, or in a distributed manner. Global approaches [46, 61], use a centralized arbiter that controls all transmissions and hence face scalability challenges. Distributed proactive designs operate either end-to-end [13, 16, 31, 56] or with switch involvement in credit management [5, 20]. In end-to-end protocols, which are receiver-driven (RD), each receiver explicitly schedules its downlink by transmitting special-purpose credit tokens to senders. The credit rate can be explicitly controlled by a pacer [20, 36] or be self-clocked [16, 31, 56].

2.1 Exclusive and shared links

Packets flowing to a receiver host can experience congestion either at the ToR-host link, which is exclusive to the receiver, or at host-ToR uplinks and the network core, which are shared among multiple receiving hosts.

Exclusive links: In the context of RD protocols, ToR-to-Host links (downlinks) can be seen as exclusively controlled by a single entity, the receiver. By controlling the rate of credit transmission, a receiver can explicitly control the rate of data arrival - assuming the bottleneck is the downlink. In fact, downlinks are the most common point of congestion in datacenter networks [33, 44, 65, 72]. Congestion at the ToR downlink is the result of incast traffic from multiple senders to one receiver. In turn, incast is the result of the fan-out/fan-in patterns of datacenter applications [25].

RD schemes excel in managing incast traffic because each receiver explicitly controls the arrival rate of data [13, 16, 20, 31, 36, 56]. This level of control also allows RD schemes to precisely dictate which message should be prioritized at downlinks and can even factor-in application requirements [60]. Recent work has delivered significant message latency improvements by scheduling messages based on their remaining size (SRPT policy) [31, 39, 56]. In contrast, SD schemes [8–10, 44, 47, 50, 54] treat downlinks as any other link. Senders must first detect incast at a receiver's downlink and then independently adjust their rates/windows such that the level of congestion falls below an acceptable target.

Shared links: Unlike exclusive downlinks, core and sender-ToR link bottlenecks pose a challenge for RD schemes. Shared link bottlenecks can appear when multiple receivers concurrently pull data packets over the same link. Before discussing existing approaches to deal with this key challenge, we delve deeper into the specifics of shared-link congestion.

When congestion occurs in the network *core*, multiple flows from unrelated senders and receivers compete for bandwidth on core links. Note that although the core may comprise multiple tiers, it is fundamentally shared infrastructure. Congestion at the core of a fabric is less common than at down-

links [33, 56, 65] but can still occur due to core network oversubscription. Oversubscription can be permanent to reduce cost [65] or transient due to component failures. Congestion at the core can also occur because of static ECMP IP routing decisions that cause multiple flows to saturate one core switch and one core-Tor downlink, while other core switches have idle capacity [7, 35].

Uplink congestion occurs due to the fan-out of multiple flows to different receivers or due to bandwidth mismatch between the sender's uplink and the receiver's downlink. Because the resulting packet buffering is in hosts and not in the fabric, sender congestion is a less severe problem from the perspective of packet loss. For RD schemes, uplink congestion is known as the *unresponsive sender problem* [56] and leads to degraded throughput as independent scheduling decisions of receivers may conflict, wasting downlink bandwidth. For example, two receivers may both decide to schedule the same sender A at the same time, even if one of the receivers could be receiving data from sender B. Whereas the term *unresponsive* has been used in the context of SRPT scheduling, where messages are transmitted to completion, we will use *congested sender* as a general description of uplink congestion.

RD protocols have proposed various mechanisms for overcoming the tension between independent receiver scheduling and the fact that some links are shared. One of the earliest designs, pHost [31], employs a timeout mechanism at receivers to detect unresponsive senders and direct credit to other senders. NDP [36] employs very shallow buffer switches and eagerly drops packets, using packet trimming to recover quickly. NDP receivers handle uplink sharing by only crediting senders if they transmit data packets - whether dropped or not. pHost and NDP do not explicitly deal with core congestion and, further, analysis by Montazeri *et al.* [56] showed that neither of the two achieves high overall link utilization. Homa [56] introduced *controlled overcommitment* in which each receiver can send credit to up to k senders at a time. Homa achieves high utilization as it is statistically likely that at least one of the k senders will respond. However, it trades queuing for throughput by meaningfully increasing the amount of expected inbound traffic to each receiver. To let short messages bypass network queues, Homa leverages switch priority queues, which are typically used for application-level QoS guarantees [12, 37, 44]. dcPIM [16] employs a semi-synchronous round-based matching algorithm where senders and receivers exchange messages to achieve a bipartite matching. This coordinates the sharing of sender uplinks but congestion at the core is only implicitly addressed by the protocol's overall low link contention. The downside of this link-sharing approach is message latency. dcPIM delivers small messages quickly by excluding them from the matching process. However, messages larger than the bandwidth-delay-product (BDP) of the network must wait for several RTTs before starting transmission. ExpressPass [20] manages all links, exclusive and shared, via a hop-by-hop approach which configures switches to drop excess credit packets, which in turn rate limits data packets in the opposite direction. To reduce credit drops, ExpressPass uses recent credit drop rates as feedback to adjust the future sending rate, and in this way also improves utilization and fairness across multiple bottlenecks. Out of the designs discussed so far, ExpressPass is the only one that explicitly manages core congestion and can operate with highly oversubscribed topologies. ExpressPass's hop-by-hop design helps it achieve near-zero queuing [20] but is more complex to deploy and maintain due to its switch configuration and path symmetry requirements.

2.2 The impact of ASIC trends on buffering

Congestion control protocols depend on buffering to offset coordination and control loop delays and, as a result, face a throughput-buffering trade-off. Maximum bandwidth utilization can trivially be achieved with high levels of in-network buffering, but at the cost of queuing-induced latency and expensive dropped packet retransmissions. Conversely, low buffering can lead to throughput loss for protocols that are slow in capturing newly available bandwidth.

High-speed packet buffering is handled by small SRAM buffers in switch ASICs, the size of which is not increasing as fast as bisection bandwidth. For example, nVidia's top-end Spectrum 4 ASIC has a 160MB buffer, which corresponds to 3.13MB per Tbps of bisection bandwidth [58]. The previous 12.8Tbps and 6.4Tbps top-end Spectrum ASICs were equipped with 5MB and 6.6MB per Tbps respectively [52, 53] (full list in Appendix A). Unfortunately, future scaling of SRAM densities appears unlikely given CMOS process limitations [17, 24]. In parallel, datacenter round-trip times (RTT) are not falling as they are dominated by host software processing, PCIe latency, and ASIC serialization latencies. Consequently, as link speeds increase, CC protocols must handle higher BDPs with less switch buffer space at their disposal.

To better absorb instantaneous bursts of traffic, switch packet buffers are generally shared among egress ports. Some ASICs advertise fully shareable buffers while others implement separate pools or statically apportion some of the space to each port [33]. On top of the physical implementation, various buffer-sharing algorithms dynamically limit each port's maximum allocation to avoid unfairness [8]. However, if a large part of the overall buffer is occupied, the per-port cap becomes more equitable, limiting burst absorbability [33].

Figure 1 provides some context by comparing the buffer use of Homa, a state-of-the art RD protocol, with recent switch buffer capacities under the Websearch workload [10]. The dotted lines represent the per-port (left) and completely shared (right) buffer capacities of the hypothetical switches, adjusted on a per-unit basis to the port radix of our simulation methodology (§6.2). In practice, the buffer is neither partitioned nor fully shared; however, between the two extremes, hardware trends are constraining the throughput-buffering trade-off.

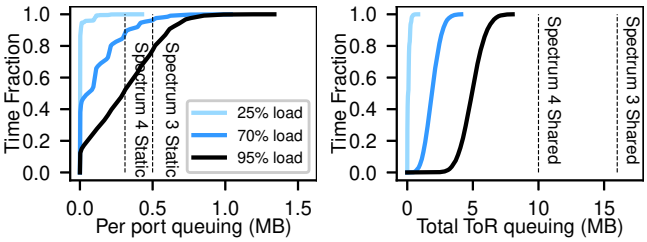


Figure 1: Homa queuing CDFs under various network loads for workload Websearch [10]. The dotted lines represent the switch buffer size adjusted to the actual radix of our simulated ToR; see §6.2.

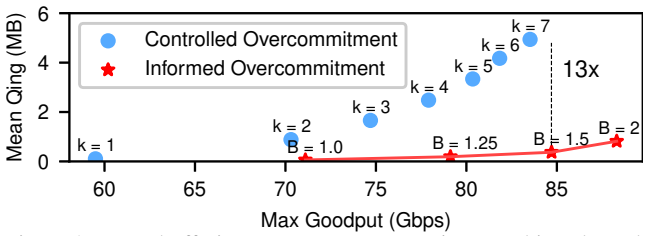


Figure 2: Mean buffering at ToRs versus maximum achieved goodput when sweeping the overcommitment parameter for SIRD (informed overcommitment) and Homa (controlled overcommitment). Results obtained in simulation by running the Websearch workload at 95Gbps on 100Gbps links across 144 servers; see §6.2.

3 SIRD Design Pillars

SIRD is an end-to-end receiver-driven scheme that manages exclusive links proactively and shared links reactively. In practice, this means that receivers perform precise credit-based scheduling when the bottleneck is their downlinks and use congestion feedback to coordinate over shared links. SIRD does not face the limitations of prior work as it:

- is end-to-end, with all decision-making happening at end hosts, and does not rely on advanced switch features or configuration.
- achieves high throughput by using sender and core link congestion feedback to allow receivers to direct credit to links with spare capacity.
- causes minimal buffering in the network by drastically reducing the need for downlink overcommitment.
- does not need switch priority queues to deliver messages with low latency, thanks to minimal buffering.
- can start message transmission immediately without a prior matching stage.
- explicitly tackles core congestion through its shared link management approach.

Efficient credit allocation: SIRD's design allows receivers to send the appropriate amount of credit depending on the real-time availability of the bottleneck link. For downlinks the task is trivial as the link's capacity is managed by one receiver. However, when the bottleneck is shared (sender uplink or core link), SIRD receivers detect competition for bandwidth and adjust the amount of issued credit dynamically. For example, if a sender is the bottleneck for two receivers, each will allocate the appropriate amount of credit for their share of the sender's bandwidth. This approach makes the distribution of credit efficient because it is given to hosts that can promptly use it rather than being accumulated by congested senders. As a result, SIRD can achieve high link utilization using a small amount of credit, *i.e.,* with little overcommitment.

SIRD combines efficient credit allocation and overcommitment in *informed overcommitment*. Each receiver is allotted a limited amount of available credit B. The minimum valid value of B is $1 \times BDP$ as this is the amount of credit required to pull $1 \times BDP$ of inbound traffic and fully utilize the downlink. Higher values of B lead to downlink overcommitment, which increases tolerance to congested shared links but also increases expected queuing.

Figure 2 highlights the benefit of efficient credit allocation in SIRD by comparing the throughput-buffering trade-off across equivalent overcommitment levels for Homa (k), which introduced and uses RD overcommitment, and SIRD (B) under maximum network load. Informed overcommitment yields the same overall goodput despite overcommitting downlinks $14\times$ less, which leads to $13\times$ lower queuing. The goodput-queuing trade-off benefit is maximum at high network load, under which congestion and loss are most likely, and naturally fades as load lessens.

SIRD implements informed overcommitment through a control loop at each receiver, which dynamically adjusts credit allocation across senders based on sender and network feedback. When a sender is concurrently credited by multiple receivers, the sender receives credit faster than it can consume it, causing it to accumulate. This is undesirable as receivers have a limited amount of credit to distribute. Equivalently, limiting accumulation makes credit available to other senders that can actually use it. SIRD's control loop rebalances credit by setting an accumulated credit threshold, $SThr$, that is similar in spirit to DCTCP's marking threshold [8], Swift's target delay [44], or HPCC's η [47]. At each sender, while the amount of accumulated credit from all receivers exceeds $SThr$, a bit is set in all outgoing data packets. Based on arriving bit values, each receiver adjusts the maximum amount of credit that can be allocated to the congested sender in an additive-increase/multiplicative-decrease (AIMD) manner. In this paper, we use DCTCP's [8] AIMD algorithm.

SIRD uses the same mechanisms to handle core congestion with ECN as the signal. Receivers read the CE bit of data packets and adjust per-sender credit limits using AIMD.

Each receiver runs two AIMD algorithms in parallel, one for senders and one for the core, each with its own input signal. The algorithm instance of the most congested area determines credit allocation. This separation of signals allows SIRD to use different signal/algorithm combinations to handle bot-

tlenecks at hosts and switches. Beyond ECN, SIRD can use signals such as end-to-end delay on infrastructures with timestamping support [44] or In-Band Telemetry [47, 69]. In this paper, we use the same proven and simple combination for both and leave the exploration of more complex algorithms and signals to future work.

Starting at line rate: Prior work [31, 36, 39, 56] has demonstrated the latency benefits of starting transmission at line rate, without a gradual ramp-up or preceding control handshakes that take at least one RTT. SIRD's credit allocation mechanism operates *while transmission progress is made*, and thus, senders immediately start at line rate, sending the first BDP bytes *unscheduled* (without waiting for credit).

We further optimize the design based on the following simple observation: small messages benefit the most from unscheduled transmission since their latency is primarily determined by the RTT while throughput-dominated messages see minimal gain. For example, in the absence of queuing, delaying transmission by one RTT increases the end-to-end latency of a message sized at $10 \times BDP$ by 9% compared to 200% for a single-packet message. Therefore, to reduce unnecessary bursty traffic and queuing, SIRD senders do not start transmitting messages larger than a configurable threshold (UnschT) before explicitly receiving credits from the receiver. For messages smaller than UnschT, senders send the first $min(BDP, msg_size)$ bytes without waiting for credit.

Switch priority queues not required: By efficiently allocating credit, SIRD achieves high link utilization with minimal fabric queuing, thereby largely eliminating its impact on latency. Thus, SIRD can be deployed without any in-network QoS support, though it can benefit in terms of tail latency from having a total of two priority levels.

4 SIRD Design

At a high level, SIRD is an RPC-oriented protocol, similar to other recently proposed datacenter transports [16, 31, 56]. SIRD may be used to implement one-way messages or remote procedure calls [42, 43, 56]. SIRD assumes that the length of each message is known or that data streams will be chunked into messages. SIRD further assumes that ECN is configured in all network switches, with the ECN threshold set according to DCTCP best practices [8]. SIRD is designed to be layered on top of UDP/IP for compatibility with all network deployments. The UDP source port is randomly selected for each packet for fine-grain load balancing, allowing an ECMP network to behave as effectively as random packet spraying [26]. We make no assumption of lossless delivery, in-network priority queues, or smart NICs and switches.

SIRD defines two main packet types: **(1)** `DATA`: a packet that contains part of the payload of a message. Data packets can be scheduled, requiring credit to be sent, or unscheduled. Messages with size $> UnschT$ are scheduled and make an initial credit request by sending a zero-length `DATA` packet; **(2)** `CREDIT`: a control packet sent by the receiver to the sender to schedule the transmission of scheduled `DATA`.

Generally, the flow of packets consists of credit flowing from receivers to senders and data flowing in the opposite direction. Credit leaves the receiver in a `CREDIT` packet, is used by the sender, and returns with a scheduled `DATA` packet.

4.1 Credit Management

Each SIRD receiver maintains two types of credit buckets that limit the amount of credit it can distribute: a global credit bucket and per-sender credit buckets.

The *global credit bucket B* controls overcommitment by capping the total number of outstanding credits that a receiver can issue. Configuring $B \geq BDP$ is necessary to enable link saturation. Further, B bounds the queuing length from scheduled packets to $B - BDP$ bytes at the ToR's downlink. Our analysis (§4.2) and evaluation (§6) show that setting B as low as $1.5 \times BDP$ is sufficient for high link utilization.

Each receiver also maintains a credit bucket per sender machine it communicates with. The *per-sender credit bucket* caps the number of outstanding credits the receiver can issue to a sender. Informed overcommitment is implemented by adjusting the size of the per-sender credit bucket according to the level of congestion at the core and the sender (max $1 \times BDP$). Reducing the bucket size means that less of the receiver's total credit can be allocated to a congested source or path, and thus, more is available for other senders.

4.2 Informed Overcommitment

Informed overcommitment uses two input signals that communicate the extent of congestion in the network and at senders. The signals are carried in `DATA` packets and are the ECN bit in the IP header set by the network and a bit in the SIRD header set by the sender. Each receiver runs two separate AIMD (additive-increase, multiplicative decrease) control loops and uses the most conservative of the two (similar to Swift [44]) to adjust per-sender credit bucket sizes. Each control loop is configured with its own marking threshold ($NThr$ and $SThr$). $NThr$ should be set according to DCTCP best practices [8] to limit queuing in the network core. Note that $NThr$ is much higher than the maximum allowed $B - BDP$ of persistent queueing at ToR switches. Thus, ToR switches never have to mark ECN.

$SThr$ should be set to limit the amount of credit a sender can accumulate, thus allowing receivers to efficiently distribute their limited aggregate credit. Intuitively, $SThr$ determines the level of accumulated credit the control loop is targeting when a sender is congested, *i.e.,* receiving credit faster than it can use it. Setting it too high means each sender can accumulate substantial amounts of credit, and consequently, B would

need to be configured higher to increase aggregate credit availability. Conversely, setting $SThr$ too low does not allow the control loop any slack when converging to a stable state and can cause throughput loss.

To understand the relationship between $SThr$ and B, we analytically examine a simple congested-sender scenario from the perspective of a receiver R to find: how much total credit (B) does R need when receiving from k congested senders, to still have enough credit available to saturate its downlink. Assuming each congested sender sends to f receivers in total, its available uplink bandwidth for R is BW/f. Assuming uniform link speeds, we are interested in the case where senders are the bottleneck and cannot saturate R, or:

$$BW_{supply_k} < BW_{demand_R} => \frac{BW}{f} < \frac{BW}{k} => f > k \quad (1)$$

In this case, each of the k senders accumulates up to $SThr$ credit in stable state, or $SThr/f$ from each receiver assuming an equal split (see §4.4). Therefore, to be able to saturate its downlink, R's B must be large enough to allow $1xBDP$ of credit to be in flight despite accumulation at congested senders, or:

$$B \geq BDP + \sum_{k} \frac{SThr}{f} \; ; \; f \geq 2, f > k \quad (2)$$

If k_{max} is the number of congested senders that maximizes the right side of the inequality, then $f_{max} = k_{max} + 1$ (f in denominator) and the maximum value of the term is:

$$\sum_{k}^{k_{max}} \frac{SThr}{k_{max}+1} = \frac{SThr}{k_{max}+1} \sum_{k}^{k_{max}} 1 = SThr \frac{k_{max}}{k_{max}+1} < SThr \quad (3)$$

It follows that, in steady state, R can account for any number of congested senders as long as $B \geq BDP + SThr$. Under dynamic traffic patterns (see §6.2.4) higher values of B can help increase the supply of credit in transient phases. Further, policies other than fair sharing can loosen this property. For example, if senders strictly prioritize some receivers, then, in the worst case where R is de-prioritized by all k senders, Equation 2 loses f from the denominator and B depends on the number of worst-case congested senders.

Table 1: Core configuration parameters.

UnschT	Messages that exceed UnschT in size ask for credit before transmitting.
B	Per-receiver global credit bucket size. Caps credited-but-not-received bytes.
NThr	ECN threshold, configured as for DCTCP.
SThr	Sender marking threshold (*sird.csn*).

Algorithm 1 Receiver Logic

Variables:
- $\boldsymbol{b}$: consumed credit from global receiver bucket of size B,
- $\boldsymbol{sb_i}$: consumed credit from the bucket of sender i,
- $\boldsymbol{senderBkt_i, netBkt_i}$: Sender and network credit bucket size,
- $\boldsymbol{rem_i}$: Requested but not granted credit for sender i.

1: **procedure** onDataPacket(pkt, i) ▷ i:sender
2: $credit \leftarrow getCredit(pkt)$
3: $b \leftarrow b - credit$
4: $sb_i \leftarrow sb_i - credit$
5: $senderBkt_i \leftarrow Sender_{AIMD}(senderBkt_i, pkt.sird.csn)$
6: $netBkt_i \leftarrow Net_{AIMD}(netBkt_i, pkt.ip.ecn)$
7: **end procedure**
8: **procedure** onSendCreditTick() ▷ Runs when $b + min(rem_i, MSS) \leq B$
9: $senderList \leftarrow activeSenders.filter($
$sb_i + min(rem_i, MSS) \leq min(senderBkt_i, netBkt_i))$
10: $s \leftarrow policySelect(senderList)$
11: $credit \leftarrow min(rem_i, MSS)$
12: $sendCredit(s, credit)$
13: $b \leftarrow b + credit$; $sb_i \leftarrow sb_i + credit$; $rem_i \leftarrow rem_i - credit$
14: **end procedure**

Algorithm 2 Sender Logic

Variables: $\boldsymbol{c_r}$: available credit for outbound messages to receiver r

1: **procedure** onCreditPacket($r, credit$)
2: $c_r \leftarrow c_r + credit$
3: **end procedure**
4: **procedure** sendData()
5: $rcvrList \leftarrow activeReceivers.filter(c_r > 0)$;
6: $(r, dataPkt) \leftarrow policySelect(rcvrList)$
7: $dataPkt.sird.csn \leftarrow (\sum_i (c_i) \geq SThr)$
8: $c_r \leftarrow c_r - dataPkt.size$
9: $send(r, dataPkt)$
10: **end procedure**

4.3 Congestion Control Algorithm

Algorithm 1 describes the behavior of a SIRD receiver. When credit is available in the global bucket and on the command of the credit pacer (§4.4), the receiver tries to allocate credit to an active sender (ln. 8). It first selects one of the senders with available credit in the per-sender bucket (ln.9) based on policy (ln.10). The receiver sends a `CREDIT` packet (ln. 12) and reduces the available credit in the global and per-sender buckets, and updates the remaining required credit for that message (ln. 13). Whenever a `DATA` packet arrives (ln. 1), if it is scheduled, the receiver replenishes credit in the global bucket (ln. 3) and the per-sender bucket (ln. 4). Then, it executes the two independent AIMD control loops to adapt to a congested sender (ln. 5) and a congested core network (ln. 6), respectively. The receiver sets the size of the per-sender bucket as the minimum of the two values (ln. 9).

Algorithm 2 describes the implementation of the sender-side algorithm for scheduled `DATA` packets. A host can send

data to a receiver only if it has credit from said receiver. Congested senders mark the *congested sender notification (sird.csn)* bit if the total amount of accumulated credit exceeds $SThr$ (ln. 7). The sender can send unscheduled `DATA` packets at any point in time. It is worth noting that SIRD senders naturally handle scenarios where a meaningful portion of uplink bandwidth is consumed by unscheduled packets. Since senders limit credit accumulation by informing receivers to allocate less credit, the transmission rate of scheduled packets converges to the leftover bandwidth.

4.4 Other Design Concerns

SIRD can be configured to schedule for fairness *e.g.,* by crediting messages in a round-robin manner, or for latency minimization, *e.g.,* by crediting smaller messages first (SRPT), or to accommodate different tenant classes. SIRD implements policies at the receiver (ln .10), which is the primary enforcer, and at the sender (ln. 6). By minimizing queuing in the fabric, SIRD does not need to enforce policies there, simplifying the design. Regardless of which policy is configured at senders, SIRD allocates part of the uplink bandwidth fairly across active receivers, as to ensure a regular flow of congestion information between sender-receiver pairs.

SIRD receivers pace credit transmission to match their downlink's capacity. Pacing improves message latency by reducing downlink queuing from scheduled packets even below the tight $B - BDP$ bound but is not needed for correctness.

If switch priority queues are available, `CREDIT` packets are sent over a higher priority lane to further reduce RTT jitter. The unscheduled prefixes of messages smaller than $UnschT$ also use this lane to bypass transient network queueing. SIRD does not require in-network priority queues to deliver high performance but sees tail latency benefits in some cases (§6.2.4).

Packet loss in SIRD will be very rare by design, but the protocol must still operate correctly in the presence of CRC errors or packet drops due to faults or restarts. SIRD employs Homa's [56] retransmission design in which receivers infer loss when no new packets are received for an incomplete message after a period of a few milliseconds. Upon detecting the loss of a scheduled segment, receivers also reclaim the credit allocated for said segment. SIRD's modest credit overcommitment (*e.g.,* 50%) ensures that receivers can continue fully utilizing their bandwidth even in the improbable event where a sizeable chunk of packets is lost. Finally, SIRD incurs no performance penalties from out-of-order packet deliveries as it does not rely on packet order for loss detection and only delivers completed messages to applications.

5 Caladan Implementation

SIRD, with its message-level API and simple credit management design, can be implemented in NIC hardware or in software. The former eliminates PCIe latencies from the RTT, reducing the *BDP*, and allows for nanosecond-scale pacing and delivery of packets on the wire. We demonstrate that the latter is also possible, at 100Gbps, by implementing SIRD in 4300 SLOC as a UDP-based transport protocol in Caladan [29], a state-of-the-art kernel-bypass dataplane operating system and CPU scheduler. We chose Caladan because of its mature network stack and its lightweight green thread abstraction, suitable for building latency-sensitive applications. SIRD is open source and available on GitHub [1].

The SIRD stack follows an asymmetric design where dedicated spinning threads perform specialized tasks. On the receiver-side, each SIRD node has one dedicated thread that runs the control loops and implements downlink scheduling policies, like round-robin or SRPT, by transmitting paced `CREDIT`s according to Algorithm 1. SIRD reduces network queuing by pacing credit at slightly less than the line rate, as proposed in Hull [9]. Dedicated receiver cores read from RX rings and adjust credits, ensuring the system can quickly replenish new credits for redistribution. Each dedicated receiver core has two RX rings, one for packets of scheduled messages and one for packets of unscheduled messages; this helps avoid drops of scheduled packets during high rates of unscheduled traffic. For efficiency, and to sustain 100Gbps line rates, memory allocation and data copying operations are removed from the credit replenishment path to a small set of dedicated threads which copy packets, reassemble messages from packets, which can be received out of order, and deliver messages to applications via inline callbacks.

The sender-side implementation also has a central thread, which accepts `CREDIT` packets and schedules the transmission of all `DATA` packets, whether these are unscheduled or scheduled. This thread implements Algorithm 2 which marks the *sird.csn* bit of `DATA` packets by comparing the amount of accumulated credit to $SThr$. Like the central receiver thread, this thread paces itself to ensure that packets are not given to the NIC faster than it can consume them. The thread accepts new message requests via asynchronous shared-memory queues, where the application API returns error codes if resource limits are exceeded.

6 Evaluation

We evaluate SIRD using our Caladan implementation and through simulation to answer the following questions: **(1)** Can a software implementation of SIRD deliver 100Gbps along with minimal queuing under incast (§6.1.1) and efficient credit management under outcast (§6.1.2)? **(2)** How well does SIRD navigate the throughput-buffering-latency trade-off in larger scale workloads compared to existing work (§6.2.1)? **(3)** Is SIRD's congestion response robust at high load pressure and can it handle core congestion (§6.2.2)? **(4)** Can SIRD deliver messages with low latency in large scale workloads (§6.2.3)? **(5)** How important is SIRD's sender-informed design in maximizing utilization with minimal buffering? How sensitive is

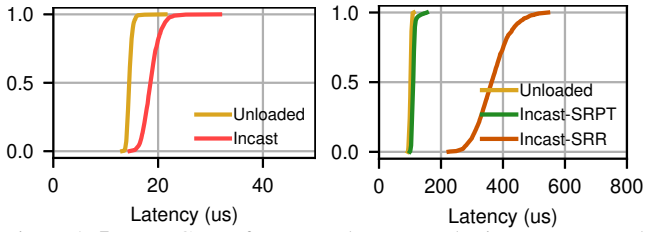


Figure 3: **Incast:** CDF of message latency under incast compared to an unloaded baseline for 8B requests (left) and 500KB requests (right). The incast is formed by six senders transmitting 10MB messages in an open loop.

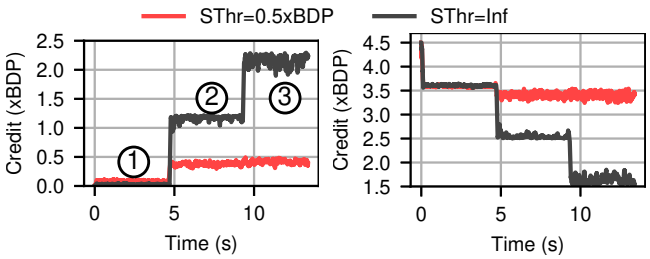


Figure 4: Left: credit accumulated at congested sender. Right: sum of credit available at the three receivers - initial total: $4 \times 1.5 = 4.5 \times BDP$. 100ms moving average. The circled numbers indicate the number of receivers at that stage of the experiment.

SIRD to its parameters (§6.2.4)?

6.1 System Evaluation

We evaluate SIRD at a small scale using Cloudlab's [27] sm110p machines equipped with 100Gbps ConnectX-6 DX NICs and Xeon 4314 CPUs, all located in the same rack. Our implementation uses 9KB jumbo frames and the unloaded RTT of the SIRD-Caladan stack is approximately 18 µs. Latency measurements are end-to-end, measured by the client, and include four request/reply copies to/from the application layer. We sample other system metrics every 2ms. While the size of requests may vary from experiment to experiment, replies are of minimal size.

We configure SIRD parameters as follows: $BDP = 216KB$ (24 jumbo frames), $B = 1.5 \times BDP$, $SThr = 0.5 \times BDP$, $UnschT = 1.0 \times BDP$. We do not use switch priority queues.

6.1.1 Receiver Congestion

SIRD's main objectives when handling incast are eliminating queuing-induced latency for small messages and being able to reduce latency for larger messages through credit-based scheduling. To evaluate this, we run an experiment where six senders saturate a receiver by sending 10MB requests at a rate of 17Gbps each. A seventh sender periodically transmits either 8B or 500KB requests and captures the latency distribution.

Figure 3 (left) plots the round-trip latency experienced by short requests, which are unscheduled. When the receiver is saturated, we observe only a few microseconds of additional latency compared to when the receiver is unloaded, which corresponds to a couple of packets of queuing, either at the ToR or at the host stack. For context, the median latency under load for the same experiment using kernel TCP Cubic was above 1ms. SIRD achieves this result because it limits the number of scheduled inbound bytes to B and thus the extent of downlink queuing to $B - BDP$. Further, since in this case senders respond to credit immediately, credit pacing effectively eliminates downlink queuing, while still achieving 96Gbps in this experiment.

Figure 3 (right) shows the latency of *scheduled* 500KB requests under two receiver scheduling policies. Under SRPT, the receiver allocates credit to the message with the fewest remaining bytes and thus quickly prioritizes 500KB over 10MB requests, leading to near-unloaded latency despite the saturated downlink. SIRD can also implement fairer policies like per-sender round robin ("SRR") as shown. Finally, as it causes minimal network queuing, SIRD can implement such policies without in-network QoS support.

6.1.2 Sender Information

When senders are the bottleneck (outcast), SIRD tackles the congested sender problem by providing congestion feedback to receivers and allowing them to adjust their credit allocation. To evaluate the mechanism's efficacy at 100Gbps, we run an outcast experiment where a single sender sends 10MB messages at full rate to three receivers in a time-staggered manner. Figure 4 (left) shows that, without informed overcommitment ("$SThr = Inf$"), the level of credit accumulated at the congested sender increases with every new receiver. This is because each receiver independently gives the sender $1 \times BDP$ worth of credit to achieve full rate. Through congestion feedback ("$SThr = 0.5 \times BDP$"), receivers coordinate and scale down their credit allocation until credit accumulation at the sender is less than $SThr = 0.5 \times BDP$, on average. Credit, instead, stays at receivers, and can be used to schedule other senders. Figure 4 (right) shows the sum of credit available at the three receivers, each receiving a max-min fair share of the sender's bandwidth. Without informed overcommitment, each receiver allocates $1 \times BDP$ worth of credit to the congested sender. With it, each allocates $(BDP + SThr)/num_rcvers$.

6.2 Simulations

We use network simulation to evaluate SIRD on large-scale workloads and compare it to other protocol designs.

We compare SIRD to 5 baselines: DCTCP [8], a widely deployed sender-driven scheme [33], Swift [44], a state-of-the-art production sender-driven scheme, Homa [56], because of its near-optimal latency and its use of overcommitment, ExpressPass [20], as it employs a hop-by-hop approach to managing credit, and dcPIM [16], as a unique point in the

design space as it explicitly matches senders and receivers. We do not extend Homa with Aeolus [39] for the reasons discussed in [40]. Note that the Homa and dcPIM papers [16,56] already include favorable comparisons to NDP [36], Aeolus [39], PIAS [13], pHost [31], and HPCC [47], thus we did not include them in our evaluation.

We implement SIRD on ns-2 [11], reuse the original ns-2 DCTCP implementation (using the same parameters as [10], scaled to 100Gbps) and the original ns-2 ExpressPass implementation [21], and port the published Homa simulator [55] to ns-2 using the same parameters as in [56] scaled to 100Gbps. The published Homa simulator does not implement the incast optimization [56], which further relies on two-way messages. We use a community Swift simulator, kindly made public [3], and configure its delay parameters to achieve similar throughput to DCTCP, respecting the guidelines [44]. Finally, we use the published dcPIM simulator [15]. SIRD's simulator is open source and available on GitHub [2].

Table 2 lists protocol parameter values. DCTCP and Swift use pools of pre-established connections (40 for each host pair). SIRD approximates SRPT scheduling like Homa and dcPIM. SIRD, Homa, and dcPIM use packet spraying, DCTCP and Swift use ECMP. Homa uses 8 network priority levels, dcPIM 3, and SIRD 2 (§4.4). The initial window of DCTCP and Swift is configured at *BDP* like in [10]. The RTT latency is similar to prior work.

Topology: We simulate the two-tier leaf-spine topology used in previous work [8, 10, 16, 31, 56] with 144 hosts, connected to 9 top-of-rack switches (16 hosts each) via 4 spine switches, with link speeds of 100Gbps to hosts and 400Gbps to spines. We simulate switches with infinite buffers, *i.e.,* without packet drops (1) to avoid making methodologically complex assumptions as drop rates and thus latency and throughput are very sensitive to switch buffer sizes, organization, and configuration [33,40] (see §2.2); and (2) to study the intended mode of operation of the protocols that leverage buffering to achieve high link utilization and operate best without drops. The intentional dropping of credit packets to adjust sender rates by ExpressPass is maintained. SIRD never uses more than a small fraction of the theoretical capacity (§6.2.1) and is not affected by this setup.

Workload: Each host operates both as client and server, sending one-way messages according to an open-loop Poisson distribution to uniformly random receivers (all-to-all). We simulate 3 workloads: **(1) WKa**: an aggregate of RPC sizes at a Google datacenter [28]; **(2) WKb**: a Hadoop workload at Facebook [64]; **(3) WKc**: a web search application [10]. We select them to test over a wide range of mean message sizes of 3KB, 125KB, and 2.5MB respectively.

We simulate these 3 workloads on 3 traffic configurations, for a total of 9 points of comparison. The configurations are: **(1) Balanced:** The default configuration described above. We vary the applied load, which does not include protocol-dependent header overheads, from 25% to 95% of link capacity. **(2) Core:** Same as (1) but ToR-Spine links are 200Gbps (2-to-1 oversubscription). Due to uniform message target selection, $128/144 \approx 89\%$ of messages travel via spines, turning the core into the bottleneck. We consequently reduce the load applied by hosts by $\times 0.89 * 2$ to reflect the network's reduced capacity. This is not meant to reflect a permanent load distribution, but we hypothesize that it is possible transiently. **(3) Incast:** We use the methodology of [16,47] and combine background traffic with overlay incast traffic: 30 random senders periodically send a 500KB message to a random receiver. Incast traffic represents 7% of the total load.

We report goodput (rate of received application payload), total buffering in switches, and message slowdown, defined as the ratio between the measured and the minimum possible latency for each message. In the incast configuration, we exclude incast messages from slowdown results.

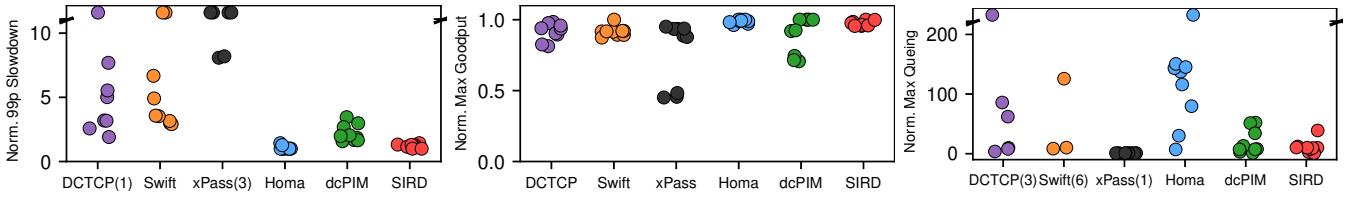
6.2.1 Performance Overview

Figure 5 shows how protocols navigate the trade-offs between throughput, buffering, and latency by plotting their relative performance across all 9 workload-configuration combinations. The best-performing protocol on each of the 9 scenarios gets a score of 1.0 (per metric) and the others are normalized to it, so that goodput is always ≤ 1.0 whereas queuing and slowdown are always ≥ 1.0. We report the highest achieved goodput and peak queuing over all the levels of applied load. We report 99^{th} percentile slowdown across all messages of each workload at 50% applied load which is a level most protocols can deliver in all scenarios. The following figures and calculations do not include cases where a protocol cannot satisfy a specific load level or cannot stop network buffers from growing infinitely.

Overall, SIRD is the only protocol that consistently achieves near-ideal scores across all metrics. Specifically, SIRD causes 12× less peak network queuing than Homa and achieves competitive latency and goodput performance.

Table 2: Default simulation parameters for each protocol.

Prot.	Parameters
all	•RTT(MSS): 5.5µs intra-rack, 7.5µs inter-rack •$BDP = 100KB$; link@100Gbps
SIRD	•B : $1.5 \times BDP$, •$UnschT$: $1 \times BDP$ •$NThr = 1.25 \times BDP$, •$SThr = 0.5 \times BDP$
DCTCP	•Initial window: $1 \times BDP$, •g=0.08 •Marking ECN Threshold: $1.25 \times BDP$
Swift	•Initial window: $1 \times BDP$ •base_target: $2 \times RTT$, •fs_range: $5 \times RTT$ •$\hbar$: $1.25 \times BDP$, •fs_max: 100, •fs_min: 0.1
XPass	•$\alpha = 1/16$ •$w_{init} = 1/16$ •$loss_tgt = 1/8$
Homa	Same as [56] at 100Gbps (incl. priority split)
dcPIM	Same as [16]



(a) Slowdown (b) Goodput (c) Queuing

Figure 5: Normalized goodput, queuing, and slowdown across all 9 configurations. Each metric is normalized based on the best-performing protocol for the given metric and configuration. For queuing and slowdown, lower is better. For goodput, higher is better. Normalized slowdown and buffering are capped at 10× and 200×, respectively, and higher values are plotted in the overflow area. The numbers in parentheses show the number of unstable configurations for each protocol which are not plotted. X-axis jitter is added for visibility. Find the data in Table 4.

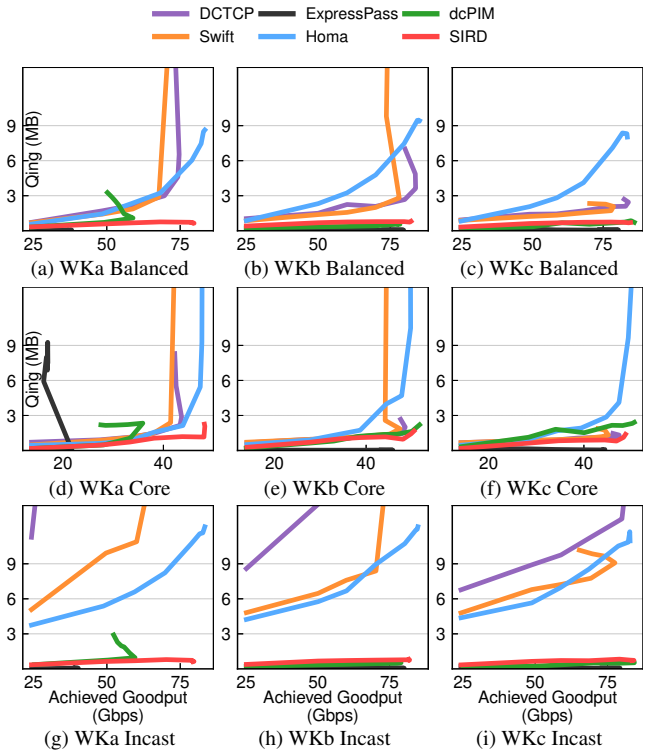
SIRD outperforms dcPIM in message slowdown, and peak goodput and queuing by 46%, 9%, and 43% respectively. ExpressPass causes practically zero queuing thanks to its hop-by-hop design, and 88% less than SIRD, but SIRD delivers 10× lower slowdown and 26% more goodput. Even under full fabric saturation, SIRD induces at most 0.8MB of ToR queuing in receiver-bottleneck scenarios and 2.3MB in core-bottleneck scenarios. Given a packet buffer capacity of 3.13MB/Tbps (see §2.2), these values correspond to a maximum buffer occupancy of 8% and 23%, respectively.

6.2.2 Congestion Response

We now zoom in on how each protocol manages congestion across various levels of applied stress. Figure 6 plots maximum buffering across ToR switches as a function of achieved goodput for balanced (top), core (middle), and incast (bottom) configurations. The reported goodput is the mean across all 144 hosts and reflects the rate of message delivery to applications. ToR queuing covers both downlinks and links to aggregation switches (core).

Overall, SIRD consistently achieves high goodput while causing minimal buffering across load levels, even under high levels of stress. Through informed overcommitment, SIRD makes effective use of a limited amount of credit and achieves high goodput while reducing the need for packet buffer space by up to 20× compared to Homa's controlled overcommitment. Further, SIRD displays a more predictable congestion response compared to dcPIM and ExpressPass. As expected, based on Figure 5a, DCTCP and Swift cause meaningful buffering without achieving exceptional goodput. Mean queuing is qualitatively similar (appendix Figure 13).

dcPIM and ExpressPass sometimes compare favorably to SIRD but only when the mean message size is not small. They both struggle in terms of achieved goodput and buffering in **WKa** for all three configurations (left column). In WKa, ≈ 99% of messages are smaller than $1 \times BDP$ and are responsible for ≈ 40% of the traffic. ExpressPass's behavior is known and discussed in [20]: more credit than needed may be sent for a small message which may then compete for bandwidth with productive credit of a large message. For dcPIM, the likely reason as discussed in [41] is that to send a small

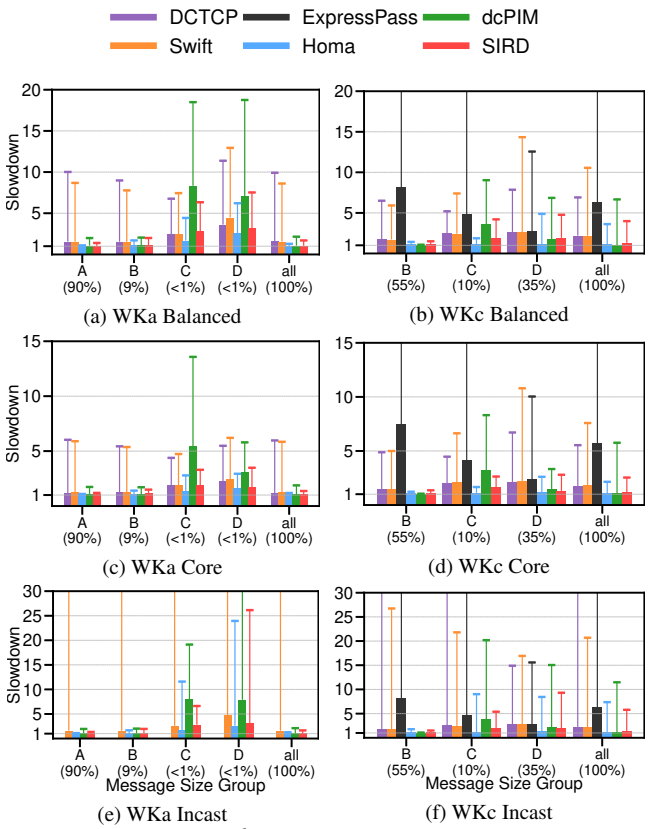


(a) WKa Balanced (b) WKb Balanced (c) WKc Balanced
(d) WKa Core (e) WKb Core (f) WKc Core
(g) WKa Incast (h) WKb Incast (i) WKc Incast

Figure 6: Maximum ToR queuing vs. achieved goodput. Configurations: Balanced (top), Core (middle), Incast (bottom).

message, a sender has to preempt the transmission of a larger message which can cause the receiver of the latter to remain partially idle. SIRD's behavior is consistent across workload message sizes thanks to slight downlink overcommitment which absorbs discontinuities in large message transmission.

In the core configuration (Figure 6 - middle row), where the core is the bottleneck, we observe that SIRD's reactive congestion management, despite sharing the same ECN-based mechanism as DCTCP, achieves steadier behavior because it limits the number of bytes in the network. Homa, though lacking an explicit mechanism to regulate queing at the core, does constrain peak queuing below 19MB (6MB/Tbps of switch BW) because it limits outstanding bytes to the aggregate overcommitment level of receivers. The same applies to dcPIM



(a) WKa Balanced (b) WKc Balanced

(c) WKa Core (d) WKc Core

(e) WKa Incast (f) WKc Incast

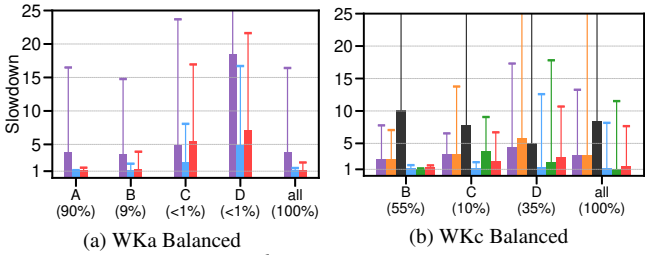
Figure 7: Median and 99^{th} percentile slowdown at 50% offered application load. Each bar group contains the messages in the following size ranges: $0 \leq A < MSS \leq B < 1 \times BDP \leq C < 8 \times BDP \leq D$. Also shown is the percentage of messages that belong to each group. WKc has no sub-MSS messages. Protocols that cannot deliver 50% load are not shown.

though queuing is much lower as it does not overcommit.

Last, the incast configuration (bottom row) illustrates the relative advantage of RD schemes over DCTCP and Swift. Homa's results would be different if the incast optimization described in [56] was implemented and the workload consisted of request-response pairs instead of one-way messages.

6.2.3 Message Latency

Figure 7 shows the median and 99^{th} percentile slowdown for different message size ranges across configurations at 50% applied load (except WKb results which fall between the other two and can be found in appendix Figure 12). Across each workload as a whole (rightmost bar cluster "all"), SIRD is generally on par with Homa, and generally outperforms dcPIM in terms of tail latency. For small, latency-sensitive messages $< BDP$ (100KB - groups A and B), SIRD, dcPIM, and Homa offer close to hardware latency. Both in aggregate and for small messages, DCTCP and Swift perform an order of magnitude worse at the tail because they cause meaningful buffering without having a bypass mechanism like Homa.



(a) WKa Balanced (b) WKc Balanced

Figure 8: Median and 99^{th} percentile slowdown at 70% offered application load. Protocols that cannot deliver 70% load are not shown. Legend same as in Figure 7.

For messages larger than $1 \times BDP$ (groups C and D), SIRD comes closest to near-optimal Homa and strongly outperforms the other protocols. SIRD achieves up to $4\times$ lower latency than dcPIM in this size range because it does not wait for multi-RTT handshakes before sending a message as discussed in §3. Note that Figure 7b is comparable to Figure 3d in [16]. Similarly, SIRD outperforms ExpressPass on latency because it does not take multiple RTTs to capture the full link bandwidth, and because it implements SRPT.

The differences in latency for protocols other than dcPIM and ExpressPass, which wait before ramping up transmission, can be explained through their *effective* scheduling policies. That is, assuming that a protocol manages to deliver enough messages to achieve 50% goodput, lowering latency is a matter of appropriately ordering the transmission of messages (assuming no conflicting application concerns). For example, DCTCP achieves equivalent or better latency than Swift in all but the incast scenarios. We argue that this is because Swift is better at fairly sharing bandwidth between messages thanks to its faster-converging control loop. Latency-wise, fair sharing is inferior to SRPT since it delays individual messages in favor of equitable progress. Along the same lines, SIRD cannot always match the latency of Homa because it approximates SRPT less faithfully: (1) unlike Homa, a portion of the sender uplink is fair-shared (50% in this case). (2) To avoid credit accumulation in congested senders, informed overcommitment adjusts per-sender credit bucket sizes equitably. This mostly impacts the latency of group C, as the bucket size may be small due to sender congestion and not grow during the lifetime of a message smaller than $8 \times BDP$. Consequently, compared to near-optimal Homa in the Balanced configuration, SIRD's 99^{th} percentile slowdown in group C is $1.85\times$ and $2.68\times$ higher at 50% and 70% application load respectively, while it outperforms the other protocols.

Figure 8 shows slowdown under a higher applied application load of 70% for the protocols that can deliver it. At this load level, message scheduling becomes more important and Homa's near-optimal SRPT implementation sees its relative performance improve in some cases.

In summary, SIRD delivers messages with low latency, outperforms dcPIM, ExpressPass, DCTCP, and Swift, and is competitive with Homa, which nearly optimally approximates

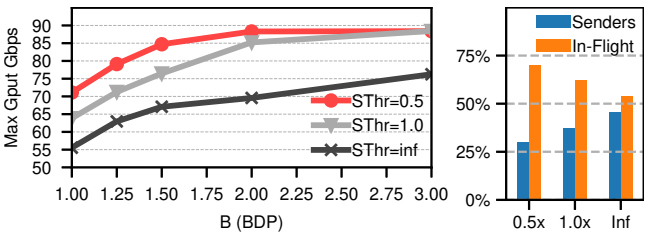


Figure 9: Left: Max goodput across values of B and $SThr$. Right: Credit location at max goodput as a function of $SThr$. $B = 1.5 \times BDP$. "In-flight" is CREDIT and DATA. Credit at receivers is low at this load.

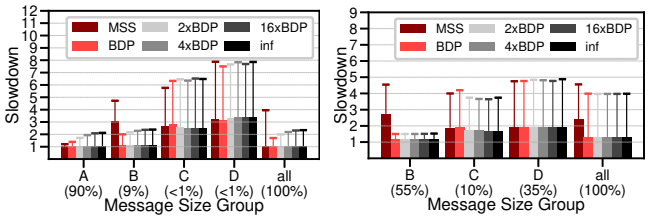


Figure 10: Slowdown as a function of UnschT for WKa (left) and WKc (right) at 50% application load in the balanced configuration;

SRPT [56].

6.2.4 Sensitivity Analysis

In this section, we explore SIRD's sensitivity to its key parameters: B, $SThr$, and $UnschT$, as well as to the availability of switch priority queues. SIRD's sensitivity to $NThr$ is the same as DCTCP's to K [8].

Informed overcommitment: Equation 2 introduced the linear steady-state relationship between B and $SThr$, the two key parameters of informed overcommitment. Figure 9 (left) shows the measured maximum achieved goodput in Balanced WKc as a function of B and $SThr$.

We observe that the presence of informed overcommitment increases achievable goodput by ∼25%, confirming that the introduced sender-informed mechanism is necessary to fully utilize the network with a limited amount of credit, and thus low queuing. When the mechanism is disabled ($SThr = \text{inf}$), there is stranded and unused credit at congested senders, which prevents receivers from achieving the full rate. Figure 9 (right) shows where in the topology credit is under max goodput. Lower $SThr$ values reduce the credit accumulation at senders, and improves throughput by increasing the number of in-flight CREDIT and DATA packets.

With the mechanism enabled, the curves asymptotically converge to the same maximum goodput of 90Gbps. Reaching the plateau with a lower value of B also demands lowering $SThr$ to reduce credit stranded at senders (Equation 2). Queuing increases with B as in Figure 2 and remains stable when varying $SThr$. We selected $B = 1.5 \times BDP$ rather than $B = 2.0 \times BDP$ for our experiments because it halves the maximum buffering caused by scheduled packets from $1\times$ to $0.5 \times BDP$ ($= B - BDP$). We do not configure $SThr$ lower than $0.5 \times BDP$ as we deem it unrealistic to implement in software, as it may cause unwanted marking due to batch credit arrivals. The minimum correct value for this topology is $0.15 \times BDP$, as per the guidelines [8].

Unscheduled transmissions: SIRD allows the unscheduled transmission of the BDP prefix of messages $\leq UnschT$. Figure 10 explores the sensitivity to this parameter using workloads WKa and WKc. Setting $UnschT = MSS$ meaningfully increases median and tail latency for messages with sizes in $[MSS, BDP]$ while higher values offer no appreciable net benefit. This also holds at 70% load. Increasing $UnschT >> BDP$ in WKc has little impact on max and mean ToR buffering, which range within [696,747]KB and [363,378]KB respectively. In contrast, because 99% of messages in WKa are unscheduled, max and mean ToR buffering scale from 501KB to 1016KB and 220KB to 435KB, respectively, as UnschT increases. It is worth noting that saturating the fabric's bandwidth, which is when queuing is maximized, is far less likely with high-request-rate workloads like WKa.

The default value of $UnschT = 1 \times BDP$ is a satisfying compromise as large values do not yield latency benefits yet can unnecessarily expose the fabric to coordinated traffic bursts (*e.g.,* incast). We confirm this by running WKc under the incast configuration which has a high concentration of $5 \times BDP$ message bursts. We observe the following performance degradation when comparing $UnschT = 4 \times BDP$ to $UnschT = 16 \times BDP$: Overall 99^{th} percentile slowdown increases by 34% while maximum and mean ToR queuing increases by 5.7× and 80% respectively. These results justify our decision to introduce a size threshold above which messages are entirely scheduled.

In deployments with severe incast consisting of messages smaller than BDP, UnschT can be configured to be smaller than MSS. If, for example, 100 hosts concurrently send a $0.5 \times BDP$ message to a single receiver, the latter will effectively receive 100 small packets requesting credit. The receiver will immediately start issuing credit based on its policy and ramp-up to full bandwidth utilization one RTT after the senders transmitted. In TCP terms, this specific scenario is roughly equivalent to each sender starting with a window of one MSS, and ramping to the correct window value just one RTT later.

Use of switch priority queues: SIRD may use a second 802.1p priority level for control packets and/or unscheduled DATA. Figure 11 shows how the use of priorities impacts message slowdown for WKa and WKc in terms of (i) possible degradation when switch priority queues are unavailable, and (ii) the benefit of prioritizing unscheduled DATA.

We observe that, across message sizes, median slowdown is largely unaffected and tail slowdown benefits from high priority transmission in some cases. Small messages benefit because they bypass the small queues SIRD forms ($\approx 1 \times BDP$ per link at the 99^{th} percentile and $\approx 0.1 \times BDP$ on average).

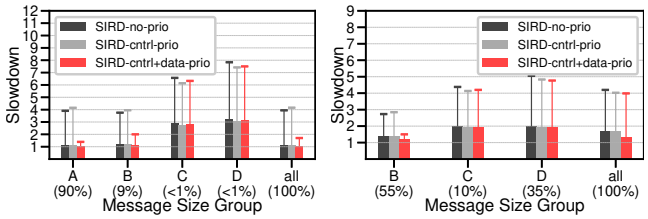


Figure 11: Slowdown as a function of priority use for WKa (left) and WKc (right) at 50% application load in the balanced configuration;

This is more evident in WKa where 99% of messages are unscheduled. Sensitivity to priorities is similar at 70% load (not shown), where, compared to 50% load, group A's tail slowdown benefits 5% more from `DATA` prioritization and group B's 6% less.

Maximum goodput increases by 1% and 2.4% with control packet prioritization for WKa and WKc respectively, as credit is delivered slightly more predictably. Queuing in the network is also insensitive to the use of priorities. Given SIRD's low sensitivity to the availability of switch priority queues, we argue that it can be deployed without this dependency with little sacrifice. This differentiates SIRD from Homa, which is designed around the use of priorities to bypass long in-network queues, and dcPIM, which strongly benefits from tight control packet delivery due to its semi-synchronous nature.

7 Related Work

The topic of datacenter congestion control has been extensively explored due to the increasing IO speeds, the µs-scale latencies, emerging programmable hardware, RPC workloads, and host-centric concerns [4–6, 8–10, 12, 13, 19, 20, 22, 31, 34, 36, 38, 42–48, 50, 51, 54, 56, 57, 61, 66–71, 74].

SD schemes like DCQCN [74], Timely [54], HPCC [47], Bolt [12], and PTT [67] use network signals in the form of ECN [8, 68, 74], delay [44, 54], in-band telemetry [47, 69, 73], and packet drops [10] to adjust sending windows [8, 44, 47, 66] or rates [38, 54, 70]. Some operate end-to-end with no or trivial switch support [8, 44, 54, 67] while others leverage novel but non-universal switch features to improve feedback quality [47, 69, 73] or to tighten reaction times [12, 19, 71].

RD protocols attempt to avoid congestion altogether by proactively scheduling packet transmissions [5, 16, 20, 31, 36, 39, 46, 49, 56, 61] and have already been discussed (§2). Orthogonally to the scope of this paper, FlexPass [49] enables fair bandwidth sharing among proactive and reactive protocols. EQDS [59] enables the coexistence of traditional protocols like DCTCP and RDMA by moving all queuing interactions to end-hosts and using an RD approach to admit packets into the network fabric.

Previous work has followed different approaches regarding the desired flow/message scheduling policy. The traditional approach is to aim for fair bandwidth allocation at the flow level [8, 44, 54, 74]. Alternatively, D^2TCP, D^3, and Karuna [18] prioritize flows based on deadlines communicated by higher-level services. A recent line of work [10, 13, 31, 39, 56], which includes both proactive and reactive protocols, attempts to approximate SRPT or SJF (shorted job first) scheduling in the fabric which minimizes average latency [10, 14]. SIRD can approximate a variety of policies at receivers and senders and avoids the need for fabric policies as it keeps in-network queuing minimal.

8 Conclusion

SIRD takes a holistic approach to datacenter congestion management by tackling the fundamental tension of receiver-driven designs, which is the management of shared links. SIRD applies proactive scheduling to exclusive receiver downlinks, which are statistically the most congested, and leverages sender and fabric congestion feedback to reactively allocate the bandwidth of shared links. Compared to existing designs, SIRD manages to simultaneously deliver high link utilization with hardly any network buffering, and low message latency. SIRD does so without assuming advanced switch ASIC features nor switch priority queues, which makes it compatible with existing and heterogeneous datacenter networks. We implemented a prototype of SIRD in software, on the Caladan stack, and showed that it can efficiently allocate credit while delivering 100Gbps.

Acknowledgements

The authors thank John Ousterhout for numerous discussions on the topic of receiver-driven protocols. We thank our shepherd Srikanth Sundaresan, Nate Foster, Katerina Argyraki, Haitham Al Hassanieh, James Larus, Charly Castes, Neelu Kalani, Boris Pismenny, Rui Yang, Mahyar Emami, Sahand Kashani, Mia Primorac, Francois Costa, and all anonymous reviewers for their insightful comments.

This work was funded in part by the Microsoft-EPFL Joint Research Center and used CloudLab [27] for 100GbE experiments. Ryan Kosta participated in EPFL's Excellence Research Internship Program.

References

[1] SIRD implementation on Caladan. https://github.com/epfl-dcsl/SIRD-Caladan-Impl, 2025.

[2] SIRD simulator repository. https://github.com/epfl-dcsl/SIRD-Simulator, 2025.

[3] Sepehr Abdous, Erfan Sharafzadeh, and Soudeh Ghorbani. Burst-tolerant datacenter networks with Vertigo. In *Proceedings of the 2021 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1–15, 2021.

[4] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–70, 2022.

[5] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David B. Shmoys, and Amin Vahdat. Harmony: A Congestion-free Datacenter Architecture. In *Proceedings of the 21st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 329–343, 2024.

[6] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 275–287, 2023.

[7] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 503–514, 2014.

[8] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[9] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, 2012.

[10] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 435–446, 2013.

[11] Eitan Altman and Tania Jimenez. NS simulator for beginners. *Synthesis Lectures on Communication Networks*, 5(1):1–184, 2012.

[12] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt: Sub-RTT Congestion Control for Ultra-Low Latency. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 219–236, 2023.

[13] Wei Bai, Kai Chen, Hao Wang, Li Chen, Dongsu Han, and Chen Tian. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 455–468, 2015.

[14] Amotz Bar-Noy, Magnús M. Halldórsson, Guy Kortsarz, Ravit Salman, and Hadas Shachnai. Sum Multicoloring of Graphs. *J. Algorithms*, 37(2):422–450, 2000.

[15] Qizhe Cai. dcpim simulator repository. https://github.com/Terabit-Ethernet/dcPIM/tree/master/simulator, 2023.

[16] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcPIM: near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 53–65, 2022.

[17] Chih-Hao Chang, VS Chang, KH Pan, KT Lai, JH Lu, JA Ng, CY Chen, BF Wu, CJ Lin, CS Liang, et al. Critical process features enabling aggressive contacted gate pitch scaling for 3nm cmos technology and beyond. In *2022 International Electron Devices Meeting (IEDM)*, pages 27–1. IEEE, 2022.

[18] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 174–187, 2016.

[19] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Center. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 17–28, 2014.

[20] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 239–252, 2017.

[21] Inho Cho, Keon Jang, and Dongsu Han. Expresspass simulator repository. https://github.com/kaist-ina/ns2-xpass, 2018.

[22] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for µs-scale RPCs with Breakwater. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 299–314, 2020.

[23] Ultra Ethernet Consortium. Overview of and Motivation for the Forthcoming Ultra Ethernet Consortium Specification. https://ultraethernet.org/wp-content/uploads/sites/20/2023/10/23.07.12-UEC-1.0-Overview-FINAL-WITH-LOGO.pdf, 2023.

[24] David Schor. IEDM 2022: Did We Just Witness The Death Of SRAM? https://fuse.wikichip.org/news/7343/iedm-2022-did-we-just-witness-the-death-of-sram/, 2022.

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

[26] Advait Abhay Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *Proceedings of the 2013 IEEE Conference on Computer Communications (INFOCOM)*, pages 2130–2138, 2013.

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

[28] Bob Felderman. Personal communication to the authors of [56], 2018.

[29] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 281–297, 2020.

[30] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 249–264, 2016.

[31] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1:1–1:12, 2015.

[32] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network Aggregation for Shared Machine Learning Clusters. In *Proceedings of the 4th Conference on Machine Learning and Systems (MLSys)*, 2021.

[33] Ehab Ghabashneh, Yimeng Zhao, Cristian Lumezanu, Neil Spring, Srikanth Sundaresan, and Sanjay Rao. A microscopic view of bursts, buffer contention, and loss in data centers. In *Proceedings of the 22nd ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 567–580, 2022.

[34] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure Flow Control. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 779–805, 2022.

[35] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 51–62, 2009.

[36] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 29–42, 2017.

[37] Torsten Hoefler, Duncan Roweth, Keith D. Underwood, Robert Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Moray McLaren, Abdul Kabbani, and Steve Scott. Data Center Ethernet and Remote Direct Memory Access: Issues at Hyperscale. *Computer*, 56(7):67–77, 2023.

[38] Chi-Yao Hong, Matthew Caesar, and Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 127–138, 2012.

[39] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 422–434, 2020.

[40] John K. Ousterhout. Homa Wiki on Aeolus. https://homa-transport.atlassian.net/wiki/spaces/HOMA/pages/262185/A+Critique+of+Aeolus+A+Building+Block+for+Proactive+Transports+in+Datacenters, 2022.

[41] John K. Ousterhout. Homa Wiki on dcPIM. https://homa-transport.atlassian.net/wiki/spaces/HOMA/pages/1507461/A+Critique+of+dcPIM+Near-Optimal+Proactive+Datacenter+Transport, 2022.

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

[43] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[44] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 514–528, 2020.

[45] Yanfang Le, Jeongkeun Lee, Jeremias Blendin, Jiayi Chen, Georgios Nikolaidis, Rong Pan, Robert Soulé, Aditya Akella, Pedro Yebenes Segura, Arjun Singhvi, Yuliang Li, Qingkai Meng, Changhoon Kim, and Serhat Arslan. SFC: Near-Source Congestion Signaling and Flow Control. *CoRR*, abs/2305.00538, 2023.

[46] Yanfang Le, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Sujata Banerjee, Aditya Akella, and Michael M. Swift. PL2: Towards Predictable Low Latency in Rack-Scale Networks. *CoRR*, abs/2101.06537, 2021.

[47] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *Proceedings of the ACM SIGCOMM 2019 Conference*, pages 44–58, 2019.

[48] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the 2021 EuroSys Conference*, pages 33–48, 2021.

[49] Hwijoon Lim, Jaehong Kim, Inho Cho, Keon Jang, Wei Bai, and Dongsu Han. FlexPass: A Case for Flexible Credit-based Transport for Datacenter Networks. In *Proceedings of the 2023 EuroSys Conference*, pages 606–622, 2023.

[50] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 47–63, 2021.

[51] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve D. Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[52] Mellanox. Spectrum 2 datasheet. https://nvdam.widen.net/s/gbk7knpsfd/sn3000-series, 2017.

[53] Mellanox. Spectrum 3 datasheet. https://nvdam.widen.net/s/6269c25wv8/nv-spectrum-sn4000-product-brief, 2020.

[54] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 537–550, 2015.

[55] Behnam Montazeri. Homa simulator repository. https://github.com/PlatformLab/HomaSimulation, 2019.

[56] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[57] Mohammad Noormohammadpour and Cauligi S. Raghavendra. Datacenter Traffic Control: Understanding Techniques and Tradeoffs. *IEEE Commun. Surv. Tutorials*, 20(2):1492–1525, 2018.

[58] Nvidia. Spectrum 4 datasheet. https://nvdam.widen.net/s/mmvbnpk8qk/networking-ethernet-switches-sn5000-datasheet-us, 2022.

[59] Vladimir Andrei Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 761–777, 2022.

[60] Amy Ousterhout, Adam Belay, and Irene Zhang. Just In Time Delivery: Leveraging Operating Systems Knowledge for Better Datacenter Congestion Control. In *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*, 2019.

[61] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 307–318, 2014.

[62] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. *CoRR*, abs/2308.00852, 2023.

[63] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. Updated by RFCs 4301, 6040.

[64] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123–137, 2015.

[65] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 183–197, 2015.

[66] Le Sun, Hai Dong, Omar Khadeer Hussain, Farookh Khadeer Hussain, and Alex X. Liu. A framework of cloud service selection with criteria interactions. *Future Gener. Comput. Syst.*, 94:749–764, 2019.

[67] Lide Suo, Yiren Pang, Wenxin Li, Renjie Pei, Keqiu Li, Xiulong Liu, Xin He, Yitao Hu, and Guyue Liu. PPT: A Pragmatic Transport for Datacenters. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 954–969, 2024.

[68] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-aware datacenter tcp (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 115–126, 2012.

[69] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 255–274, 2023.

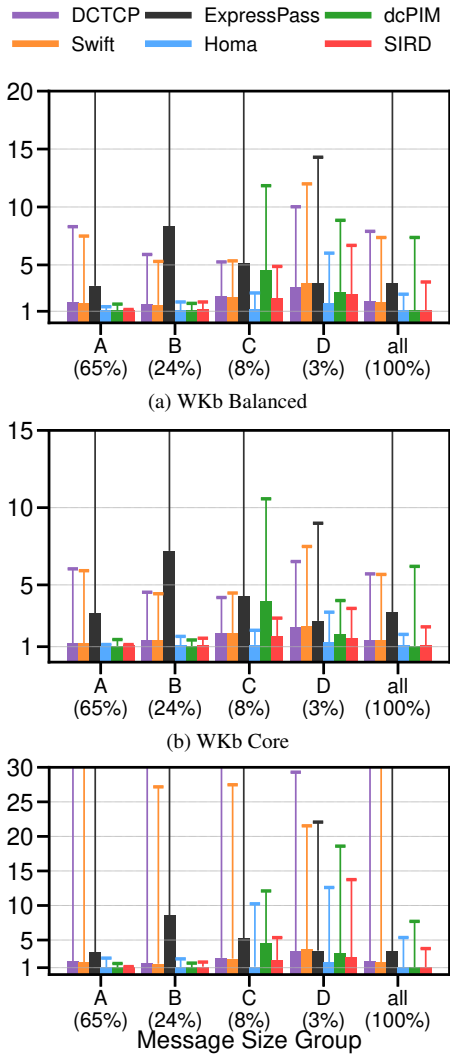
[70] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Antony I. T. Rowstron. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 50–61, 2011.

[71] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy H. Katz, Ion Stoica, and Amin Vahdat. FastLane: making short flows shorter with agile drop notification. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 84–96, 2015.

[72] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 17th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 78–85, 2017.

[73] Renjie Zhou, Dezun Dong, Shan Huang, and Yang Bai. FastTune: Timely and Precise Congestion Control in Data Center Network. In *Proceedings of the 2021 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 238–245, 2021.

[74] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 523–536, 2015.



(a) WKb Balanced

(b) WKb Core

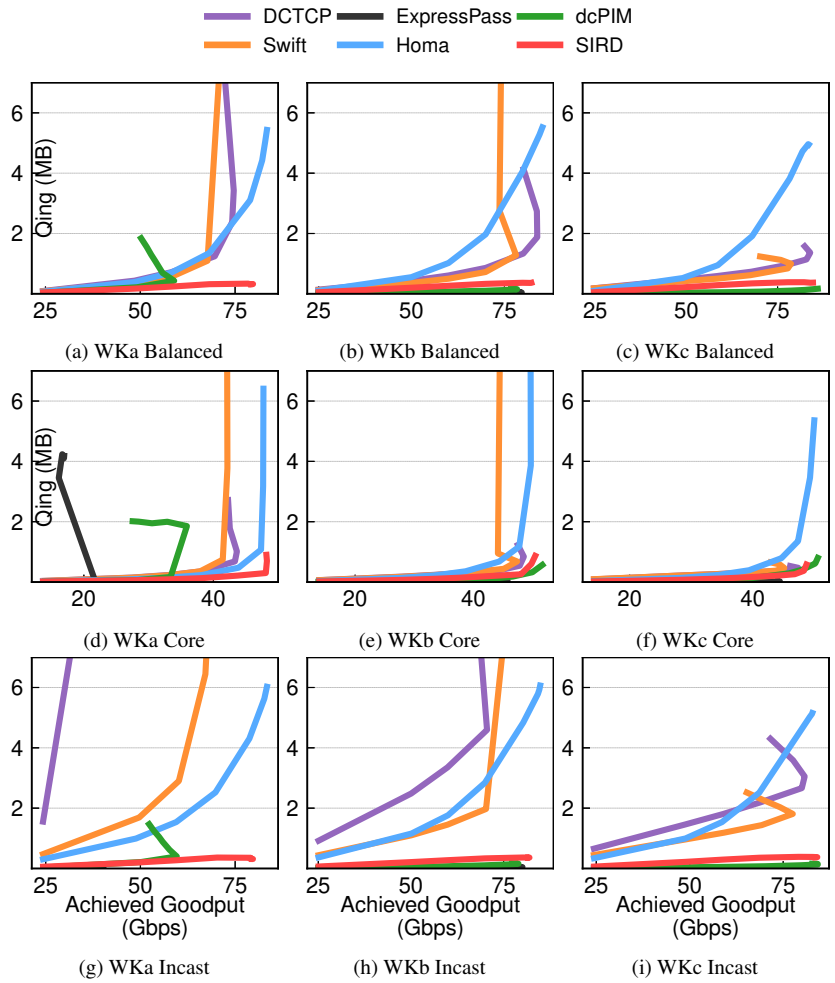
(c) WKb Incast

Figure 12: Median and 99^{th} percentile slowdown at 50% offered application load. Each bar group contains the messages in the following size ranges: $0 \leq A < MSS \leq B < 1 \times BDP \leq C < 8 \times BDP \leq D$.

A Appendix

Table 3: ASIC bisection bandwidth (in Tbps) and buffer sizes (in MB). Note that buffer architectures differ regarding the extent to which they are shared between ports.

	ASIC/Model	BW	Buffer
Broadcom	Trident+	0.64	9
	Trident2	1.28	12
	Trident2+	1.28	16
	Trident3-X4	1.7	32
	Trident3-X5	2	32
	Tomahawk	3.2	16
	Trident3-X7	3.2	32
	Tomahawk 2	6.4	42
	Tomahawk 3 BCM56983	6.4	32
	Tomahawk 3 BCM56984	6.4	64
	Tomahawk 3 BCM56982	8	64
	Tomahawk 3	12.8	64
	Trident4 BCM56880	12.8	132
	Tomahawk 4	25.6	113
nVidia	Spectrum SN2100	1.6	16
	Spectrum SN2410	2	16
	Spectrum SN2700	3.2	16
	Spectrum SN3420	2.4	42
	Spectrum SN3700	6.4	42
	Spectrum SN3700C	3.2	42
	Spectrum SN4600C	6.4	64
	Spectrum SN4410	8	64
	Spectrum SN4600	12.8	64
	Spectrum SN4700	12.8	64
	Spectrum SN5400	25.6	160
	Spectrum SN5600	51.2	160



(a) WKa Balanced (b) WKb Balanced (c) WKc Balanced

(d) WKa Core (e) WKb Core (f) WKc Core

(g) WKa Incast (h) WKb Incast (i) WKc Incast

Figure 13: Mean ToR queuing vs. achieved goodput. Configurations: Balanced (top), Core (middle), Incast (bottom).

Config	Default			Core			Incast			mean	range
Wload	WKa	WKb	WKc	WKa	WKb	WKc	WKa	WKb	WKc		
DCTCP	7.69	3.19	1.91	5.02	3.18	2.58	unstable	18.88	5.54	6.0	16.97
Swift	6.68	2.98	2.92	4.92	3.16	3.53	53.5	10.23	3.57	10.17	50.58
ExpressPass	unstable	14.94	8.09	unstable	15.9	11.37	unstable	12.61	8.2	11.85	7.81
Homa	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.42	1.27	1.08	0.42
dcPIM	1.67	2.98	1.84	1.58	3.45	2.68	1.66	2.05	1.98	2.21	1.87
SIRD	1.32	1.43	1.1	1.15	1.27	1.18	1.3	1.0	1.0	1.19	0.43
Normalized 99th percentile slowdown of all messages at 50% load.											
DCTCP	0.9	0.98	0.98	0.9	0.94	0.94	0.81	0.83	0.96	0.92	0.17
Swift	0.89	0.92	0.91	0.87	0.92	0.89	0.92	1.0	0.92	0.92	0.13
ExpressPass	0.46	0.93	0.93	0.45	0.89	0.88	0.48	0.94	0.95	0.77	0.5
Homa	1.0	1.0	0.97	0.99	0.96	0.99	1.0	0.99	0.98	0.99	0.04
dcPIM	0.71	0.92	1.0	0.74	1.0	1.0	0.71	0.92	1.0	0.89	0.29
SIRD	0.96	0.97	0.98	1.0	0.98	0.96	0.96	0.96	1.0	0.97	0.04
Normalized maximum goodput across applied load levels.											
DCTCP	unstable	85.81	9.71	3.68	62.13	7.94	unstable	unstable	260.51	71.63	256.83
Swift	unstable	unstable	8.41	unstable	unstable	10.12	unstable	unstable	125.77	48.1	117.36
ExpressPass	1.0	1.0	1.0	unstable	1.0	1.0	1.0	1.0	1.0	1.0	0.0
Homa	137.45	115.99	30.18	7.06	434.71	79.62	143.38	150.57	145.29	138.25	427.65
dcPIM	52.07	7.76	3.1	1.04	51.24	13.09	34.25	7.18	7.41	19.68	51.03
SIRD	12.05	9.91	2.68	1.0	38.9	7.53	9.37	10.03	10.27	11.3	37.9
Normalized maximum ToR queuing across applied load levels.											

Table 4: Normalized data used in Figure 5. Performance is normalized to the best performing protocol on each experiment. Experiments in which the protocol is unable to deliver the specified throughput or network queuing grows infinitely (unstable) are excluded from the calculation of mean and range.

Config	Default			Core			Incast			mean	range
Wload	WKa	WKb	WKc	WKa	WKb	WKc	WKa	WKb	WKc		
DCTCP	9.92	7.9	6.91	5.97	5.71	5.54	unstable	71.06	32.06	18.13	65.52
Swift	8.61	7.37	10.56	5.85	5.68	7.58	68.21	38.51	20.69	19.23	62.53
ExpressPass	unstable	36.94	29.26	unstable	28.56	24.42	unstable	47.45	47.45	35.68	23.03
Homa	1.29	2.47	3.62	1.19	1.8	2.15	1.27	5.36	7.37	2.95	6.18
dcPIM	2.16	7.37	6.66	1.88	6.2	5.76	2.11	7.7	11.48	5.7	9.6
SIRD	1.7	3.53	3.99	1.37	2.28	2.54	1.65	3.76	5.79	2.96	4.42
99th percentile slowdown of all messages at 50% load.											
DCTCP	74.65	83.85	83.95	43.63	48.54	47.42	67.8	70.53	80.74	66.79	40.32
Swift	74.52	78.16	78.69	42.16	47.41	45.15	76.8	85.46	77.57	67.32	43.3
ExpressPass	38.04	79.68	80.42	21.78	45.83	44.42	40.26	80.1	80.1	56.74	58.64
Homa	83.39	85.23	83.55	47.73	49.73	50.02	83.39	84.87	82.79	72.3	37.5
dcPIM	58.94	78.42	86.03	35.91	51.68	50.59	59.61	79.02	84.26	64.94	50.12
SIRD	79.74	82.27	84.71	48.27	50.47	48.75	79.7	81.98	84.13	71.11	36.44
Maximum goodput across applied load levels (Gbps).											
DCTCP	unstable	7.0	2.7	8.3	2.67	1.46	unstable	unstable	21.06	7.2	19.6
Swift	unstable	unstable	2.33	unstable	unstable	1.87	unstable	unstable	10.17	4.79	8.3
ExpressPass	0.06	0.08	0.28	unstable	0.04	0.18	0.08	0.08	0.08	0.11	0.24
Homa	8.63	9.46	8.37	15.91	18.68	14.68	12.17	12.17	11.75	12.42	10.31
dcPIM	3.27	0.63	0.86	2.34	2.2	2.41	2.91	0.58	0.6	1.76	2.69
SIRD	0.76	0.81	0.75	2.26	1.67	1.39	0.79	0.81	0.83	1.12	1.51
Maximum ToR queuing across applied load levels (MB).											

Table 5: Raw data used in Figure 5. Experiments in which the protocol is unable to deliver the specified throughput or network queuing grows infinitely (unstable) are excluded from the calculation of mean and range.

Accelerating Design Space Exploration for LLM Training Systems with Multi-experiment Parallel Simulation

Fei Gui[⋆§‡], Kaihui Gao[†], Li Chen[†], Dan Li[⋆], Vincent Liu[¶], Ran Zhang[†], Hongbing Yang[†], Dian Xiong[⋆]

[⋆]*Tsinghua University* [†]*Zhongguancun Laboratory* [¶]*University of Pennsylvania*
[§]*BNRist* [‡]*Tsinghua Shenzhen International Graduate School*

Abstract

The rapid expansion of large language models (LLMs) requires the development of extensive GPU clusters, with companies deploying clusters with tens to hundreds of thousands of GPUs. This growth significantly expands the design space for LLM training systems, requiring thorough exploration of different parallelization strategies, communication parameters, congestion control, fabric topology, *etc.* Current methods require up to 10k simulation experiments to identify optimal configurations, with inadequate exploration leading to significant degradation of training performance.

In this paper, we tackle the overlooked problem of efficiently conducting parallel simulation experiments for design space exploration. Our analysis and experiments show that Single-process Multi-experiment (SPME) achieves superior performance by reducing scheduling overhead and optimizing resource utilization, yet remains insufficient for current AI cluster scales. To enhance SPME's efficacy, we introduce Multiverse, a novel GPU-based AI training simulator. Multiverse leverages the computing throughput of GPUs efficiently with optimizations such as a pull-based synchronization, high-fidelity intra-server communication, and a kernel-fusion technique. Extensive experiments validate the accuracy and efficiency of Multiverse, demonstrating less than 3.0% discrepancy with real-world LLM training on clusters of up to 54,000 GPUs, achieving $43.1-73.2\times$ speedup over state-of-the-art CPU-based simulators in various use cases.

1 Introduction

The growth of the scale of LLM models requires large training systems [4, 27]. Companies like Tesla and Meta are building clusters with more than 24 thousand GPUs [16, 30] with O(100k) GPU clusters on the horizon. As a consequence, the design space of the LLM training becomes wider and deeper [15, 28, 54], encompassing the parallelization strategy, parameters of collective communication primitives, congestion control algorithms and parameters, fabric topology

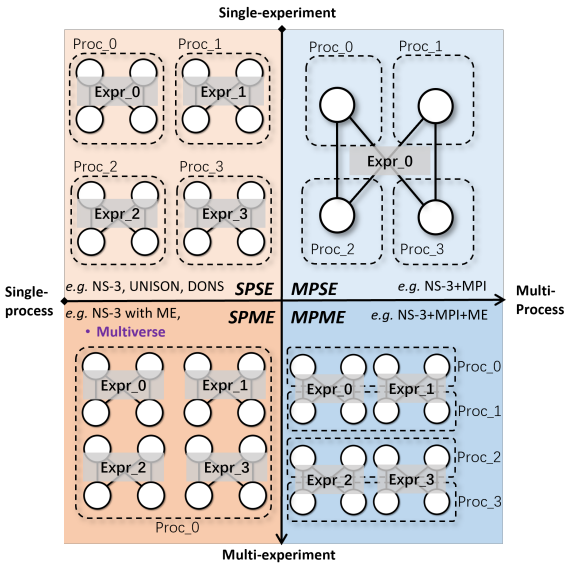


Figure 1: Four types of multi-experiment parallel strategies.

design, and others (see §2.1). As all of these options can interact in complex and unpredictable ways, discovering the optimal design point for a training system requires a massive number of simulation experiments with different option combinations, to fully search the design space. For example, exploring the optimal parallel group size requires ∼100 experiments, while identifying the optimal topology for connecting large-scale GPUs requires over 10k experiments. Insufficient exploration of the design space can lead to suboptimal operational regimes, for instance, employing a suboptimal topology could result in a $3.4\times$ longer training iteration time [58, 59].

Fortunately, we observe that the experimental runs for different points in the design space are generally embarrassingly parallel, which we can leverage to reduce the end-to-end exploration time. How do we best parallelize these experiments? The obvious answer — running n experiments on n CPU cores — falls short of current performance needs, especially when considering that competition for shared resources like memory capacity and low-level cache of CPUs mean that scaling is

sublinear. We believe that, by considering all the experiments at the same time, we can do better.

Our intuition is that batch processing a significant number of experiments aligns well with GPU architecture, which relies on a Single Instruction, Multiple Data (SIMD) execution model. We note that recent work has taken a similar approach to speed up single-experiment network simulation, using a technique called Data-Oriented Design (DOD) [18]. In a DOD-based simulator like DONS [21], users pack data, *e.g.*, packets, in a way where the simulation framework can process steps of the simulation procedure in unison (*e.g.*, running all of ingress pipelines of all switches at the same time) for parallelism and cache efficiency benefits.

We propose to extend the SIMD abstraction to multi-experiment simulation using a strategy in which users run multiple experiments *in a single simulation process*, a strategy we call SPME. Akin to the advantages of Deep Learning and DONS, the advantage of this approach is a substantial reduction in process-inherent overhead and duplicate cache/memory usage. A major additional benefit is that SPME is more amenable to deployment on actual GPUs, whose vastly superior core count and specialization for SIMD provide sizable performance benefits for this type of application.

In this paper, we present Multiverse, the first GPU-based AI training simulator. Multiverse is a DOD-based simulator, but the first to characterize an Entity-Component-System (ECS) model for AI training systems, the first to apply a SPME execution strategy, and the first to adapt all of the above to the constraints of GPU execution. For the latter in particular, Multiverse incorporates three novel techniques for GPU-based AI training simulation: a pull-based synchronization method for lock-free data transfer, a high-fidelity intra-server communication model, and a kernel-fusion technique to greatly improve GPU efficiency.

We perform extensive experiments to confirm the speed and accuracy of SPME and Multiverse. For SPME, our approach occupies a new point in the design space shown in Figure 1. On one axis, we have an existing, well-known criterion: whether to use a single process (SP)[1] or try to accelerate it with multiple processes (MP). On the other axis: running a single experiment in a single process (SE) or running multiple experiments in a single process (ME). We implement all four types of parallel strategies using state-of-the-art simulators (§2.2) and find that on both GPUs and CPUs, SPME achieves the best performance among the four (MP∗E suffers from synchronization overhead, while ∗PSE is inefficient in resource utilization).

For Multiverse as a whole, results show that it can achieve $43.1-73.2\times$ speedup over the state-of-the-art across various use cases (from 50 to 10k simulation experiments) and cluster scales (from 128 to 54k GPUs). Further, the simulation results, such as iteration time, align closely with actual LLM training in real clusters with 1,024 H100 GPUs, exhibiting a difference of less than 3.0%.

We summarize the contributions of this paper as follows:

- We identify the need for parallel experimentation in the design space exploration of large LLM training systems. We then conduct an in-depth analysis and testing of various strategies for parallelizing multiple experiments. The results indicate that the SPME approach outperforms others. We believe that the superior performance of SPME can be attributed to its reduction in process scheduling overhead and its cache-friendly nature.
- We further discover that the DOD principle and the GPU can maximize the potential of the SPME approach. The operational mode of DOD enables parallel execution of identical simulation logic across experiments and devices, such as the forwarding in switches and flow control at the hosts, closely resembling the Single Instruction, Multiple Data (SIMD) tasks that GPUs excel at. We design Multiverse, an AI training simulator, and propose three optimization techniques.
- We evaluate Multiverse on a real testbed, and Multiverse can achieve up to $73.2\times$ speedup over other methods, across various use cases and cluster scales. For simulation accuracy, the training performance metrics output by our simulator closely match the real-world results on a cluster with 1,024 H100 GPUs, showing a discrepancy of $<3.0\%$. Multiverse is open sourced at https://github.com/NASP-THU/multiverse.

2 Background and Motivation

2.1 Exploring the LLM Training Design Space

LLMs require specialized AI training infrastructure, leveraging dozens to 10s/100s of thousands of GPUs that work in concert to complete pre-training or fine-tuning tasks [15, 16, 30, 54]. Constructing large-scale infrastructure to support efficient training involves navigating a deep and broad design space, as illustrated in Figure 2. Mainstream training frameworks such as Megatron [54] and DeepSpeed [48] offer efficient implementations of various parallelization strategies, Data Parallelism (DP), Pipeline Parallelism (PP), and Tensor Parallelism (TP), to utilize large amounts of GPUs efficiently. Collective communication plays a pivotal role in facilitating result interactions among GPUs, utilizing operations like allreduce for gradient synchronization and allgather for full parameter access. The transport protocols and physical network topology underpin the operation of the aforementioned software layers, exerting a fundamental influence on the training performance of LLMs.

Navigating such a vast design space requires extensive exploratory experiments based on simulators, which serve as a crucial tool to validate system designs due to their lower

[1] SP can utilize multi-cores on a single server with multi-threading [8, 21].

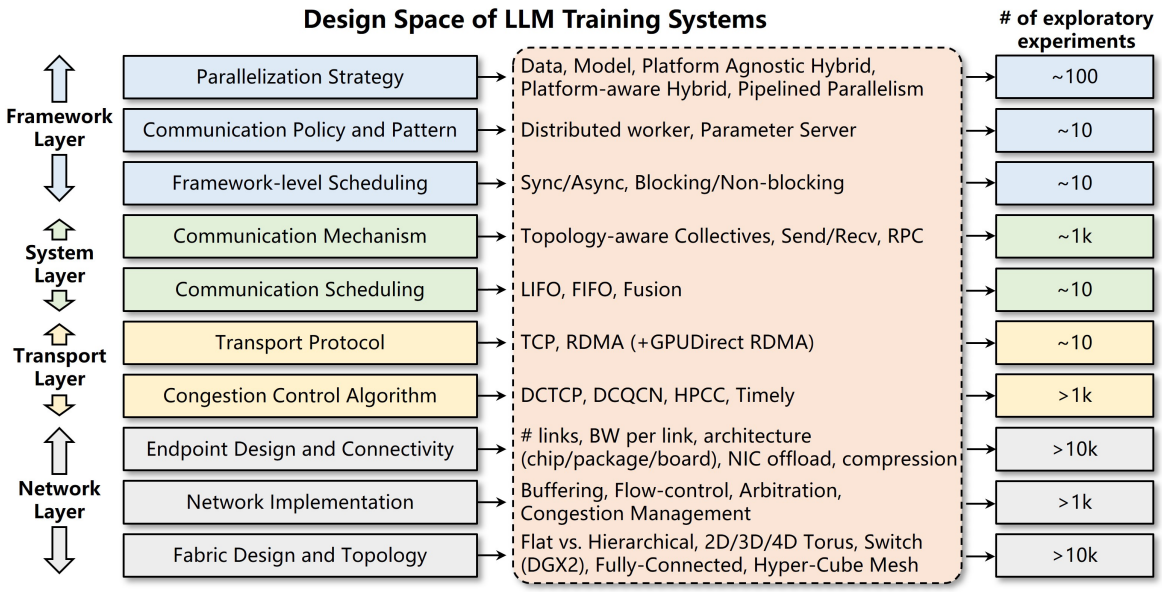


Figure 2: Design space and the number of explorations of LLM training systems.

cost compared to physical testbeds and their higher accuracy relative to analytical models [21, 34, 62]. We list two practical cases to demonstrate how simulation can be utilized to explore optimal design choices for LLM training.

Topology search: There are tons of ways to connect a large number of GPU servers, with classic topologies such as Rail-Optimized Fattree (ROFT) [43], Torus [29], BCube [22], Fat-tree [5], and Dragonfly [31]. However, these well-known topologies may fall short when faced with the evolution of model architectures and training techniques. For example, although ROFT can provide strong all-reduce performance, its all-to-all performance remains less than optimal. Researchers usually try to change parameters such as layers, port numbers, connection relationships, and more in the classic topologies, conduct >10k simulation experiments to compare them, and finally find the most suitable topology for LLM training. For instance, deploying a superior topology can yield a 3.4× increase in training speed [58, 59].

Parallel group size optimization: Allocating GPUs into TP/PP/DP group is also a crucial design decision [26, 42], because TP demands high bandwidth regions while PP needs the avoidance of bubbles. The comparison of group sizes typically requires almost one hundred simulation experiments, depending on the model size and the hardware topology [36].

Although the optimization process can be divided into two distinct phases—a long-term phase for GPU topology search and a short-term phase for other optimizations—to reduce the number of required experiments, a significant number of experiments are still needed in the short-term phase when considering parallel strategy search and collective communication algorithm search. Thus, enhancing simulation speed continues to be an essential objective.

2.2 Multi-experiment Parallelization

We taxonomize the parallelization strategy of multiple independent simulation experiments into four types along two axes: (1) using a single-process or multi-processes to execute one simulation program, and (2) running a single experiment or multiple experiments within one simulation program.

1. Single-Process Single-Experiment (SPSE): This is the most commonly used method during design space exploration, where multiple processes are initiated, each executing a single experiment. For a given experiment, NS-3 [49] and OMNeT++ [57] default to using a single process with a single thread, whereas DONS [21] and UNISON [8] utilize multi-threading (multi-core) to accelerate a single experiment.

2. Multi-Process Single-Experiment (MPSE): This approach uses multiple processes to run a single simulation experiment, like NS-3 or OMNeT++ with MPI [20].

3. Single-Process Multi-Experiment (SPME): Within one program, this strategy extends the single experiment in SPSE to multiple experiments to economize on the inherent overhead of processes.

4. Multi-Process Multi-Experiment (MPME): Similarly, this approach employs multiple processes to execute one program in SPME. For instance, NS-3 utilizes various processes to accelerate a single program, which encompasses several independent topologies running different experiments.

We implement these four parallel strategies based on state-of-the-art simulation technology, and conduct extensive ex-

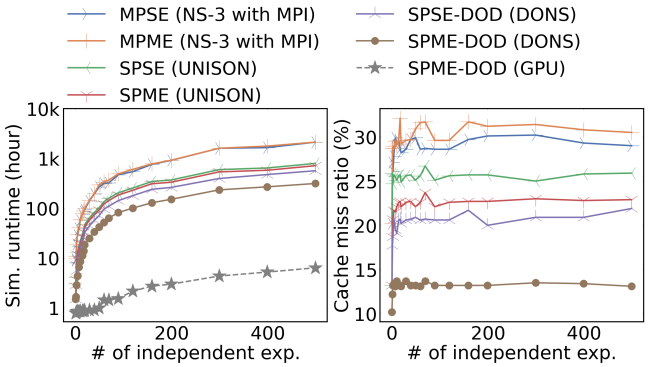


Figure 3: The performance comparison of four multi-experiment parallel strategies.

periments to verify their respective performance. The experimental scenario involves searching for the optimal collective communication algorithm for a cluster with 128 GPU servers, aimed at training a GPT-3 13B model. The simulated topology is Fattree with k=8 (Fattree8), which has 80 switches and 384 links with 100 Gbps. We test these strategies by measuring the time simulators required to complete 1 to 500 independent experiments on a machine, equipped with 80 CPU cores and one H100 GPU. The GPU cache configuration includes 1.3MB L1, 40MB L2, and 55MB L3. Experiments were implemented using C++. The experimental results are depicted in Figure 3, leading us to the following four conclusions.

1. High synchronization overhead in MP∗E: Both MPSE and MPME (the number of processes is set to the optimal value that maximizes CPU utilization) exhibited the poorest performance, necessitating 2k hours to complete 500 experiments. Process-based parallelization cannot share data without incurring context-switching overhead. Experimental results show that high synchronization overhead among multiple processes degrades simulation performance, this problem has been demonstrated in previous literature [21, 62].

2. High cache miss rate in SPSE: We use UNISON and DONS to assess the performance of SPSE. UNISON [8] introduces an efficient multi-threading parallelism technique for NS-3 and claims to achieve super-linear speedup as the number of CPU cores increases, making it the strongest variant to date. DONS [21], leveraging the Data-Oriented Design (DOD) concept, restructures the architecture of the network simulator, achieving improved performance on multi-core CPUs. In both methods, we set the optimal number of cores for each experiment (∼24 cores for Fattree8) to achieve the highest super-linear speedup ratio (∼30×). Consequently, both exhibit superior performance compared to the multi-process versions. However, under multiple experiments, this approach that naively parallelizes by running copies of the simulators will prolong the time of each experiment spent (*i.e.*, the scaling of this dimension is sublinear), due to frequent CPU context switches and severe cache misses.

3. SPME has better performance: We realize the SPME concept by configuring multiple topologies (∼16) within UNISON and DONS. The results show that the performance of both UNISON and DONS improves, benefiting from lower cache misses and process scheduling overhead across experiments. However, DONS with SPME still require significant time (∼370 hours) to complete this case; when used as an exploratory tool, this slow speed is not simply acceptable.

4. Unlocking the potential of SPME requires DOD and GPU: SPME offers a significant performance enhancement for DOD-based DONS. DOD disentangles logic from data, allowing logic to concurrently manipulate the same type of data across all entities. This enables SPME+DOD to identify cross-experiment batching and parallelization opportunities that existing methods ignore. Moreover, this pattern aligns seamlessly with the Single Instruction Multiple Data (SIMD) characteristics of GPUs. The extensive cores within GPUs can further accelerate parallelism across multiple experiments. As a comparison and a preview, Figure 3 illustrates the performance of SPME+DOD+GPU, which surpasses the best available solutions on CPUs by more than 70 times.

2.3 Realizing and Accelerating SPME+DOD for AI Training Systems

With the above observations, we believe that SPME+DOD represents a viable path towards realizing high-performance exploration tools. DOD can improve the execution efficiency of the SPME parallel strategy by separating simulation logic from data as mentioned earlier. Thus, this paper introduces Multiverse, a simulator that realizes SPME+DOD for simulating AI training systems, centered around three key ideas.

1. ECS modeling to enable SPME+DOD for AI training systems. Multiverse uses the Entity-Component-System (ECS) architecture [3, 17] to realize the DOD principle. ECS decomposes objects into separate logic (named *system*) and data (named *component*), and it is currently the default way to realize DOD. We choose ECS because it is easy to develop and prior work, namely DONS [21], has proven its performance benefits. We describe the ECS-based modeling for AI training systems in §3.2.

2. Realizing the SPME+DOD simulation in GPU. SPME+DOD allows multi-experiment simulators to exploit parallelism and coherence (both instruction and data) across experiments to achieve high efficiency on a high-throughput parallel processor like a GPU. To make Multiverse a practical tool for multi-experiment exploration, we delegate all aspects of the simulation to GPUs (§3.3). We centrally manage storage within the GPUs and compile all ECS logic into the GPU, which will adaptively utilize all cores to execute the same simulation logic across all experiments.

3. Optimizing the GPU-based LLM training simulator with three techniques. In the process of developing Multiverse, we see plenty of room to improve performance. Firstly,

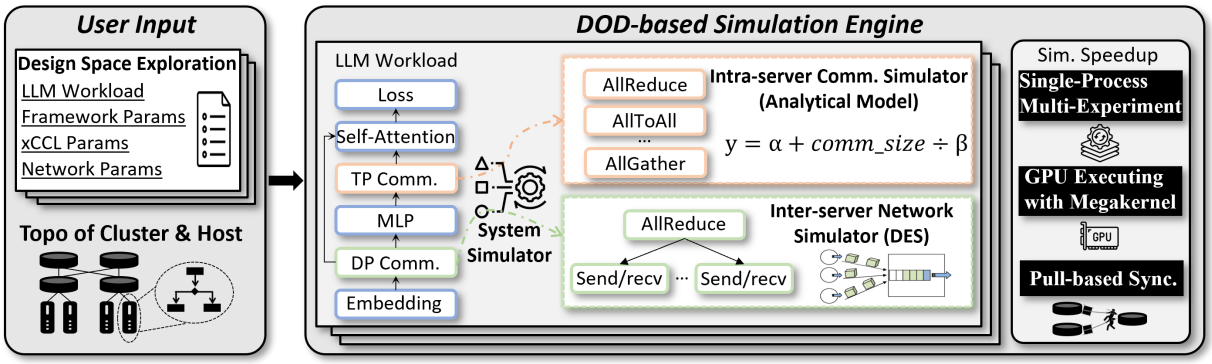


Figure 4: Multiverse's architecture.

executing network simulations with multithreading frequently led to many-to-one write conflicts. To reduce the overhead associated with locks, we implement a pull-based synchronization method. For example, one egress port actively pulls packets from the corresponding ingress ports. Secondly, the communication protocols between NVIDIA GPUs within a single server are not open source. Through extensive profiling experiments, we build an analytical model to estimate the time that GPUs take to execute various collective communication operations. Lastly, we propose a kernel-fusion technique to significantly improve GPU efficiency, compiling the simulation functions, along with wrapper code for component management, into a unified GPU megakernel [45].

3 Multiverse Design

We first overview the architecture of Multiverse and then describe its ECS-based modeling for the AI training system. Finally, we show how to map the ECS models to the GPU. We present the optimization techniques for Multiverse in §4.

3.1 Architecture

Figure 4 shows the key components in Multiverse. Users only need to write the application code, set up an AI training system, including workload, CCL parameters, topology, *etc.* , and specify multiple experiments to run; then Multiverse automatically perform multi-experiment simulations on GPUs.

System Simulator. The system simulator controls and schedules the process of AI training simulation. Its input is the model workload [2], similar to the input in ASTRA-sim [47], which characterizes the computation graph per GPU. Its node is computation operations (such as `Embedding`, `Attetion_linear`, and `MLP_linear`) or collective communication operations (such as `Allreduce`, `Allgather`). We assume that the computation operations are already annotated with the computation times, as the workload produced by Chakra [2]. Multiverse supports typical parallel strategies (*e.g.*, TP, PP, and DP).

For inter-server collective communication operations, the system simulator generates a series of point-to-point send and receive flows by implementing the collective communication algorithms (CCAs) of NCCL [23]. However, within a given CCA, the inherent overhead of the NCCL software stack influences the start time of each flow. To improve the simulation accuracy, we hijack the NCCL APIs to profile the start and end time of each communication in a CCA operation. Consequently, we calibrate the simulation of inter-server collective communications by introducing the measured overhead.

Intra-server Communication Simulator. For an intra-server collective communication operation among GPUs (*e.g.*, TP communications), Multiverse simulates it directly based on the proposed analytical model, which has different empirical parameters according to the operator type and GPU type. This model provides both fast and accurate intra-server communication simulation.

Inter-server Network Simulator. The system simulator registers many point-to-point cross-network communication demands in this network simulator, which would conduct a discrete-event simulation (DES) to rigorously enforce packet-level events and guarantee correctness, like NS-3 [49]. The system simulator is notified when the flow is completed.

GPU Memory Simulator. Since some experimental setups (*e.g.*, the parallel group sizes) may exceed the GPU memory space, Multiverse supports the simulation of GPU memory usage. Before the actual simulation, Multiverse checks whether the parallel strategy and other settings respect the memory size limits. If not, Multiverse generates an *OOM* error.

GPU Runtime. The runtime utilizes ECS abstractions to efficiently allocate multi-experiment simulation tasks to the GPU. Multiverse firstly compiles the ECS system functions, along with wrapper code for component management, into a unified GPU megakernel. Then, the execution graph of ECS systems is transferred to the GPU and processed by the megakernel in every simulation step. The system execution graph informs the runtime that identical systems are executed in each simulation step, and then the GPU adaptively uses all cores to process this system for the corresponding entities across all experiments.

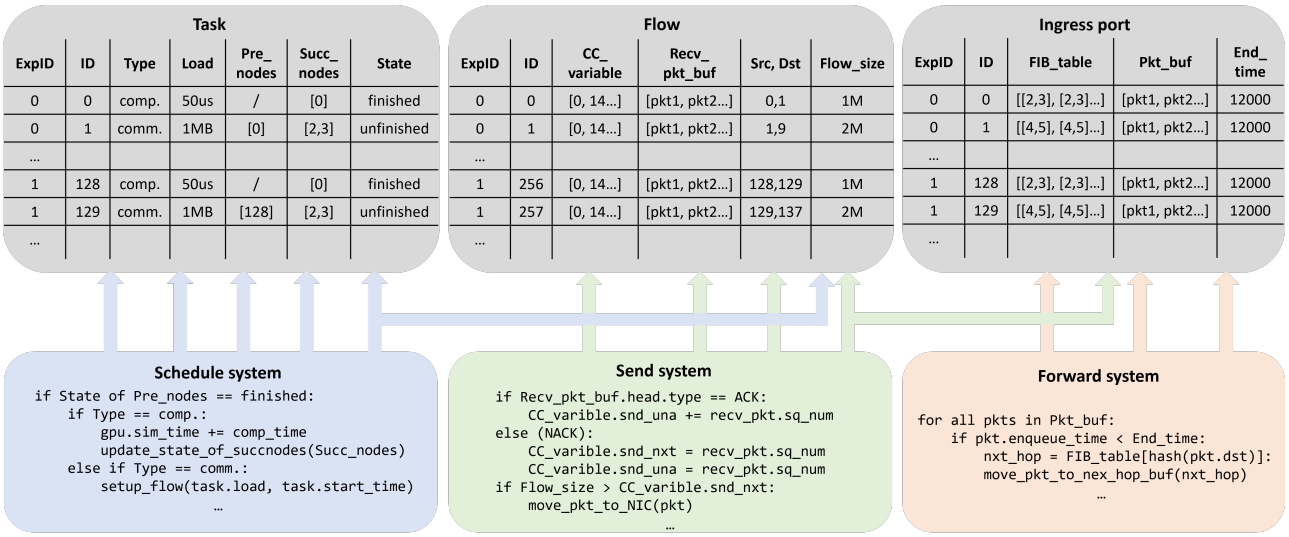


Figure 5: Archetypes and Systems (simplified) in LLM training systems.

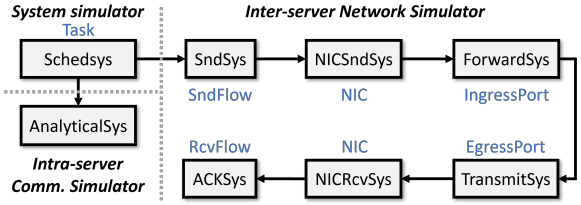


Figure 6: The system execution graph of Multiverse. The terms marked in blue refer to entities.

3.2 ECS Modeling for an AI Training System

Multiverse models the AI training process using Entity-Component-System (ECS) abstractions to realize a high-performance multi-experiment simulator.

Entities and Components. In the context of AI training, we retain the network entities in DONS, such as *Sender*, *IngressPort*, *EgressPort*, and *Receiver*, while introducing pivotal new entities like training *Task*, and *Flow*. The state of an entity is characterized by the values of its components. For instance, as shown in Figure 5, the components defining a *Task* entity encompass type (computation or communication), load (computation time or communication volume), predecessor nodes, and successor nodes. The *Flow* entity is distinguished by components that include its state, congestion control variables, and sending buffer. Entities of the same kind that have the same components are said to share an archetype. Figure 5 illustrates three archetypes of AI training system, as well as example component values for 4 entities of each archetype. Other entities and components are consistent with DONS.

Systems and Queries. Systems represent the data-parallel computations (*i.e.*, simulation logics) executed on collections of entities. A system is delineated by a query that specifies the component data included in the input and a function that is executed on these data. Queries are designed to select entities that possess a predetermined set of components.

For instance, the `Schedule` system in Figure 5 is executed per *Task*. At each step, it checks the completion status of the predecessor nodes, and activates the successor task nodes. For computation tasks, it increases the simulation clock to advance the simulation process; For inter-server communication tasks, it inputs new flows in the packet-level *Network Simulator*; For intra-server communication tasks, the `AnalyticalSys` system executes the analytical model to estimate the communication time.

System execution graph. This graph defines the entire set of ECS systems needed to execute in a simulation step, as illustrated in Figure 6. Nodes in the graph represent systems, and edges represent the execution order.

In a simulation step, Multiverse executes eight systems: `Schedule`, `AnalyticalSys`, `SendSys`, `NICSndSys`, `ForwardSys`, `TransmitSys`, `NICRcvSys`, and `ACKSys`. After Task entities execute the `Schedule` system, the intra-server communication is simulated by `AnalyticalSys`, then new flows (new *SndFlow* entities are created) are injected into the network simulator, where *SndFlow* entities execute the `Send` system to send packets to the corresponding destinations. Then packets traverse *NIC* and a forwarding path comprised of consecutive *IngressPort* entities and *EgressPort* entities. Finally, the packets arrive at the *RcvFlow* entity, which sends back an acknowledgment (ACK) if needed. After the round, Multiverse starts the next simulation step until the simulation ends. As proved in DONS, this mode of execution ensures the simulation correctness if the length of one simulation step is set reasonably (§5).

3.3 Multi-experiment Simulation in GPUs

The system execution graph enables the runtime to recognize that all experiments will utilize the same systems during sim-

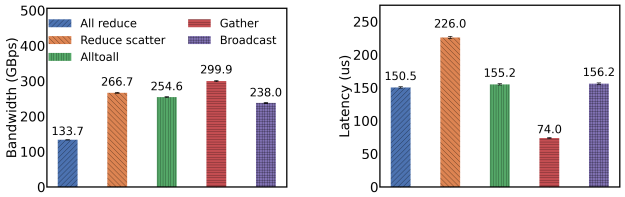
ulations. This understanding allows for safe and efficient parallel execution. By having clear knowledge of execution and data types, Multiverse gains significant control over how simulation logic is assigned to GPUs, leading to high-performance simulations. Multiverse compiles all ECS systems along with the necessary wrapper code for component access into a single GPU megakernel [45]. Then, this megakernel executes the system execution graph for each step of the simulation. Notably, archetypes and their components are set up as column stores in GPU memory.

Management of Component Storage across Experiments. In multi-experiment simulations, Multiverse centrally oversees the storage of all component data. It constructs a single in-memory table that holds component data for entities sharing the same archetype across all experiments, as illustrated in Figure 5. To facilitate efficient access to consecutive components, these tables are organized as column stores, with component data stored in contiguous memory, adhering to standard ECS practices [3]. To handle the per-experiment state when processing entities, Multiverse includes an implicit `ExpID` component in each table, which allows for quick retrieval of experiment-specific data for each entity.

This cross-experiment storage approach improves the cache miss rate of Multiverse. During execution, neighboring GPU threads that process consecutive entities in a table can access component data coherently, even if those entities belong to different experiments. If separate tables were maintained for each experiment, data access would become incoherent when only a few entities per experiment matched an ECS query, as neighboring threads would access data from different tables.

Executing ECS Systems in GPUs. At a higher level of abstraction, Multiverse uses the NVIDIA CUDA C++ compiler to compile these systems for execution on the GPU. Each system invocation is mapped directly to a single GPU thread, utilizing the single instruction multiple threads (SIMT) programming model. To facilitate the parallelization of entity updates and data flow across the execution graph, ECS systems are structured as functions that take the components of individual entities as inputs. When an ECS system is integrated into the execution graph, Multiverse uses the provided ECS query to identify all matching archetypes and their corresponding column indices for each accessed component. This information enables Multiverse to generate the necessary code for managing component access and to invoke the ECS system function across multiple GPU threads for each entity that matches the criteria.

For example, in Figure 5, the `Send` system requires data from columns 3 and 4 in the `Flow` table. Using this information, Multiverse creates a system entry function that links GPU threads to table row indices and supplies the retrieved component data to the ECS system function. Given a table with N rows, Multiverse would call the system entry function N times, with each call resulting in a single GPU thread executing the ECS system function. In the case of the `Send` system, if the query identifies N rows in the `Flow` table, Multiverse would invoke the entry function N times, providing the first N invocations with column pointers for the `Flow` table.



(a) Bandwidth (β). (b) Latency (α).

Figure 7: Calibrated bandwidth and latency parameters in the analytical model.

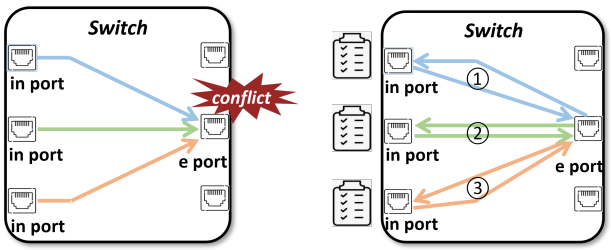
4 Multiverse Optimization

4.1 Intra-server Communication Simulation

In LLM training systems, TP traffic among intra-server GPUs constitutes a significant proportion of the aggregate network load, rendering the precise simulation of TP traffic necessary to accurately simulate LLM training. TP traffic predominantly transfers through high-bandwidth, intra-server conduits such as NVLink and PCIe. Existing simulation methodologies, however, grapple with accurately modeling TP traffic. Dedicated packet-level simulators (*e.g.*, NS-3 [49], DONS [21]) for inter-server networks face challenges in accurately simulating intra-server communication due to the fundamentally different communication protocols involved. Existing AI training simulators employ analytical models (*e.g.*, ASTRA-sim [47]) to estimate this communication: $y{=}\alpha{+}comm_size/\beta$, where α and β represent the communication latency and bandwidth, respectively. And *comm_size* denotes the total communication volume for a GPU during the collective communication operation. The advantage is the speed of simulation. However, a significant drawback is the inaccuracy of the model parameters, as they fail to capture the overhead introduced by the NCCL software stack.

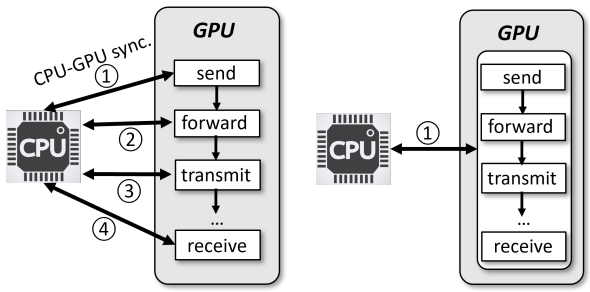
Through extensive empirical testing, we discover that the parameters within analytical models cannot remain static. Deploying a server equipped with 8 A100 GPUs interconnected via 300GB/s NVLink, we use the analytical model from ASTRA-sim to estimate the completion time of TP yields an error margin ranging from 20% to 72%. We identify that the issue does not lie with the linear model itself, but rather with the need for its parameters to vary according to different scenarios. To encapsulate this pattern, we conduct tests to examine the relationship between the communication size and the completion time of various collective communication operators. By fitting a linear model to the results, we are able to derive revised parameters.

As shown in Figure 7, the analytical model we proposed



(a) Multiple-to-one write conflict. (b) Pull-based sync.

Figure 8: Resolute the multiple-to-one write conflict with pull-based synchronization.



(a) Kernel-based. (b) Megakernel-based.

Figure 9: The execution of the ECS systems in GPU.

meticulously adjusts specific parameters (α and β) for each collective communication operation, contingent upon the number and type of GPU (*e.g.*, A100/H100) within a server. The bandwidth values deviated from officially documented metrics, and the latency can be 70*us* to 200*us*. This calibrated approach facilitates the accurate simulation of intra-server communication; evaluation results reveal that the error is merely $0.7\%-1\%$. Crucially, this modeling approach enables a significant enhancement in speed without compromising accuracy. This attribute is beneficial for augmenting the compute throughput of ECS architectures within GPUs, affording a substantial uplift in the simulation performance.

Simulating the overlap of communication and computation. Furthermore, the computation time can be significantly influenced by the degree of overlap with communication due to resource contention, such as streaming multiprocessors (SM) and high-bandwidth memory (HBM) bandwidth. To address that, Multiverse adjusts the computation time based on the overlap between computation and communication. Initially, Multiverse specifies the duration of each computation operator without overlap in the input file. During the simulation, this duration is modified according to the overlap ratio between computation and communication operators and the extent to which the computation duration is extended due to this overlap. To determine the overlap ratio, the duration of the communication operator is obtained using either a scale-out package-level network simulator or a scale-up analytical model. By comparing the durations of the communication and computation operators, the overlap ratio is calculated. Additionally, an empirical modeling approach is used to evaluate the extent to which computation time is extended when the computation operator overlaps with the communication operator. This model considers various factors, including the model type, the computational operator, and the GPU type.

4.2 Pull-based Data Synchronization

In Multiverse, the `nic_forward` and `forward` systems extensively perform multiple-to-one write operations, a scenario where current ECS frameworks (like Unity ECS [56]) exhibit significant limitations due to the lack of atomic operations. As a workaround, these frameworks are compelled to accumulate all send events in a global buffer, subsequently processing them sequentially in the main thread. This methodology severely compromises parallel efficiency. The key challenge is managing concurrent writes without resorting to the conventional lock-based synchronization techniques.

We adopt a pull-based data synchronization strategy to enhance parallel efficiency. This approach bifurcates the multi-to-one writing process within the system into two distinct phases: `set_write_plan` and `write`. In the `set_write_plan` phase, packets slated for dispatch are preliminarily listed in a to-do queue. Then, in the `write` phase, each destination actively retrieves the packets intended for it from the source. For example, we divide the `forward` system into two subsystems: `set_forward_plan`, and the `forward`. In the former, a to-do list (realized as a bitmap) is maintained for each ingress port within the current switch, earmarking impending forward events. Subsequently, in the `forward`, each egress port independently pulls the packet from the ingress ports based on its to-do list one by one, as illustrated in Figure 8b. This method allows all ports to execute the `forward` system in a lock-free manner, improving parallel efficiency.

4.3 Execution with Megakernel

A crucial aspect of Multiverse setup is the execution of the entire system execution graph. The straightforward method would be to run each system as an individual CUDA kernel. However, this strategy has a significant problem. In GPU, initiating a kernel is a time-consuming process because it needs to be initialized by the CPU. As shown in Figure 9a, Multiverse consists of numerous ECS systems, leading to a high amount of CPU-GPU synchronization overhead, which notably decreases GPU efficiency.

As shown in Figure 9b, Multiverse uses a GPU-driven method with a large-scale kernel design [45]. Multiverse compiles all systems in the execution graph into a single CUDA kernel, *i.e.*, megakernel. This kernel is initiated by the CPU once per batch simulation and completes the entire execution graph before returning to the CPU. To manage both the execution of systems defined by the application and necessary engine-level operations, such as evaluating ECS queries and sorting tables, Multiverse creates a task graph. This graph

outlines the full execution of each batch simulation step. Multiverse uses a simple task graph scheduling policy in which all GPU threads work on the same node together before moving to the next node in the task graph. While this approach could lead to low GPU utilization if there is not enough work per node, large-batch simulations of multi-experiment typically provide a significant amount of work per node. So we can easily fully utilize all the GPU cores with a megakernel.

5 Implementation

We base Multiverse on the Madrona framework [52]. Multiverse 's current core code base has $\sim 13k$ lines of C++ code, which we release for the networking community. Currently, Multiverse supports both TP (Tensor Parallelism) and DP (Data Parallelism) parallelization strategies, along with collective communication algorithms including Ring Allreduce, Allgather, and Reducescatter. Additionally, Multiverse integrates multiple congestion control algorithms such as DCQCN, HPCC, and DCTCP, as well as load balancing algorithms like ECMP and packet spraying. Here we briefly describe the length of a step in Multiverse, and how it use multiple GPUs.

The length of one simulation step. Multiverse traverses the execution graph in one simulation step, advancing the simulation clock incrementally. Given that the execution of distinct systems within the execution graph is sequential, prolonged step length can lead to temporal errors. This step length effectively corresponds to the notion of *lookahead* concept in parallel DES [20]. Taking inspiration from DONS [21], we set $min\{link_delay\}$ as the lookahead. Because event propagation between servers and switches inherently involves at least one link delay.

Using multiple GPUs. Given the finite resources within a single GPU, evaluation reveals that the number of experiments that one GPU can simultaneously run is limited, when each experiment is of substantial scale. Multiverse can leverage multiple GPUs for parallel exploration experiments. Upon user configuration of available GPUs, Multiverse will assess the necessity of reallocating certain independent experiments across different GPUs. If necessary, Multiverse will automatically balance the workload among multiple GPUs to maximize simulation speed. The execution across GPUs are independent.

6 Evaluation

We evaluate Multiverse's simulation speed, accuracy, and scalability. We summarize our results as follows:

Key Results.

- **Simulation speed:** Multiverse can achieve a 57.4-73.2× speedup over the state-of-the-art across various use cases, ranging from 50 to 10,000 simulation experiments, and cluster sizes, from 128 to 8,192 GPUs.
- **Scalability:** Using only one GPU, Multiverse can run one simulation experiment for a cluster of up to 54,000 GPUs and achieve up to 43.1× speedup over other methods. And Multiverse can parallel run 52 simulation experiments for a cluster of 1,024 GPUs and 13 simulation experiments for a cluster of 8,192 GPUs.
- **Fidelity:** Multiverse's simulation results, such as iteration time, align closely with real-world LLM training on testbeds, with a difference of $<3.0\%$, even at 1,024 GPUs.

Alternatives. We select existing state-of-the-art simulators for AI training systems as our comparisons.

- **ASTRA-sim+UNISON:** ASTRA-sim [47, 60] is a comprehensive AI training simulator. To enhance its simulation speed, we improve the network simulator using UNISON [8], a recently proposed multi-threading simulation kernel. This approach can utilize the SPSE and SPME manners, denoted as *ASTRA-sim+UNISON (SPSE)* and *ASTRA-sim+UNISON*, respectively.
- **ASTRA-sim+DONS:** We incorporate ASTRA-sim with another state-of-the-art network simulator, DONS, to implement an accelerated ASTRA-sim. This method can utilize the SPSE and SPME manners, denoted as *ASTRA-sim+DONS (SPSE)* and *ASTRA-sim+DONS*, respectively.
- **Multiverse (SPSE)**: This approach takes Multiverse to follow the SPSE manner. And it employs all the optimization technologies we proposed.

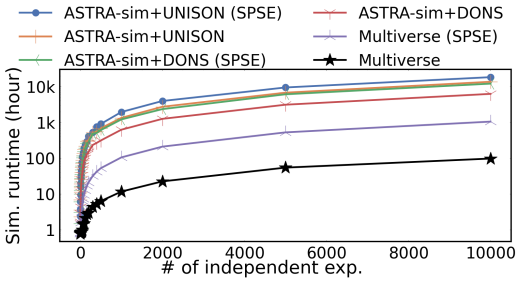
Setup. We conduct experiments with these alternatives on a Linux server configured with one NVIDIA H100 GPU, an 80-core Intel CPU@2.10GHz, and 256GB of memory.

To verify Multiverse's accuracy, we run real LLM training jobs (with LLaMA 65B and GPT-3 175B model) on a cluster, which is based on the Fattree, inter-host RoCEv2 network. The cluster consists of 128 servers, each equipped with eight NVIDIA H100 GPUs and eight Mellanox ConnectX-7 NICs (2×200Gbps). Intra-server GPUs are interconnected via NVLink, providing 900GBps bandwidth. The clusters include two kinds of scale: 128 GPUs and 1,024 GPUs.

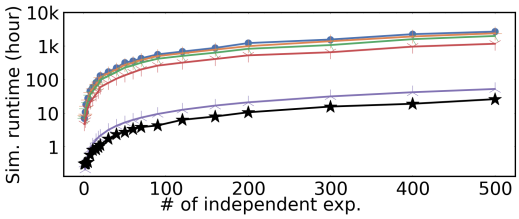
We take four real use cases of design space exploration to test the performance of Multiverse, as shown in Table 1. These use cases span a comprehensive spectrum, including topology search, collective communication enhancement, fine-tuning of parallel group sizes, and the assessment of congestion control algorithms. These cases cover a wide range of cluster scales (ranging from 128 to 54,000 GPUs) and model sizes (from 13 billion to 175 billion parameters). We obtain the approximate number of experiments required for these four explorations from the production datacenters. For instance, topology search needs varying layers, switch radixes, and connectivity configurations to generate a myriad of topologies, requiring 10k simulation experiments.

Use case	GPU scale	Topology	Workload	Explored parameters	# of independent exp.
#1: Topology optimization	128 GPUs	Fattree-like	GPT-3 13B	Set different layers, switch radix, connections, *etc.*	10k
#2: Collective communication optimization	1,024 GPUs	Fattree k=16	LLaMA 65B	Set different flow priority and load balancing strategies.	500
#3: Selection of TP/DP/PP group size	8,192 GPUs	Fattree k=32	GPT-3 175B	Set different TP/DP/PP group size.	100
#4: Comparing congestion control algorithms	54,000 GPUs	Fattree k=60	GPT-dense 175B	Set different CC algorithm, such as DCQCN [63], HPCC [35].	4

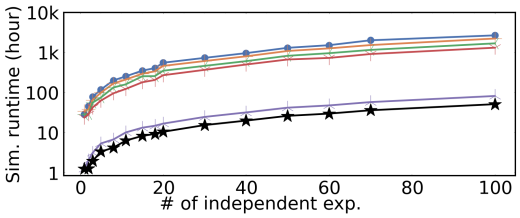
Table 1: The detailed setting for the four different use cases in evaluation.



(a) The total spent time of multiple simulation experiments for a cluster with 128 GPUs (use case #1).



(b) The total spent time of multiple simulation experiments for a cluster with 1,024 GPUs (use case #2).



(c) The total spent time of multiple simulation experiments for a cluster with 8,192 GPUs (use case #3).

Figure 10: The comparison of simulation speed.

6.1 Multiverse is Fast

We first compare the simulation speed of Multiverse, with other techniques across 3 use cases in Table 1. As depicted in Figure 10, Multiverse consistently outperforms all other methods, achieving speed increases of up to 73.2×, 67.6×, and 57.4× compared to other alternatives when simulating a cluster with 128 GPUs, 1,024 GPUs, and 8,192 GPUs, respectively. Furthermore, Multiverse surpasses Multiverse (SPSE), with speed improvements reaching up to 7.3×, 2.4×, and 1.7×, respectively. These speed enhancements can be attributed to the reduced cache miss ratio, as shown in Figure 11, a result of the DOD paradigm and batch and parallel simulation across multiple experiments. Additionally, Multiverse benefits from

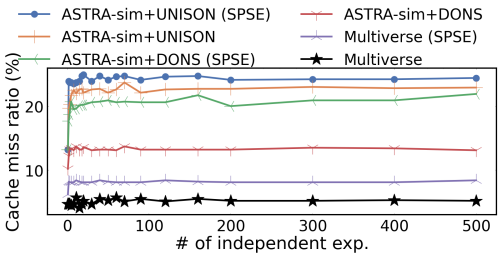


Figure 11: Cache miss ratio in use case #2.

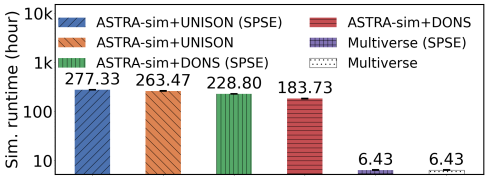


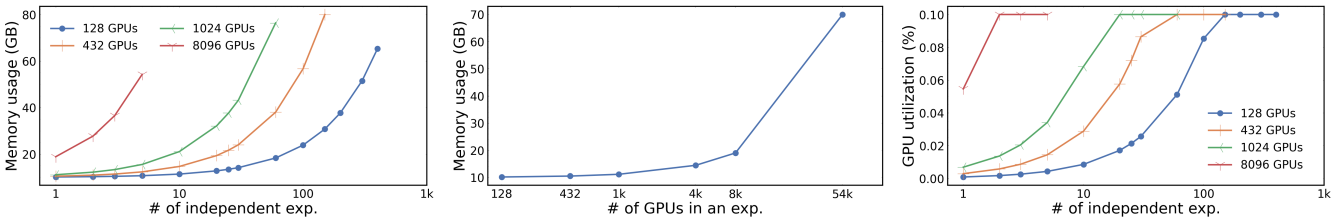
Figure 12: The speed comparison in simulating a large-scale cluster with 54k GPUs (use case #4).

reduced memory overhead as it distributes the inherent memory cost in all experiments. Together, these factors improve the parallel efficiency of Multiverse.

ASTRA-sim+DONS performs better than ASTRA-sim+UNISON, achieving a speedup of approximately 1.9−2.1×, as ECS programming abstractions are well suited for creating novel multi-experiment simulators. These ECS abstractions also provide essential structure over custom logic and state, enabling simulators to efficiently manage memory, distribute work, and identify coherent parallel computations within and across different experiments.

Multiverse also outperforms ASTRA-sim+DONS, with speed improvements of up to 25.1−47.2×. This is because ASTRA-sim+DONS, despite using batch simulation across multiple experiments to achieve good performance against ASTRA-sim+DONS (SPSE), has its parallelism constrained by the number of CPU cores. In contrast, Multiverse can leverage the extensive number of GPU cores to further accelerate the simulation. Moreover, Multiverse uses pull-based synchronization for lock-free data transfer, a fast analytical model, and a megakernel technique to enhance GPU efficiency.

Maximum scale. We test the maximum cluster that Multiverse can simulate using a single GPU. Subject to the constraints imposed by the GPU memory size in our testbed, Multiverse can simulate an LLM training cluster that includes up to 54k GPUs, 4.5k switches, and 162k links. This scale surpasses



(a) GPU memory usage. (b) GPU memory usage running 1 exp. (c) GPU utilization.

Figure 13: The performance of Multiverse.

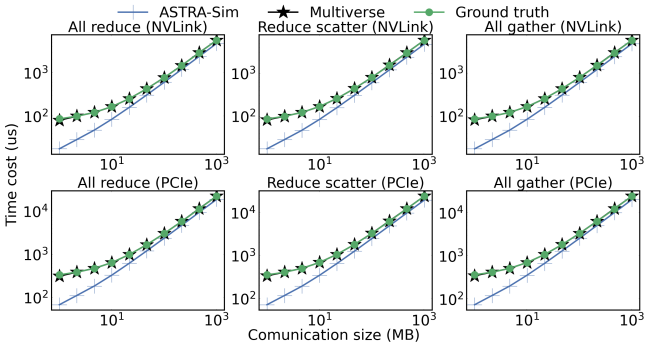
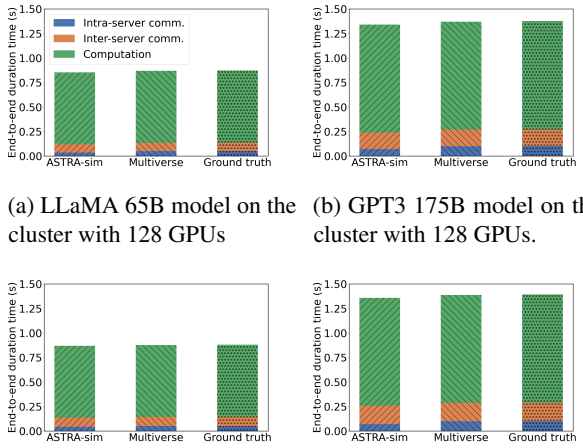


Figure 14: The time cost of intra-server collective communication operations.



(a) LLaMA 65B model on the cluster with 128 GPUs (b) GPT3 175B model on the cluster with 128 GPUs.

(c) LLaMA 65B model on the cluster with 1,024 GPUs (d) GPT3 175B model on the cluster with 1,024 GPUs.

Figure 15: The iteration time of different workloads running on ASTRA-sim, Multiverse, and real clusters.

that of most LLM training systems. Even under these conditions, Multiverse can significantly outperform alternative methods, achieving a speedup factor between 28.6× to 43.1×, as depicted in Figure 12.

The memory consumption and GPU utilization of Multiverse are illustrated in Figure 13. With the use of a single GPU, Multiverse can execute one simulation for a cluster with up to 54k GPUs at a time, this being limited by the GPU memory size, as demonstrated in Figure 13b. Furthermore, Multiverse can concurrently run 520 simulation experiments for a cluster of 128 GPUs, 70 simulation experiments for a cluster of 1,024 GPUs, and 5 simulation experiments for a cluster of 8,192 GPUs.

In terms of GPU utilization, Multiverse can fully engage all GPU cores when running a single simulation experiment for a cluster with more than 8,192 GPUs. Multiverse can also fully utilize all the GPU cores when running more than 30 simulation experiments for a cluster with 128 GPUs, and more than 10 simulation experiments for a cluster with 1,024 GPUs.

6.2 Multiverse is Accurate

The precision of intra-server communication simulation. As depicted in Figure 14, we evaluate the accuracy of Multiverse in simulating different sizes (2-2560MB) of collective communication on a server with 8 A100 GPUs(300GBps NVLink and PCIe5.0). For smaller communication sizes, ASTRA-sim can deviate by as much as 72.1% from the actual values. We test the accuracy of both NVLink and PCIe connected, respectively. Although this error decreases with increasing communication size, it remains above 22.0%. In contrast, Multiverse shows an error of 1.0-1.2% for small communication sizes, but this discrepancy reduces to less than 0.8% as the communication size increases.

From an end-to-end simulation perspective, the accuracy of intra-server communication for both ASTRA-sim and Multiverse improves with larger communication sizes, as seen in Figure 15. This is particularly noticeable with the collective communication operations of GPT-3 175B, which are larger and thus more accurately simulated. However, ASTRA-sim can still exhibit an error greater than 20.0% even when simulating GPT-3 175B, while Multiverse reduces the error to less than 1.0% for both LLaMA 65B and GPT-3 175B.

The precision of inter-server network simulation. As seen in Figure 15, due to the precision of DES, both using Multiverse and ASTRA-sim, the inter-server communication time closely approximates the real-world LLM training.

The precision of end-to-end simulation. We test the accuracy of Multiverse by examining its end-to-end metric across various workload and cluster sizes. Figure 15 demonstrates that the iteration time for all workloads when using Multiverse closely matches that of real LLM training tasks. The difference is less than 3.0%, even at the scale of 1,024 GPUs. Moreover, as the model size increases (for instance, from

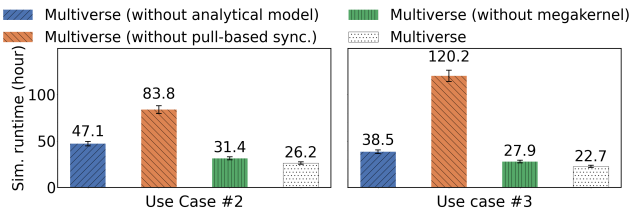


Figure 16: Performance breakdown of Multiverse.

LLaMA 65B to GPT-3 175B), Multiverse's accuracy improves correspondingly, because the message sizes of the collective communication operations in GPT-3 175B are larger.

6.3 Ablation Study of Multiverse

In Figure 16, we perform an ablation study to assess the impact of key features of Multiverse on its overall performance.

The term Multiverse (without analytical model) refers to the method that employs a packet-level DES to simulate intra-server communications, where all intra-server GPUs are interconnected via a switch. Compared to this method, Multiverse can achieve a speedup of 1.7-1.8×, because the calibrated analytical model has lighter computational loads.

Multiverse (without pull-based synchronization) uses the global synchronization method typically utilized in existing ECS frameworks. Compared to this approach, Multiverse can achieve a speedup of 3.2-5.4×. This is because pull-based synchronization allows all ports to execute the forward system in a lock-free parallel manner, significantly enhancing parallelism efficiency.

Lastly, Multiverse (without megakernel) is a method by which each ECS system is initialized as an independent kernel function in the GPU. Compared to this method, Multiverse can reduce the simulation time by 16.6%-18.6%, as shown in Figure 16. This is because using megakernel can substantially decrease the number of times the CPU needs to launch a kernel, resulting in lower CPU-GPU synchronization overhead.

7 Related Work

AI training simulation. ASTRA-sim [47, 60] is a comprehensive AI training simulator, which aims to simulate the software and hardware co-design stack of distributed training systems. Building upon ASTRA-sim, SimAI [7] enhances simulation accuracy, Dally [53] additionally simulates modern networking hardware. Approaches like [1, 9, 19] adopt a more coarse-grained simulation paradigm, such as operator-level or task-level, to achieve faster simulation speeds with the sacrifice of accuracy. SimBricks [34] and SplitSim [33] achieve end-to-end network system simulation by integrating multiple simulators, such as NS-2/3 [24, 49] and gem5 [6, 12, 41]. While they can be extended to simulate AI/LLM training, their performance is constrained. Notably, they fail to fully leverage the fine-grained parallelism opportunities provided by SPME.

Network simulation. Existing network simulators can be divided into three categories: discrete-event simulation (DES) [24, 38, 49, 57], mathematical model estimation [32, 39, 46] and AI-based estimation [50, 61, 62]. DES simulators offer high-fidelity packet-level simulation. Some work leverage parallel and distributed DES (PDES) [20, 25] approaches to accelerate simulation. However, multiple processes often come with poor performance and complex configurations. UNISON [8] introduces fine-grained partitioning and load-adaptive scheduling to enable efficient multi-threading. DONS [21] uses the data-oriented design to reduce cache miss and can be automatically parallelized.

Entity-Component-System (ECS). The ECS architecture has gained recent popularity through implementations in major game engines such as Unity [3] and Unreal [17], as well as a wide array of open-source implementations of the architecture [14, 40, 52]. The ECS architecture has been shown to provide performance benefits due to data access efficiency and parallelism [13, 56].

GPU-based DES. Many efforts employ GPUs to accelerate DES of queuing networks [44, 51, 55, 62], P2P networks [37], and mobile networks [10, 11]. However, they are still rooted in conventional PDES techniques, lacking the utilization of ECS architecture to optimize performance.

8 Conclusion

This paper demonstrate that the parallel strategy of Single-process Multi-experiment (SPME) achieves superior performance by reducing scheduling overhead and optimizing resource utilization, yet remains insufficient for current AI cluster scales. To enhance SPME's efficacy, we introduce Multiverse, a novel GPU-based AI training simulator. Using DOD/ECS modeling, Multiverse leverages the computing throughput of GPUs efficiently with optimizations such as a pull-based synchronization method, high-fidelity intra-server communication model, and a megakernel technique. Extensive experiments validate Multiverse 's accuracy and efficiency, achieving $43.1-73.2\times$ speedup over state-of-the-art CPU-based AI training simulators.

9 Acknowledgement

We thank our shepherd, Dr. Hang Zhu and the anonymous NSDI reviewers for their constructive comments. Kaihui Gao and Li Chen are the corresponding authors. This work was supported by National Key R&D Program of China (Grant No. 2022YFB3105000), the Beijing Outstanding Young Scientist Program (Grant No. JWZQ20240101008), and the National Natural Science Foundation of China(Grant No. U23B2001). It was also supported by Zhongguancun Laboratory.

References

[1] Online; Last accessed Sep. Arcadia: An end-to-end AI system performance simulator. https://engineering.fb.com/2023/09/07/data\-infrastructure/arcadia-end-to-end-ai-system\-performance-simulator/, 2024.

[2] Online; Last accessed Sep. Chakra Working Group. https://mlcommons.org/working-groups/research/chakra/, 2024.

[3] Online; Last accessed Sep. Entity Component System concepts. https://docs.unity3d.com/Packages/com.unity.entities@1.3/manual/concepts-intro.html, 2024.

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM CCR*, 2008.

[6] Mohammad Alian, Daehoon Kim, and Nam Sung Kim. pd-gem5: Simulation infrastructure for parallel/distributed computer systems. *IEEE Computer Architecture Letters*, 15(1):41–44, 2015.

[7] Anonymous authors. Simai: Unifying architecture design and performance tunning for large-scale large language model training with scalability and precision. In *USENIX NSDI*, 2025.

[8] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. Unison: A parallel-efficient and user-transparent network simulation kernel. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 115–131, 2024.

[9] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. vtrain: A simulation framework for evaluating cost-effective and compute-optimal large language model training. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 153–167. IEEE, 2024.

[10] Ben Romdhanne Bilel and Nikaein Navid. Cunetsim: A gpu based simulation testbed for large scale mobile networks. In *2012 International Conference on Communications and Information Technology (ICCIT)*, pages 374–378. IEEE, 2012.

[11] Ben Romdhanne Bilel, Nikaein Navid, and Mohamed Said Mosli Bouksiaa. Hybrid cpu-gpu distributed framework for large scale mobile networks simulation. In *2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, pages 44–53. IEEE, 2012.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[13] Blizzard. Overwatch2. https://overwatch.blizzard.com/, 2023.

[14] Michele Caini. Ecs back and forth (part 1). https://skypjack.github.io/2019-02-14-ecs-baf-part-1/.

[15] DeepSeek-AI and et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[16] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[17] Unreal Engine. Mass entity. https://docs.unrealengine.com/5.0/en-US/mass-entity-in-unreal-engine/, 2023.

[18] Richard Fabian. Data-oriented design. *framework*, 21:1–7, 2018.

[19] Yicheng Feng, Yuetao Chen, Kaiwen Chen, Jingzong Li, Tianyuan Wu, Peng Cheng, Chuan Wu, Wei Wang, Tsung-Yi Ho, and Hong Xu. Echo: Simulating distributed training at scale. *arXiv preprint arXiv:2412.12487*, 2024.

[20] Richard M Fujimoto. *Parallel and distributed simulation systems*, volume 300. Citeseer, 2000.

[21] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *ACM SIGCOMM*, pages 167–181, 2023.

[22] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 63–74, 2009.

[23] NVIDIA Inc. Nccl. https://developer.nvidia.com/nccl. (Accessed on 09/14/2024).

[24] Teerawat Issariyakul, Ekram Hossain, Teerawat Issariyakul, and Ekram Hossain. *Introduction to network simulator 2 (NS2)*. Springer, 2009.

[25] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, 30:54–73, 2013.

[26] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[27] Peng Jiang, Christian Sonne, Wangliang Li, Fengqi You, and Siming You. Preventing the immense increase in the life-cycle energy and carbon footprints of llm-powered intelligent chatbots. *Engineering*, 2024.

[28] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[29] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–14, 2023.

[30] Tech Journalist. xai colossus supercomputer comes online with 100k nvidia gpus. https://www.techopedia.com/news/elon-musks-xai-colossus-supercomputer-comes\-online-with-100k-nvidia-gpus, 2024.

[31] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3):77–88, 2008.

[32] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[33] Hejing Li, Praneeth Balasubramanian, Marvin Meiers, Jialin Li, and Antoine Kaufmann. Splitsim: Large-scale simulations for evaluating network systems research. *arXiv preprint arXiv:2402.05312*, 2024.

[34] Hejing Li, Jialin Li, and Antoine Kaufmann. Simbricks: End-to-end network system evaluation with modular simulation. In *ACM SIGCOMM*, page 380–396, 2022.

[35] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM special interest group on data communication*, pages 44–58. 2019.

[36] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[37] Xinhu Liu and Philipp Andelfinger. Time warp on the gpu: Design and assessment. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 109–120, 2017.

[38] Zheng Lu and Hongji Yang. *Unlocking the power of OPNET modeler*. Cambridge University Press, 2012.

[39] Marco Ajmone Marsan, Michele Garetto, Paolo Giaccone, Emilio Leonardi, Enrico Schiattarella, and Alessandro Tarello. Using partial differential equations to model tcp mice and elephants in large ip networks. *IEEE/ACM Transactions on Networking*, 13(6):1289–1301, 2005.

[40] Sander Mertens. Flecs 2.0 is out. https://ajmmertens.medium.com/flecs-2-0-is-out-1255a2443fbd.

[41] Alian Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. dist-gem5: Distributed simulation of computer clusters. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 153–162. IEEE, 2017.

[42] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[43] NVIDIA. Roft. https://docs.nvidia.com/https:/docs.nvidia.com/

dgx-superpod-reference-architecture-dgx-h100.pdf.

[44] Hyungwook Park and Paul A Fishwick. An analysis of queuing network simulation using gpu-based hardware acceleration. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 21(3):1–22, 2011.

[45] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)*, 29(4):1–13, 2010.

[46] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H Low. Multipath tcp: Analysis, design, and implementation. *IEEE/ACM Transactions on networking*, 24(1):596–609, 2014.

[47] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 81–92, 2020.

[48] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[49] George F Riley and Thomas R Henderson. The ns-3 network simulator. *Modeling and tools for network simulation*, pages 15–34, 2010.

[50] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet: Leveraging graph neural networks for network modeling and optimization in sdn. *IEEE Journal on Selected Areas in Communications*, 38(10):2260–2270, 2020.

[51] Janche Sang, Che-Rung Lee, Vernon Rego, and Chung-Ta King. Experiences with implementing parallel discrete-event simulation on gpu. *The Journal of Supercomputing*, 75:4132–4149, 2019.

[52] Brennan Shacklett, Luc Guy Rosenzweig, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Transactions on Graphics (TOG)*, 42(4):1–13, 2023.

[53] Aakash Sharma, Vivek M Bhasi, Sonali Singh, George Kesidis, Mahmut T Kandemir, and Chita R Das. Gpu cluster scheduling for network-sensitive deep learning. *arXiv preprint arXiv:2401.16492*, 2024.

[54] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[55] Wenjie Tang and Yiping Yao. A gpu-based discrete event simulation kernel. *Simulation*, 89(11):1335–1354, 2013.

[56] Unity. Data-oriented technology stack (dots). https://unity.com/dots, 2023.

[57] András Varga. A practical introduction to the omnet++ simulation framework. *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem*, pages 3–51, 2019.

[58] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training llms with trillion parameters. *arXiv preprint arXiv:2307.12169*, 2023.

[59] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. {TopoOpt}: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, 2023.

[60] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 283–294, 2023.

[61] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. Deepqueuenet: towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 441–457, 2022.

[62] Qizhen Zhang, Kelvin KW Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. Mimicnet: fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 287–304, 2021.

[63] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

Optimizing RLHF Training for Large Language Models with Stage Fusion

Yinmin Zhong[1] Zili Zhang[1] Bingyang Wu[1] Shengyu Liu[1] Yukun Chen[2]
Changyi Wan[2] Hanpeng Hu[2] Lei Xia[2] Ranchen Ming[2] Yibo Zhu[2] Xin Jin[1]

[1]*School of Computer Science, Peking University* [2]*StepFun*

Abstract

We present RLHFuse, an efficient training system with *stage fusion* for Reinforcement Learning from Human Feedback (RLHF). Due to the intrinsic nature of RLHF training, *i.e.*, the data skewness in the generation stage and the pipeline bubbles in the training stage, existing RLHF systems suffer from low GPU utilization. RLHFuse breaks the traditional view of RLHF workflow as a composition of individual tasks, splitting each task into finer-grained subtasks, and performing *stage fusion* to improve GPU utilization. RLHFuse contains two key ideas. First, for generation and inference tasks, RLHFuse splits them into sample-level subtasks, enabling efficient *inter-stage fusion* to overlap the execution of generation and inference stages, thus mitigating the original generation bottleneck dominated by long-tailed samples. Second, for training tasks, RLHFuse breaks them into subtasks of micro-batches and performs *intra-stage fusion* to concurrently execute these subtasks in the training stage with a fused pipeline schedule, effectively mitigating the pipeline bubbles. The experiments show that RLHFuse increases the training throughput by up to 3.7×, compared to existing systems.

1 Introduction

The rise of Large Language Models (LLMs) marks a revolutionary leap in generative AI. Despite the impressive capabilities of LLMs, many studies [1–3] have shown that LLMs often display unintended behaviors. To address these issues, Reinforcement Learning from Human Feedback (RLHF) [4] has been introduced, aiming to align LLMs with human intent after pre-training. With a concrete and well-defined reward target, RLHF enhances the ability of LLMs in multiple domains. Figure 1 shows the typical RLHF training workflow, which involves six different tasks divided into three stages. The `Actor` model first generates samples with the input prompts and passes them to the `Reference`, `Critic`, and `Reward` models for inference, finally the loss is calculated to train both the `Actor` and `Critic` models, while the other two models remain frozen. There are barriers between stages due to the data and weight dependencies.

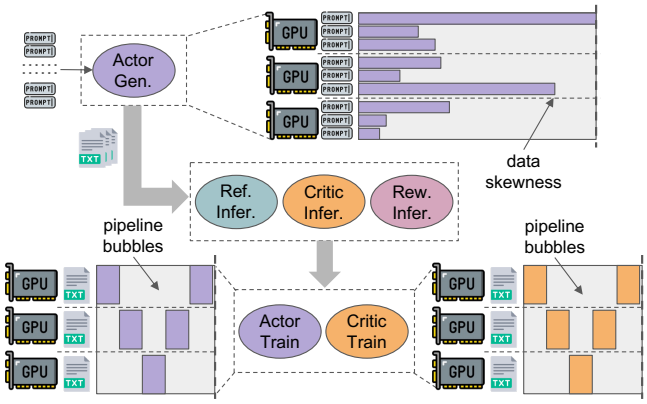


Figure 1: RLHF workflow [4] and the problems in the generation and training stages of existing RLHF training frameworks.

To enable efficient training of RLHF, existing frameworks [5–7] have proposed a series of optimizations, such as selecting tailored parallel strategies for each task and optimizing the data exchange during stage transitions. However, these techniques *fundamentally treat the RLHF workflow as a simple composition of individual tasks*, failing to delve into the inherent characteristics and structure inside the tasks, thereby missing significant optimization opportunities. In production deployments, we have observed two issues within current frameworks that severely impact training efficiency, as indicated by the grey area in Figure 1.

The first issue arises from the *data skewness* in the generation stage, where the output length of generated samples follows a long-tailed distribution. This phenomenon is broadly observed during LLM generation (§2.2) and this data diversity is critical to the robustness of RL training. On the one hand, during the latter part of the generation stage, only a small number of long-tailed samples remain processing, but a large batch size is required for generation to improve GPU utilization (§2.1). On the other hand, later tasks cannot start in advance due to data dependencies. As a result, the generation time is dominated by the longest sample and the GPU

utilization is low, as shown in the upper right of Figure 1. Although numerous optimization techniques [8–12] are proposed to improve the generation speed and throughput, they cannot alter the inherent long-tail phenomenon and thus fail to solve the problem. Making the problem even worse, as the maximum output length of LLMs continues to grow [13, 14], this long-tail phenomenon will become more significant.

The second issue stems from the *pipeline bubbles* in the training stage. With the explosive growth in the size of LLMs, higher pipeline parallelism (PP) size is needed to scale training. When LLMs reach hundreds of billions of parameters, it is quite common for PP size to reach a few dozen [15, 16]. However, the proportion of pipeline bubbles increases with PP size, significantly reducing training efficiency. In RLHF, there are two training tasks, each may evolve a specific LLM with hundreds of billions of parameters, which amplifies this inefficiency. Despite that various approaches have been proposed to reduce pipeline bubbles when training individual models [17–20], the remaining bubbles still pose a significant challenge under synchronous training semantics.

To this end, we propose RLHFuse, which tackles the *long-tailed generation issue* and *pipeline bubbles* by employing *subtask-level optimizations*. By breaking down the task into finer-grained subtasks, RLHFuse opens up new design space to *fuse the stage execution*. This is orthogonal to current RLHF frameworks [5–7] that optimizes on the task-level to choose the optimal parallel strategies for each task or reduce the task switching overhead. RLHFuse focuses on how to execute the RLHF workflow efficiently with stage fusion after the parallel strategies are chosen. RLHFuse exploits the opportunities at the subtask level and proposes two new techniques, i.e., *data-aware inter-stage fusion* and *model-aware intra-stage fusion*. These two techniques leverage the unique system characteristics of RLHF training to fuse the execution across different tasks, thus effectively addressing the data skewness and pipeline bubbles, respectively.

For data skewness, we split the generation and inference tasks into sample-level subtasks and then the dependency granularity between the generation and inference stage can be refined from stage-level to sample-level. This is due to the fact that the computation of the two stages is essentially independent for different samples. To leverage this insight, we design a lightweight and efficient migration algorithm to automatically detect when the generation stage enters the inefficient long-tail processing phase and migrate the remaining long-tailed samples to a limited number of GPUs. The freed-up resources are used to launch the inference tasks. This approach enables inter-stage fusion and dynamically adjusts the migration timing in each iteration according to the workload, thereby maximizing overall efficiency.

For pipeline bubbles, we break the training task into subtasks of micro-batches. Pipeline bubbles are essentially caused by dependencies between subtasks of the same model. Fortunately, in the RLHF training stage, there are two independent models, and their subtasks are mutually independent. Based on this observation, we employ a fused pipeline schedule to enable intra-stage fusion, which executes the training tasks on the same set of GPUs with bidirectional pipelines, effectively filling each other's pipeline bubbles. We design a lightweight schedule generation algorithm that can produce near-optimal schedules for `Actor` and `Critic` models of any size and parallel configurations.

In addition, we apply a series of system optimizations covering each RLHF stage, transforming RLHFuse into a production-ready framework to support the RLHF training of our internal products. We evaluate RLHFuse on various LLMs and real-world datasets. Compared to state-of-the-art solutions, RLHFuse improves the throughput up to 3.7×.

In summary, we make the following contributions:

- We identify the key issues within current RLHF training frameworks and propose to view the RLHF workflow from a finer-grained subtask-level perspective.
- We present RLHFuse, a highly efficient RLHF training framework that utilizes inter- and intra-stage fusion to improve the training throughput.
- We conduct a comprehensive evaluation of RLHFuse and compare it with state-of-the-art RLHF training solutions.

2 Motivation

In this section, we first introduce the basics of RLHF training (§2.1) and then point out the problems in existing RLHF training systems (§2.2).

2.1 Background

One complete RLHF process consists of three key steps: supervised fine-tuning, reward model training, and model alignment using the PPO algorithm [21]. The first two stages both train a single LLM, which has been studied by previous work like Megatron-LM [15] and MegaScale [16]. The third stage, which this paper focuses on, is unique and complex in terms that it involves multiple models and stages. Next, we provide a detailed overview of the RLHF models and workflow.

RLHF models. Typically, RLHF training involves four LLMs in total: `Actor`, `Ref`, `Critic`, and `RW`. The `Actor` model, which is the primary model to be trained and the final product after RLHF, serves as the *agent* in the RL semantics. Given a prompt, each token it generates autoregressively is considered an *action*, and the goal of the PPO algorithm is to guide it to produce actions that align with the reward target. The Reference model (`Ref`) is initialized from the original `Actor` model, but is frozen (*i.e.*, weights do not update) during training. It provides Kullback-Leibler (KL) divergence regularization to ensure the `Actor` model does not deviate excessively from its original version during training or generate nonsensical outputs. The Reward model (`RW`) is trained on human-labeled preference data to score each sample generated by the `Actor`

model, which guides the Actor model to generate responses that align with human preferences. It is also frozen during the RLHF training. The Critic model, initialized from the RW model, serves as the value model to evaluate the actions taken by the Actor. It provides finer-grained feedback on the action level to guide Actor towards better decisions.

RLHF workflow. The workflow of one RLHF training iteration contains three main stages:

Generation stage. The Actor model generates responses for the prompts in the current batch. This process involves a prefill phase and a decoding phase. The prefill phase processes the prompt to generate the first output token. Then the decoding phase sequentially generates subsequent tokens in multiple steps autoregressively (i.e., each decoding step generates a new token based on tokens generated in previous steps until reaching a termination token). Ultimately, each prompt and its corresponding response form one training sample for this iteration, referred to as a *rollout* or *trajectory* in the context of RL. To increase sample efficiency, it is common to generate many samples in one iteration.

Inference stage. The Ref, RW, and Critic models each perform a forward pass on the generated samples. It is similar to the prefill phase in the generation stage but does not generate the next token. Instead, the output logits are used to calculate the training loss in the *Training* stage. Note that the execution time of each inference task depends on the model size and is generally not the same.

Training stage. In the PPO algorithm, unlike traditional LLM training, all the samples are divided into several mini-batches and the model weights need to be updated after training on each mini-batch. Concretely, for each mini-batch, the Actor and Critic models first perform a forward pass, then calculate the loss using the results from the inference stage and performing a backward pass to update the parameters. After the Actor completes training on all mini-batches, it uses the *up-to-date* parameters for the generation in the next iteration.

LLM parallelization. Various parallelism methods are proposed to scale training, guided by the scaling law [22]. Data Parallelism (DP) replicates the model weights and distributes the data among the replicas to execute in parallel. It requires gradient synchronization across the replicas after each iteration. Tensor Parallelism (TP) assigns individual operators over multiple GPUs, with each executing part of the computation in parallel. It requires significant communication, so it is typically used within a node to leverage the high-bandwidth NVLINK connections. Pipeline Parallelism (PP) organizes LLM layers into stages, each running on a separate device or node. It partitions the input batch into multiple "micro-batches" to form pipeline execution and accumulate the gradients of the entire input batch. Each parallel method has its own advantages, so in practice all three parallel strategies are used together to scale training, such as Megatron-LM [15] and

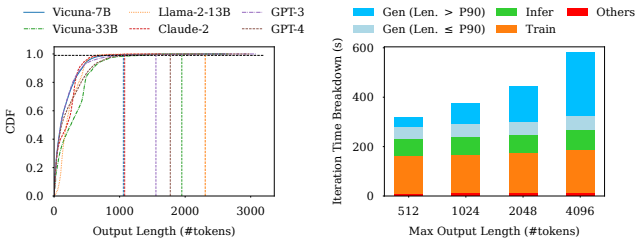


Figure 2: Left: The output length CDF of models in the LMSYS-Chat-1M dataset. The vertical dotted line indicates the P99.9 output length. Right: The RLHF training iteration breakdown on the internal model and datasets under different maximum output lengths.

MegaScale [16] in pre-training [15] as well as ReaLHF [6] and HybridFlow [5] in RLHF training.

2.2 Problems in RLHF Training

During our practical deployment of RLHF training, we identified two key issues in the *Generation* and *Training* stage within existing RLHF training systems that contribute to substantial GPU under-utilization.

Generation Stage: data skewness. One major problem in the generation stage is that the response length of generated samples exhibits a long-tailed distribution, meaning that a few samples are significantly longer than the others. Due to data dependencies, the inference tasks cannot start until the generation task finishes. Consequently, even only a few long-tailed samples force the inference tasks to wait. This leads to significantly low GPU utilization in the generation stage because the decoding phase of LLM generation is memory-bandwidth-bound, requiring a large batch size (typically a few hundred) to maintain high GPU utilization.

This long-tail phenomenon is prevalent and pronounced in the LLM generation. Figure 2 (left) shows the CDF of output length distribution in LMSYS-Chat-1M dataset [23], which collects one million user requests and corresponding responses from various models on the popular LLM evaluation platform Chatbot Arena. We observe this long-tail distribution across models of varying sizes, both open-source and proprietary. The vertical dotted lines mark the 99.9-th percentile length corresponding to each model, which is more than ten times the median length. In practice, it is common to generate over 1,000 samples in each iteration, meaning that almost every iteration will encounter long-tailed samples.

Internally, we also observe this pattern during the RLHF training of a proprietary model with hundreds of billions of parameters. Figure 2 (right) presents the iteration time breakdown under different maximum output length settings. The time spent generating long-tailed samples (dark blue bar) accounts for more than half of the total generation time. The situation gets worse as the maximum generation length increases, which leads to a substantial rise in iteration time. It is worth noting that this change only affects the long-tailed samples ($< 1\%$), but the results indicate that this very small

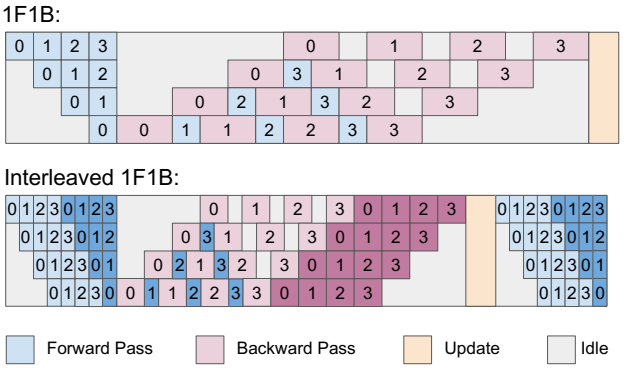


Figure 3: The timeline of 1F1B [18] and interleaved 1F1B [15] pipeline schedule with 4 pipeline stages and 4 micro-batches.

portion of long-tailed samples can have a significant impact on the overall system performance. As user applications increasingly demand lengthy outputs [13, 14, 24], the inclusion of samples with long response lengths in RLHF training becomes more common and important, making the long-tail phenomenon an urgent problem in real deployment.

Training Stage: pipeline bubbles. With the exponential growth in the size of LLMs, higher parallelism is required to scale training. While TP size is generally limited to the number of GPUs within a single node (typically 8) due to the significant communication overhead, and DP size increases the memory consumption of model weights linearly. Consequently, PP has become a critical method for scale training.

Scaling the PP size is not a free lunch, as pipeline bubbles can significantly impact training efficiency. Concretely, in the most commonly used 1F1B pipeline schedule [18] as shown in Figure 3 (upper), the *pipeline bubble percentage* is $\frac{N-1}{N-1+M}$ with N PP stages and M micro-batches. One simple way to reduce the bubble percentage is to increase M. However, the global batch size is constrained by the converging conditions and cannot be increased indefinitely. Additionally, in RLHF, the global batch is first divided into several mini-batches, then distributed among the DP groups, and finally split into micro-batches, further limiting M. As N scales with the model size to approach M, which is quite common when scaling LLMs to hundreds of billions of parameters, the bubble percentage is about 50%. This means that about half of the GPUs are idle during training, leading to a significant waste of resources.

Interleaved 1F1B scheduling [15] is proposed to mitigate the pipeline bubbles. As shown in Figure 3 (bottom), it divides the LLM into more fine-grained model chunks, with each stage hosting $K(K > 1)$ model chunks, reducing the pipeline bubble percentage to $\frac{N-1}{N-1+KM}$. However, it not only introduces a K-fold communication overhead but also exacerbates the imbalances across PP stages. As a result, K is typically set to a small constant in practice, leaving room for further optimizations.

Other works [18, 20] aim to eliminate pipeline bubbles; however, they either violate the synchronous training semantics [18] or rely on specific assumptions [20]. As a result, pipeline bubbles still pose a significant problem in LLM training, especially when the model size keeps scaling.

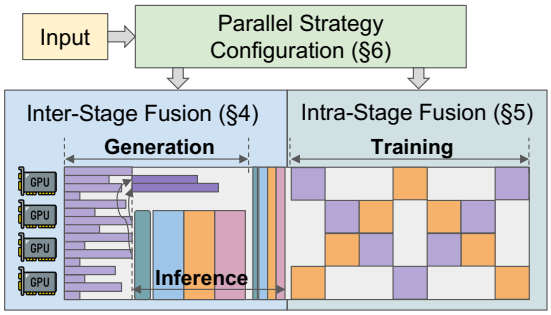


Figure 4: RLHFuse architecture.

Summary. Essentially, the above issues are caused by the inherent characteristics of RLHF, which are difficult to eliminate through task-level optimizations. Existing systems treat the task as the smallest execution unit and overlook its internal structure, thereby missing a huge optimization opportunities.

3 RLHFuse Overview

To this end, we present RLHFuse to address data skewness in the generation stage through data-aware inter-stage fusion (§4) and mitigates pipeline bubbles in the training stage with model-aware intra-stage fusion (§5). Additionally, RLHFuse adopts a series of system optimizations (§6) tailored for RLHF training. Here we provide a brief overview of RLHFuse as shown in Figure 4.

Workflow. With specific algorithm, model, and cluster configurations as input, RLHFuse first configures efficient parallel and deployment strategies following the approach in ReaLHF [6] and HybridFlow [5]. For each task within the RLHF workflow, RLHFuse assigns it a tailored parallel strategy to maximize GPU utilization. At runtime, RLHFuse launches each task with its corresponding parallel strategy on the designated device mesh following the workflow dependency and handle the weight redistribution and data transmission between different tasks to ensure synchronous training semantics. Meanwhile, RLHFuse utilizes data-aware inter-stage fusion to fuse the generation and inference tasks and model-aware intra-stage fusion to fuse the training tasks.

Inter-stage fusion. As depicted in the left part of Figure 4, during the generation stage, there are multiple model instances, each managing complete model weights and part of the samples. RLHFuse actively monitors each generation instance and triggers migration when the number of remaining samples is below a particular threshold. The remaining samples are migrated to some dedicated generation instances which are designated to handle long-tailed samples. Subsequently, the resources of the original generation instances are repurposed to launch inference tasks, thereby allowing for the

overlap of inference tasks with the generation of long-tailed samples and fusing the execution of the two stages.

Intra-stage fusion. As shown in the right part of Figure 4, in the training stage, RLHFuse leverages the insight that another inverse pipeline can complement current pipeline execution. It utilizes a lightweight algorithm to generate a fused pipeline schedule based on the model size and parallel strategies of the `Actor` and `Critic` model, which minimizes the pipeline bubbles and activation memory usage.

4 Data-aware Inter-Stage Fusion

In this section, we first analyze the opportunities of fusing the generation and inference stages (§4.1). We then introduce the fused execution plan to maximize the overlap between the two stages while preserving the data dependency (§4.2).

4.1 Opportunities and Challenges

Opportunities. The dependency between the two stages is predicated on the sample level, implying that once a sample completes its generation stage, it can seamlessly advance to the inference stage. This key observation motivates us to break the task into sample-level subtasks without violating original synchronous training semantics. Consequently, we can initiate inference tasks as soon as there exist completed samples in the generation stage.

Based on this, we design a fused execution plan that overlaps the generation and inference stages. Specifically, the fused plan first detects the point when the generation instance enters the long-tail decoding phase. At this juncture, most of the samples have finished their generation stage but are waiting for the inference stage to start. Each instance only leaves a few long-tailed samples to process, which is memory-bandwidth-bound as discussed in §2.2. This condition facilitates the migration of samples from all instances to a few designated instances. This migration not only consolidates the GPU utilization of the receiving instances but also releases resources from the sending instances. The released resources are then promptly utilized to launch inference tasks in advance. One example of a fused execution plan is shown in Figure 5. This approach greatly improves the GPU utilization and optimizes the overall execution time of the two stages.

Challenges. With the inter-stage fusion, the problem now lies in generating an efficient fused execution plan that decides the migration timing, destination, and mechanism. These three factors significantly affect the overall execution time of the two stages, and therefore must be chosen carefully.

Given that inter-stage fusion appears so promising, a natural question arises: can we break the synchronization between the inference and training stages similarly? Unfortunately, training tasks require each mini-batch to maintain the same data distribution, which necessitates random sampling from all the generated samples. To satisfy this requirement, it implicitly introduces an unavoidable synchronization boundary.

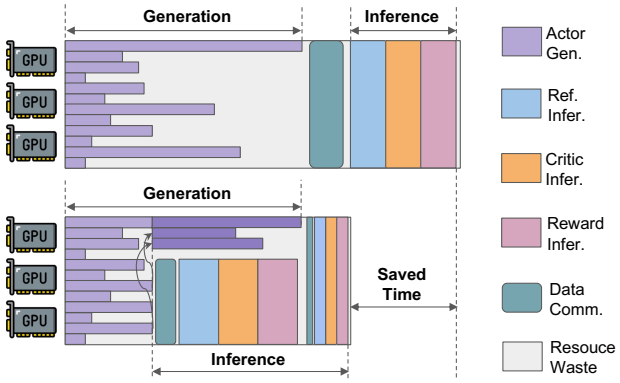


Figure 5: The timeline of serial (top) and fused (bottom) execution of generation and inference stages.

4.2 Fused Execution Plan

The objective of the inter-stage fusion is to overlap the execution of the two stages as much as possible without affecting the original generation task's execution time, thereby minimizing the overall execution time of the two stages. Following this objective, we detail the migration triggering, destination, and mechanism below.

Migration triggering. Triggering the migration significantly influences the efficiency of overlapping the two stages. If migration is triggered too early, there remain excessive generation samples. The migration results in only a few model instances remaining available to continue the generation task, as most are redirected to handle inference tasks. This scenario places an excessive load on the few generation instances, leading to prolonged execution times for the original generation task and ultimately extending the overall process. Conversely, if the migration is triggered too late, it loses most of the overlapping opportunity. Based on the above analysis, we propose a migration threshold, denoted as R_t. When *the number of remaining samples in the generation stages falls below* R_t, the migration is triggered. To optimize system performance and minimize total execution time, carefully determining R_t is essential to find the best trade-off.

To determine the optimal R_t, we initially conducted offline generation trials on the training dataset to analyze the length distribution of the generated samples. This analysis enables us to estimate the computational load for both the generation and inference stages accurately and simulate the execution time of the fused execution plan. Due to the determinism and reproducibility of LLM computation, the simulation technique for LLM training [25, 26] and inference [27, 28] is well-studied. We then systematically test R_t values ranging from 5% to 95% of the global batch size, simulating the overall execution time under each R_t. The optimal R_t is identified as the value that yields the minimum simulated execution time. Additionally,

during runtime, we refine the distribution by incorporating new generation samples, which allows us to update R_t as needed to continually optimize performance.

Note that it is possible to achieve finer-grained overlapping by triggering migration more than once. However, we find that in practice, the generation time of long-tailed samples often exceeds the execution time of inference tasks, so one migration is usually sufficient to fully overlap the entire inference stage, as shown in §7.2.

Migration destination. Once the migration timing is determined, we need to select m generation instances to handle the remaining samples. Assume there are n generation instances in total, then there will be $n-m$ instances to be repurposed for inference tasks. To determine m, we revisit the computation characteristics of the generation task. During the decoding phase of generation, which is highly memory-bandwidth-bound, as long as the batch size does not exceed a certain threshold that saturates the GPU (denoted as BS_{max}), the latency of each decoding iteration remains almost constant [28, 29] thanks to the massive parallel computing units of modern GPUs. The value of BS_{max} depends on the specific GPU hardware and can be determined through prior profiling. Based on this, we set the first constraint as $m \geq \frac{R_t}{BS_{max}}$, which ensures that the generation time of long-tailed samples remain unchanged as before the migration.

Another factor to consider is the memory constraint. During LLM generation, the key-value cache is maintained for each token position for autoregressive generation. When the sequence length is long, this memory consumption becomes non-negligible [30]. Therefore, we need to ensure that the target instances have sufficient memory to accommodate the remaining long-tailed samples, avoiding out-of-memory issue or blocking the sample processing. Thus, we set the second constraint as $m \geq \frac{R_t * M}{C}$, where M denotes the key-value cache consumption of the sample with the maximum output length and C is the available GPU memory of the target instance allocated for key-value cache. In summary, the final m is determined by the maximum of the two constraints.

Next, we need to determine which m out of the n generation instances to process the remaining long-tailed samples. To minimize migration overhead, we select the top m instances that have the most samples remaining. This strategy minimizes the total number of samples requiring migration.

Migration mechanism. To migrate an unfinished sample to the target instance, we have two choices. The first approach is to transfer the generated key-value cache to the target instance over the network, allowing it to continue the subsequent generation immediately upon receipt. The overhead in this case primarily comes from the network transmission. The second approach is to discard the key-value cache and only transmit the generated tokens of the sample. Compared to the first approach, this method incurs minimal network latency but comes at the cost of rerunning the prefill phase to generate the key-value cache. The choice of migration mechanism depends on the GPU hardware and network bandwidth. In our experiments, thanks to the high-bandwidth RDMA connections, we choose the first approach and the migration overhead is negligible compared to the iteration time, as shown in §7.2.

After completing the migration, we can release the GPU resources of the remaining $n-m$ generation instances to launch inference tasks. This process involves weights redistribution and data transmission to prepare the context of inference tasks, which we discuss in §6. Once the inference tasks are launched, the remaining long-tailed samples can be streamed to the inference instances as soon as their generation stage is completed, allowing for seamless processing. Additionally, if the long-tail generation task finishes first, we release its resources for the inference tasks.

5 Model-aware Intra-Stage Fusion

In this section, we first analyze the opportunities of fusing the two tasks in the training stage (§5.1). Then we propose a lightweight algorithm to generate the fused pipeline schedule to minimize the pipeline bubbles and memory usage (§5.2).

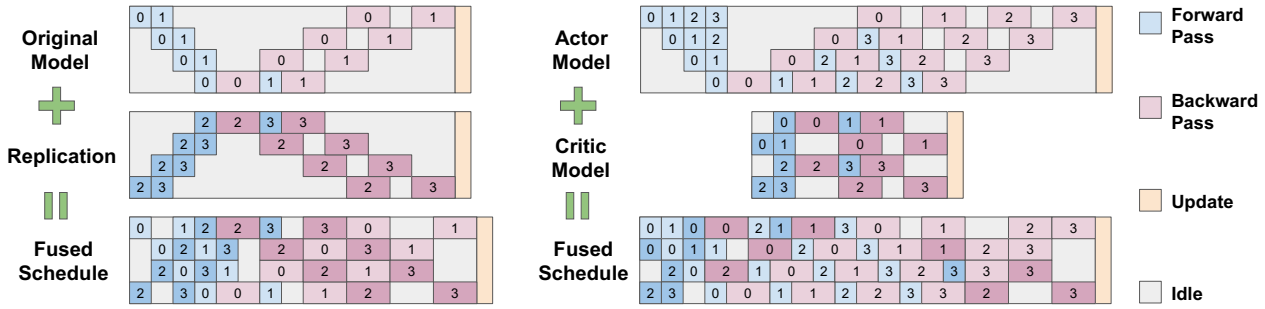
5.1 Opportunities and Challenges

Opportunities. During the RLHF training stage, the `Actor` and `Critic` models are trained independently. Inspired by the bi-directional pipeline schedule [19] from Chimera, we can break the two training tasks into subtasks of micro-batches and co-locate these subtasks to mutually fill the pipeline bubbles of each other. Originally, Chimera enhances single-model training by replicating the model and trains the replicated model with 1F1B schedule in opposite pipeline directions, as depicted in Figure 6(a). This approach not only mitigates pipeline bubbles but also brings a balanced distribution of activation memory. RLHF training stage inherently involves two distinct models, enabling us to apply the fused pipeline schedule without extra model replication.

Challenges. In Chimera, the bi-directional pipeline schedule utilizes one identical model replica with uniform size and parallel strategy, resulting in a symmetric execution flow. In contrast, RLHF involves training heterogeneous `Actor` and `Critic` models, which differ not only in size but also in their optimal parallel strategies. Consequently, the symmetric schedule from Chimera is no longer applicable in the face of model heterogeneity in terms of size and parallel configuration. In the following, we introduce a lightweight and efficient algorithm to automatically generate the fused pipeline schedule under this more generalized setting.

5.2 Fused Pipeline Schedule

Problem transformation. Assume the parallel strategies for models A and B are (dp_1, pp_1, tp_1) and (dp_2, pp_2, tp_2),



(a) Chimera [19] (b) RLHFuse

Figure 6: (a) An example of the symmetric bi-directional pipeline schedule used in Chimera [19] with four pipeline stages and four micro-batchs. (b) An example to show our method can fuse two different models with different numbers of pipeline stages.

Symbol	Description
S	The fused pipeline schedule.
l_{ij}	The latency of subtask S_{ij}.
C	The activation memory capacity of each stage.
K_1	The fusion factor of model A.
K_2	The fusion factor of model B.
N	The total number of pipeline stages to be fused.
N_1	The number of pipeline stages of model A.
N_2	The number of pipeline stages of model B.
M_1	The number of micro-batches for model A.
M_2	The number of micro-batches for model B.

Table 1: Key notations in problem formulation.

which denotes the parallel degree of each 3D-parallelism dimension. Each will utilize all the GPUs in the cluster. We require that tp is the powers of two, which is commonly adopted in practice. For the case where $tp_1 \neq tp_2$, the challenge for a fused pipeline schedule lies in the fact that the pipeline stage of the two models contains a different number of GPUs. Without loss of generality, we assume $tp_1 = s \times tp_2$. In this situation, we merge every s consecutive pipeline stages of model B into one stage, redividing model B into $\frac{pp_2}{s}$ pipeline stages. This ensures that each stage of both models uses the same number of GPU resources. Note that here we require pp_2 to be divisible by s, which is easy to realize in practice. After that, we can transform the problem into fusing K_1 pipeline groups of model A with K_2 pipeline groups of model B, where K_i denotes the fusion factor and K_1 and K_2 are coprime. Figure 6(b) illustrates an example of $(K_1, K_2) = (1, 2)$. Next, we precisely define the fused pipeline schedule and formulate the problem. Table 1 summarizes the key notations.

Problem formulation. We assume the two models to be fused each has N_1 and N_2 pipeline stages ($K_1 \times N_1 = K2 \times N_2 = N$), and each pipeline needs to process M_1 and M_2 micro-batches, respectively. Since the global batch size is fixed, we have $K_1 \times M_1 = K2 \times M_2$. The fused pipeline schedule is represented as a matrix S, where S_{ij} represents the j-th subtask (micro-batch) to be scheduled in the i-th stage. Since there are N stages in total and each stage needs to process the forward and backward computation for all micro-batches of the two models exactly once, the shape of S is $N \times 2(M_1 + M_2)$. The latency of subtask S_{ij} can be profiled in advance and is denoted as l_{ij}. Note that not all S can be scheduled. A valid schedule must satisfy the following constraints:

1. Data dependency: the forward and backward of the same micro-batch across different stages must be executed following the original data dependency.
2. Deadlock avoidance: there exists no cycles in the overall dependency graph to avoid deadlocks.
3. Memory constraint: the peak activation memory usage of each stage must be less than C to avoid out-of-memory.

The main optimization target is to find a valid schedule S that has the minimum latency. Additionally, among schedules with the same latency, we prefer the ones with lower peak activation memory to optimize resource usage.

Algorithm 1 Generate Fused Pipeline Schedule.

```
function GenerateFusedPipelineSchedule(S_0)
    s_current ← S_0
    e_current ← ComputeEnergy(s)
    T ← e_current
    s* ← s_current
    e* ← e_current
    while T > ε do
        s_neighbor ← ComputeNeighbor(s_current)
        e_neighbor ← ComputeEnergy(s_neighbor)
        if e_neighbor < e* then
            s* ← s_neighbor
            e* ← e_neighbor
        if P(e_current, e_neighbor, T) > Rand(0,1) then
            s_current ← s_neighbor
            e_current ← e_neighbor
        T ← T × α
    return s*
```

Algorithm overview: The problem has many constraints and variables. A naive solution is to extend the bi-directional pipeline [19] greedily which always schedules feasible micro-batches. If the micro-batches of two models are both ready, it favors the larger model, with the expectation that the smaller

Algorithm 2 Generate A Random Neighbor Schedule.

```
1: function ComputeNeighbor(S)
2:     while True do
3:         i ← RandInt(1, N)
4:         j ← RandInt(1, 2(M1 + M2) − 1)
5:         S' = Swap(Sij, Si(j+1))
6:         if CheckValid(S') then
7:             return S'
```

one can flexibly fill in the bubbles later. However, the greedy method lacks a global perspective. For example, certain micro-batches could be deliberately delayed to fill the bubbles later. Moreover, it provides no optimization for memory usage. Thus, the greedy approach does not yield optimal performance, as we will demonstrate in §7.3.

Instead, we use simulated annealing [31] to search for a better solution. The motivation for using simulated annealing is that we have a huge search space with many variables. Simulated annealing is effective in finding acceptable local optimums in a reasonable amount of time while finding the global optimum is computationally expensive. Furthermore, we will show that our method can achieve the theoretical lower bound most of the time as shown in §7.3.

At a high level, we use the fused pipeline schedule S as the state in simulated annealing. We use the schedule found by the greedy algorithm mentioned above as the initial state and probabilistically jump to a neighbor state in each iteration, aiming to find a schedule with the lowest latency. Our approach has two benefits. First, it is easy to scale out. We can perform searching using different random seeds across hundreds of CPU cores and select the best result among them. Second, it can easily extend to more models, which arises in multimodal [32] and multi-agent [33] training scenarios. Now we describe the algorithm in more detail below.

Simulated Annealing (Algorithm 1): The algorithm uses the solution found by the greedy algorithm as the initial state S_0, and its execution time as the initial temperature (line 2-3). s^* is used to store the schedule with the lowest latency and e^* is the energy (latency) of s^*. The algorithm searches until temperature T is less than an epsilon value (lines 7-16). T is decreased by a factor of α in every iteration. At each iteration, it uses *ComputeNeighbor* subroutine to find a neighbor state of the current one and uses *ComputeEnergy* to compute the energy of the neighbor state. If the neighbor state has lower energy than s^*, it updates s^* (lines 10-12). The algorithm uses a probabilistic function P to decide whether to transition from the current state to the neighbor state. The probabilistic function P is defined as follows: if the neighbor state has a lower energy than the current state, the probability is 1; otherwise, the probability is $e^{(e_{current}-e_{neighbor})/T}$.

ComputeNeighbor (Algorithm 2): This routine finds a neighbor state of the current state. It randomly swaps two adjacent subtasks from a random stage. If the neighbor state is invalid, it will undo the change and repeat the random swapping until finding a valid schedule. We select adjacent subtasks rather than any two subtasks in one stage primarily to control the degree of disturbance, which is found to be more effective in practice.

Algorithm 3 Compute Energy

```
1: function ComputeEnergy(S)
2:     return max_{1≤i≤N} ComputeFinishTime(S_{i,M})
3: function ComputeFinishTime(Sij)
4:     if Sij ∈ Memo then return Memo.get(Sij)
5:     S^intra ← GetIntraDependency(Sij)
6:     S^inter ← GetInterDependency(Sij)
7:     IntraTime ← ComputeFinishTime(S^intra)
8:     InterTime ← ComputeFinishTime(S^inter)
9:     FinishTime = max(IntraTime, InterTime) + lij
10:    Memo.put(Sij, FinishTime)
11:    return FinishTime
```

ComputeEnergy (Algorithm 3): This function computes the execution time for a given valid schedule S. We use a memoized recursion to calculate the finish time of the last subtask in each stage and pick the maximum as the execution time. For subtask S_{ij}, its start time is decided by two dependencies: inter-stage data dependency and intra-stage data dependency. The inter-stage dependency is the completion of subtask in the upstream pipeline stage corresponding to the same micro-batch. The intra-stage dependency is the completion of the preceding subtask in the same pipeline stage. Consequently, the end time of S_{ij} is determined by the maximum of these two dependencies plus its own computation time, l_{ij}. The detailed algorithm is shown in Algorithm 3. It has time complexity of $O(N \times (M_1 + M_2))$.

Optimizing memory usage. When we obtain a latency-optimized solution S^*, we use it as the initial state to run another round of simulated annealing similar to Algorithm 1. However, this time we replace the *ComputeEnergy* function with one that calculates the peak activation memory for the given schedule. Additionally, we only allow state transitions when the neighbor's latency does not degrade. In this way, we can achieve a solution that not only has promising latency but also optimizes the activation memory usage.

6 Implementation

We implement RLHFuse based on Megatron-LM [15] with 7K lines of code in Python, C++, and CUDA. Megatron-LM applies 3D-parallelism for single-model training. We extend it to support multiple device meshes to launch different tasks asynchronously with tailored parallelism and deployment strategy.

Parallel strategy configuration. Optimizing the parallel strategy for LLM training task is a well-studied problem [6, 15, 26, 34]. Due to the deterministic nature of LLM computation [35],

the execution time and memory cost can be accurately modeled through minimal profiling [25, 27, 28]. An optimal solution can then be found through optimizations [6, 18, 26, 34]. The generation and inference task involves only the forward pass, which can be modeled as a simplified problem of LLM training [27, 28]. ReaLHF [6] is a state-of-the-art solution that optimizes the parallel strategy for RLHF tasks and we adopt a similar model-then-optimize approach. We build a simulator to accurately estimate the runtime statistics of the RLHF task under a specific parallel strategy and workload pattern. Then we follow the guidelines in the Megatron-LM [15] paper to prune the design space and brute-force search the optimal strategy for each RLHF task.

Inter-stage fusion. We extend the simulator mentioned above from the task level to the workflow level. Given the parallel strategies, workload pattern, and migration threshold R_t, it considers the task dependencies and data transmission overhead and simulates the overall execution time of the generation and inference stages. Then we can choose the best migration threshold R_t to minimize the fused execution time.

Intra-stage fusion. We use MPI [36] to parallelize the simulated annealing algorithm used for fused pipeline schedule generation (§5.2). Under different random seeds, the computations are completely independent, so that we can easily scale the computation to hundreds of CPU cores, making it highly likely to find the optimal solution (§7.3). After finding the best solution, it will generate the sequences of NCCL [37] operations on each device following the fused pipeline schedule. The underlying execution engine will follow the instruction flow to execute the exact schedule during runtime. The code of the intra-stage fusion algorithm is publicly available[1].

System optimizations. We adopt a series of system optimizations tailored for RLHF training. Since the RLHF training workflow is well-known, many of the following optimizations are also implemented or partially supported in a similar way in other RLHF frameworks like ReaLHF [6], HybridFlow [5], OpenRLHF [38], DeepSpeed-Chat [39], and PUZZLE [7].

For the generation stage, we implement an in-house inference engine which integrates most of the modern techniques optimized for LLM autoregressive generation like continuous batching [9], prefix sharing [10], and chunked-prefill [40]. For the inference stage, we optimize the computation for Generalized Advantage Estimation [41], which unrolls its original recursive formula along the output length dimension and transforms the recursive computation into a single matrix multiplication, greatly reducing the kernel launch overhead. For the training stage, we evenly distribute each mini-batch across the DP groups [6] based on the sequence length of the samples to ensure that the workloads are roughly balanced across the DP groups, effectively tackling the straggler issue. To reduce the task switching overhead [5–7], we minimize the cross-node communication [42] for the `Actor` and `Critic` models whose latest weights after the training stage need to be redistributed to the appropriate devices according to the new parallel strategies. For the `Ref` and `RW` models whose weights remain unchanged, we keep them in CPU memory and swap them into GPU memory as needed [5, 6], overlapping with the computation of previous tasks.

To separate the differences in low-level system implementations from the core techniques (inter-stage fusion in §4 and intra-stage fusion in §5), our evaluation in §7 uses RLHFuse-Base that includes these optimizations but without inter- and intra-stage fusion as an additional baseline.

[1] https://github.com/FlexFusion/FlexFusion

7 Evaluation

In this section, we evaluate RLHFuse under different sizes of LLMs ranging from 13B to 65B on real-world datasets.

Cluster testbed. We deploy RLHFuse on a production cluster for RLHF training with 32 nodes and 256 GPUs. Each node has 2TB of memory and 8 NVIDIA Hopper GPUs connected with NVLINK. Nodes are connected by $8 * 200$ Gbps RDMA network based on RoCEv2 with rail-optimized topology. The simulated annealing algorithm in §5 is conducted on four CPU nodes, each with 2 AMD 9654 CPUs and there are 768 physical cores in total.

Models and datasets. Following previous work [7], we choose the LLaMA models [43] ranging from 13B to 65B, which is a popular LLM family used in academia and industry. The detailed specifications are listed in Table 2. We use the HH-RLHF [44] dataset, which is open-sourced by Anthropic to train a helpful and harmless assistant with RLHF.

Settings. Since the `Actor` and `Ref` models are of the same size, as are the `Critic` and `RW` models, we choose four different model size settings for `Actor/Critic` pair : 13B/33B, 33B/13B, 33B/65B, and 65B/33B. We also vary the maximum output length to see the performance under different generation settings. In each training iteration, we use a global batch size of 512, a mini-batch size of 64, and take one gradient step per mini-batch following the LLaMA technical report [45].

Metrics. For the end-to-end experiment, we measure the *sample throughput* following previous work [7]. Sample throughput is defined as the average number of samples processed per second. Under each setting, we record the sample throughput over 20 consecutive training iterations after warm-up.

7.1 End-to-End Results

We compare the end-to-end performance of RLHFuse against the following RLHF training frameworks.

- **DeepSpeed-Chat [39]** (DSChat) colocates all the models on the same set of devices and only supports ZeRO-3 data parallelism [46] during training. It utilizes a *HybridEngine*

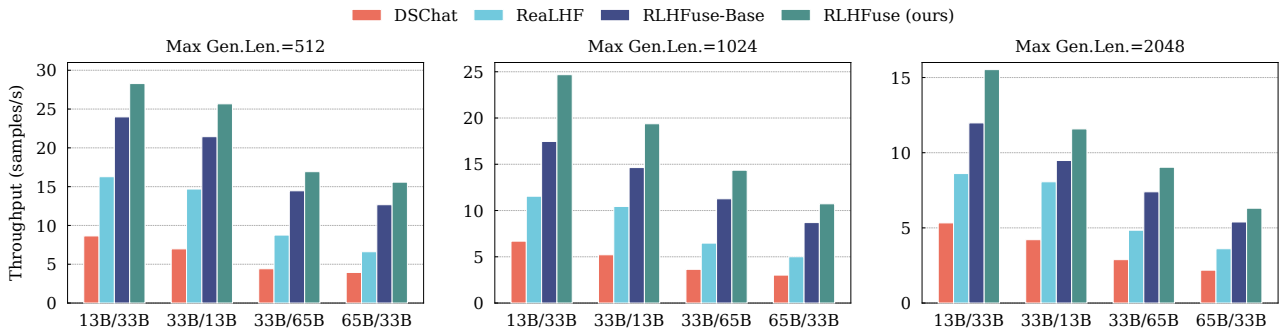


Figure 7: End-to-end throughput of RLHF training systems under different generation and model size settings.

Models	# of Layers	# of Heads	Hidden Size	Intermediate Size
LLaMA-13B	40	40	5120	20480
LLaMA-33B	60	52	6656	26624
LLaMA-65B	80	64	8192	32768

Table 2: LLM specifications.

to switch from ZeRO-3 DP to TP in the generation stage. Since each GPU requires at least one sample during training and it only supports ZeRO-3 DP, we increase its mini-batch size to 256 while maintaining the original global batch size in order to run it successfully on our testbed. Note that with a larger mini-batch size, this adjustment is more favorable for its throughput performance.

- **ReaLHF [6]** proposes parameter reallocation to flexibly redistribute parameters between tasks, enabling tailored 3D-parallel strategy for each task. However, as discussed in §2.2, it suffers from the issues of data skewness and pipeline bubbles without employing the subtask-level optimizations.
- **RLHFuse-Base** is RLHFuse without inter- and intra-stage fusion but with all the system optimizations enabled as described in §6. We include this baseline to demonstrate the performance improvements brought by our fusion techniques in §4 and §5, eliminating any unfair comparisons caused by the differences in underlying framework implementations and other optimization techniques in LLM generation and training which are orthogonal to the proposed stage fusion techniques.

Figure 7 shows the end-to-end performance of the RLHF training systems under three generation settings. We set the maximum generation length to 512, 1024, and 2048, respectively. Under a specific generation setting, the configuration X/Y denotes parameter counts in actor and critic models, mirrored in the reference and reward models, respectively. Compared to DSChat, RLHFuse achieves 2.5×–3.7× higher throughput. This is because DeepSpeed-Chat colocates all the models on the same set of GPUs, making it impossible to adopt the most suitable parallel strategy for each task, which introduces additional computation and communication overhead. ReaLHF implements a flexible execution plan generator, which selects the most efficient 3D-parallel strategy for each task, such as balancing between DP and PP sizes to reduce the number of pipeline stages while avoiding OOM, thereby partially mitigating pipeline bubbles. However, RLHFuse goes further by eliminating the vast majority of bubbles and addressing the long-tail issue in the generation stage with intra- and inter-stage fusion. Additionally, RLHFuse integrates production-grade optimizations in the underlying framework tailored for large-scale training scenarios (§6). As a result, RLHFuse achieves 1.4×–2.4× higher throughput compared to ReaLHF. Compared to RLHFuse-Base, our system achieves a relative improvement of 1.2×–1.4× on the throughput. This improvement is entirely from stage fusion, which significantly alleviates the issues of data skewness in the generation stage and pipeline bubbles in the training stage. In the following, we conduct a detailed breakdown analysis to show the performance improvement.

7.2 Performance Analysis

To further understand the performance improvement of RLHFuse, we show its RLHF iteration breakdown and compare it with RLHFuse-Base. As shown in Figure 8, each row represents a generation setting, and each column represents a model setting X/Y. We divide one RLHF iteration into three main parts: the generation plus inference stage (`Gen.+Inf.`), the training stage (`Train`), and all other overheads (`Others`), such as the data transmission and weights redistribution time.

For `Gen.+Inf.`, since RLHFuse-Base does not perform inter-stage fusion and executes the two stages in serial, we use light and dark colors to represent the time spent on the generation and inference stages respectively to better show the speedup of inter-stage fusion. It can be seen that as the maximum generation length increases, the processing time of long-tailed samples is long enough for RLHFuse to fully overlap the inference stage execution, achieving 1.2×–1.6× speedup. For the training stage, similarly, we use light and dark colors to represent the `Actor` and `Critic` training time in RLHFuse-Base. Through intra-stage fusion, RLHFuse greatly mitigates pipeline bubbles and reduces the execution time of the training stage by 1.2×–1.3×. As for the other overhead, thanks to our optimizations in task switching (§6), it only accounts for less than 3% of the total iteration time, and the migration overhead of inter-stage fusion is also negligible.

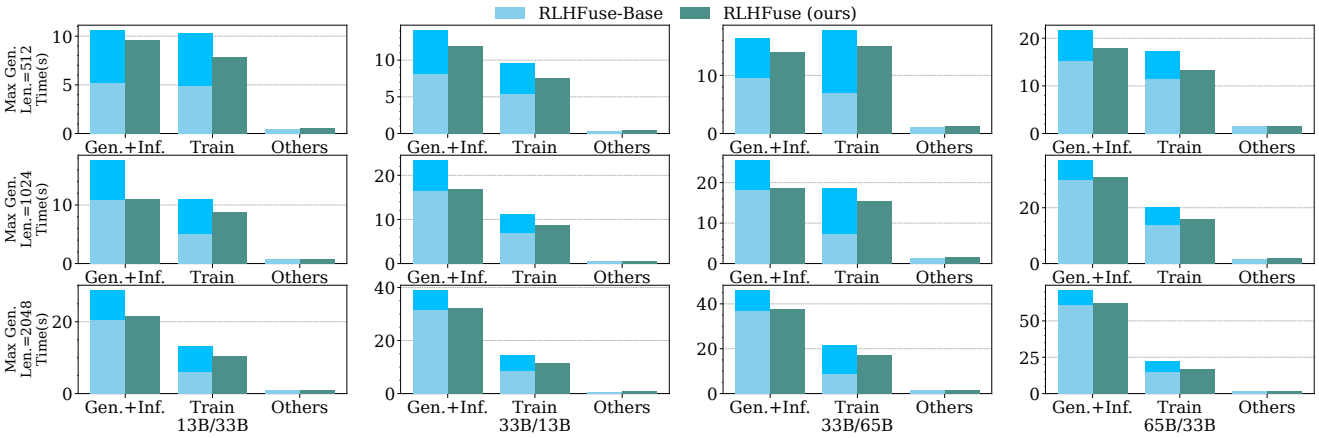


Figure 8: The RLHF iteration breakdown under different generation and model size settings.

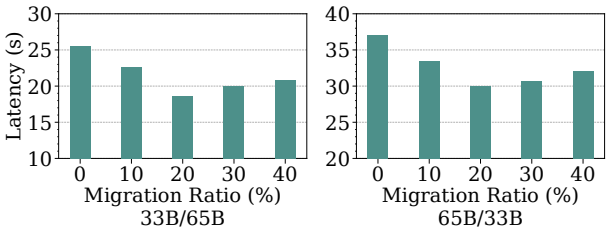


Figure 9: The fused generation and inference stage execution time of RLHFuse under different migration ratios and model settings. The maximum generation length is set to 1024.

7.3 Effectiveness of Stage Fusion

In this section, we show the effectiveness of RLHFuse's key techniques in §4 and §5.

Inter-stage fusion. One key parameter in inter-stage fusion is the migration threshold R_t. As R_t increases from zero, it shifts from fully serial execution to more aggressive fused execution. However, if R_t is set too large, the long-tail generation instances may be overloaded, extending the original generation time. We measure the execution time of the fused generation and inference stage under different migration ratios (R_t/BS) as shown in Figure 9. We can see the optimal latency is achieved when the remaining samples account for about 20% of the batch size. Note that R_t is related to the output length distribution of the model and this experiment is conducted during the early stage of training. As training progresses, R_t requires periodic adjustments to adapt to the distribution change. In practice, such changes are usually not too drastic, so the adjustment frequency remains low.

Intra-stage fusion. We compare RLHFuse's simulated annealing algorithm against the greedy approach mentioned in §5.2 and show the latency speedup and peak activation memory usage relative to the serial execution of the two models with 1F1B schedule in Table 3. For the latency, a naive solution (denoted as 1F1B+) that does not employ a fused pipeline involves using a smaller PP size without out-of-memory errors. However, this approach increases the DP size and results in fewer micro-batches allocated to each pipeline, thereby partially negating the benefits brought by shallower pipelines. Consequently, this solution is less effective compared to the fused pipeline schedule. We also include a lower bound (denoted as LB) estimation calculated as follows: we first compute the earliest possible completion time for each stage, which consists of three parts: the earliest possible arrival time of the first task, the total time required to process all the tasks, and the remaining time needed for the subsequent pipeline stages of the final task; we then take the maximum of these times across all stages as the lower bound of the fused pipeline schedule. Note that there may not necessarily exist a schedule that reaches this lower bound, but we can use it as a metric to assess how close our algorithm is to the optimal solution. Under all the settings, our approach outperforms the baseline methods and achieves the theoretical lower bound except for the last case.

Settings				Latency Speedup relative to 1F1B				Peak Memory relative to 1F1B	
Models	**PP0**	**PP1**	**GBS**	**1F1B+**	**Greedy**	**Ours**	**LB**	**Greedy**	**Ours**
33B/13B	8	4	8	1.10	1.29	**1.38**	**1.38**	1.51	**1.0**
			16	1.06	1.12	**1.30**	**1.30**	2.14	**1.0**
			32	1.03	1.10	**1.15**	**1.15**	2.75	1.26
		8	8	1.07	**1.4**	**1.4**	**1.4**	1.51	**1.0**
			16	1.05	1.15	**1.32**	**1.32**	2.00	1.19
			32	1.03	1.08	**1.17**	**1.17**	2.75	1.31
65B/33B	16	8	16	1.05	1.28	**1.48**	**1.48**	1.61	**1.0**
			32	1.03	1.14	**1.27**	**1.27**	2.23	**1.0**
			64	1.02	1.12	**1.15**	**1.15**	2.88	1.26
		16	16	1.03	1.27	**1.5**	**1.5**	1.57	**1.0**
			32	1.02	1.16	**1.33**	**1.33**	2.00	1.22
			64	1.01	1.09	1.16	**1.17**	2.88	1.47

Table 3: The latency speedup and peak activation memory cost of pipeline schedules found by different algorithms under different models, pipeline stages, and global batch size settings.

As for the memory cost, the serial 1F1B execution serves as the lower bound for the fused schedule. Our method also greatly outperforms the greedy approach and often reaches the lower bound. Even when it does not achieve the lower bound, the extra overhead remains within an acceptable range.

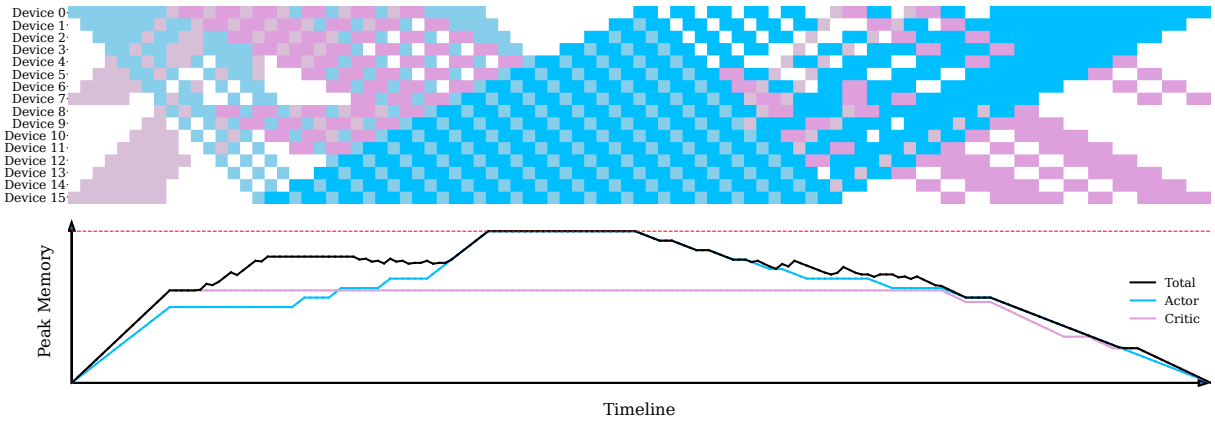


Figure 10: The fused pipeline schedule of 65B `Actor` and 33B `Critic` model generated by RLHFuse. The two models have 16 and 8 PP stages respectively and #micro-batches is set equal to PP. Top: the GPU execution timeline. Bottom: the peak activation memory usage timeline. The red dotted horizontal line denotes the memory cost of serial 1F1B execution, which is the lower bound.

7.4 Deep Dive

In this section, we deep dive into RLHFuse by showing a fused pipeline schedule produced by RLHFuse for the 65B/33B model setting. The upper of Figure 10 shows the GPU execution timeline, with forward and backward tasks in light and dark colors, respectively. With intra-stage fusion, RLHFuse strategically fuses one 65B model of 16 PP stages (blue grids) with two 33B models that each have 8 PP stages (pink grids) to fill the pipeline bubbles of each other in reversed pipeline directions. Ultimately, the execution time of this fused pipeline schedule is the same as individually executing the 65B model using the 1F1B schedule, which means we completely overlap the training of the 33B model, achieving the theoretical lower bound. Meanwhile, as shown at the bottom of Figure 10, the peak activation memory usage also achieves the lower bound of executing the two models in serial with 1F1B schedule.

8 Related Work

RLHF training systems. As RLHF gradually becomes the primary approach for LLM alignment, many frameworks are specifically designed for RLHF training. OpenRLHF [38] uses Ray [47] to distribute models onto separate GPUs and adopts vLLM [8] to accelerate the generation stage. HybridFlow [5] proposes a hierarchical hybrid programming model for the RLHF dataflow and optimizes the GPU allocation and placement of each RLHF model. ReaLHF [6] proposes to design custom parallel strategies for different RLHF tasks to avoid GPU under-utilization. PUZZLE [7] utilizes a lightweight context-switching algorithm to reduce the task switching overhead. These solutions do not consider subtask-level optimization opportunities, thus suffering from data skewness and pipeline bubble issues.

LLM generation optimizations. Many systems focus on optimizing the LLM generation performance [9, 10, 12, 30, 48], and most of them have been adopted by RLHFuse to accelerate the generation stage. Besides, AlpaServe [27] multiplexes the LLM execution with colocated parallelism to improve the throughput under the bursty workload. FastServe [29] proposes preemptive scheduling on iteration-level to mitigate the head-of-line blocking caused by long-tailed samples. Splitwise [49] and DistServe [28] split the prefill and decoding phases to avoid interference between them. These works are orthogonal to RLHFuse and can be integrated into RLHFuse.

LLM training optimizations. LLM training has been studied by many works from various aspects. In particular, [17, 20, 50, 51] use pipeline parallelism to improve the training throughput and reduce memory footprint. Alpa [26] automatically generates the optimal parallel strategy to improve training performance. MegaScale [16] provides a detailed experience in building a production LLM training system at a large scale. DistTrain [52] focuses on addressing the model and data heterogeneity in multimodal LLM training. These works are agnostic to RLHF thus do not utilizes the characteristics to optimize the RLHF training stage.

9 Conclusion

We present RLHFuse, an efficient RLHF training system. RLHFuse views the RLHF workflow from a finer-grained subtask-level perspective and opens up opportunities for efficient inter- and intra-stage fused execution, mitigating data skewness and pipeline bubbles in existing systems. The evaluation shows that RLHFuse can significantly increase GPU utilization, boosting training throughput by up to 3.7× compared to existing systems.

Acknowledgments. We thank our shepherd, Peter Pietzuch, and the anonymous reviewers for their valuable feedback. We thank Shihong Deng, Shilei Jiang, and Heng Wang for providing algorithmic and data support in this work. We thank Wei Fu and Yi Wu for their assistance in running the baseline and for their valuable feedback on the paper.

References

[1] S. Gehman, S. Gururangan, M. Sap, Y. Choi, and N. A. Smith, "Realtoxicityprompts: Evaluating neural toxic degeneration in language models," 2020.

[2] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?," in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, (New York, NY, USA), pp. 610–623, 2021.

[3] S. Lin, J. Hilton, and O. Evans, "Truthfulqa: Measuring how models mimic human falsehoods," 2022.

[4] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022.

[5] G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu, "Hybridflow: A flexible and efficient rlhf framework," *arXiv preprint arXiv:2409.19256*, 2024.

[6] Z. Mei, W. Fu, K. Li, G. Wang, H. Zhang, and Y. Wu, "Realhf: Optimized rlhf training for large language models through parameter reallocation," *arXiv preprint arXiv:2406.14088*, 2024.

[7] K. Lei, Y. Jin, M. Zhai, K. Huang, H. Ye, and J. Zhai, "{PUZZLE}: Efficiently aligning large language models through {Light-Weight} context switch," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 127–140, 2024.

[8] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

[9] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *USENIX OSDI*, 2022.

[10] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Sglang: Efficient execution of structured language model programs," 2024.

[11] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," 2022.

[12] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, K. Chen, H. Dong, and Y. Wang, "Flashdecoding++: Faster large language model inference on gpus," *arXiv preprint arXiv:2311.01282*, 2023.

[13] "Gpt-4o long output." https://openai.com/gpt-4o-long-output/, 2024.

[14] "Introducing openai o1." https://openai.com/o1/, 2024.

[15] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020.

[16] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, *et al.*, "{MegaScale}: Scaling large language model training to more than 10,000 {GPUs}," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 745–760, 2024.

[17] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism ," in *NeurIPS*, 2019.

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

[19] S. Li and T. Hoefler, "Chimera: efficiently training large-scale neural networks with bidirectional pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.

[20] P. Qi, X. Wan, G. Huang, and M. Lin, "Zero bubble pipeline parallelism," *arXiv preprint arXiv:2401.10241*, 2023.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[22] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[23] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. P. Xing, J. E. Gonzalez, I. Stoica, and H. Zhang, "Lmsys-chat-1m: A large-scale real-world llm conversation dataset," 2024.

[24] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025.

[25] J. Bang, Y. Choi, M. Kim, Y. Kim, and M. Rhu, "vtrain: A simulation framework for evaluating cost-effective and compute-optimal large language model training," 2024.

[26] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *USENIX OSDI*, 2022.

[27] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, *et al.*, "Alpaserve: Statistical multiplexing with model parallelism for deep learning serving," *arXiv*, 2023.

[28] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024.

[29] B. Wu, Y. Zhong, Z. Zhang, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[30] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

[31] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, pp. 671–680, 1983.

[32] Z. Zhang, Y. Zhong, R. Ming, H. Hu, J. Sun, Z. Ge, Y. Zhu, and X. Jin, "Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models," 2024.

[33] C. Sun, S. Huang, and D. Pompili, "Llm-based multi-agent reinforcement learning: Current and future directions," *arXiv preprint arXiv:2405.11106*, 2024.

[34] X. Miao, Y. Wang, Y. Jiang, C. Shi, X. Nie, H. Zhang, and B. Cui, "Galvatron: Efficient transformer training over multiple gpus using automatic parallelism," *Proceedings of the VLDB Endowment*, pp. 470–479, 2022.

[35] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *USENIX OSDI*, 2020.

[36] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004.

[37] "Optimized primitives for collective multi-gpu communication." https://github.com/NVIDIA/nccl, 2024.

[38] J. Hu, X. Wu, W. Wang, D. Zhang, Y. Cao, *et al.*, "Openrlhf: An easy-to-use, scalable and high-performance rlhf framework," *arXiv preprint arXiv:2405.11143*, 2024.

[39] Z. Yao, R. Y. Aminabadi, O. Ruwase, S. Rajbhandari, X. Wu, A. A. Awan, J. Rasley, M. Zhang, C. Li, C. Holmes, Z. Zhou, M. Wyatt, M. Smith, L. Kurilenko, H. Qin, M. Tanaka, S. Che, S. L. Song, and Y. He, "Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales," 2023.

[40] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency tradeoff in LLM inference with

Sarathi-Serve," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 117–134, 2024.

[41] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[42] Y. Zhuang, H. Zhao, L. Zheng, Z. Li, E. P. Xing, Q. Ho, J. E. Gonzalez, I. Stoica, and H. Zhang, "On optimizing the communication of model parallelism," 2024.

[43] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[44] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, N. Joseph, S. Kadavath, J. Kernion, T. Conerly, S. El-Showk, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, T. Hume, S. Johnston, S. Kravec, L. Lovitt, N. Nanda, C. Olsson, D. Amodei, T. Brown, J. Clark, S. McCandlish, C. Olah, B. Mann, and J. Kaplan, "Training a helpful and harmless assistant with reinforcement learning from human feedback," 2022.

[45] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[46] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models." ArXiv, May 2020.

[47] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[48] L. Chang, W. Bao, Q. Hou, C. Jiang, N. Zheng, Y. Zhong, X. Zhang, Z. Song, Z. Jiang, H. Lin, *et al.*, "Flux: Fast software-based communication overlap on gpus through kernel fusion," *arXiv preprint arXiv:2406.06858*, 2024.

[49] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132, 2024.

[50] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 431–445, 2021.

[51] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. A. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," 2021.

[52] Z. Zhang, Y. Zhong, R. Ming, H. Hu, J. Sun, Z. Ge, Y. Zhu, and X. Jin, "Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models," *arXiv preprint arXiv:2408.04275*, 2024.

Minder: Faulty Machine Detection for Large-scale Distributed Model Training

Yangtao Deng[1], Xiang Shi[2], Zhuo Jiang[2], Xingjian Zhang[1], Lei Zhang[2]
Zhang Zhang[2], Bo Li[2], Zuquan Song[2], Hang Zhu[2], Gaohong Liu[2]
Fuliang Li[3], Shuguang Wang[2], Haibin Lin[2], Jianxi Ye[2], Minlan Yu[4]
[1]*Tsinghua University* [2]*ByteDance* [3]*Northeastern University* [4]*Harvard University*

Abstract

Large-scale distributed model training requires simultaneous training on up to thousands of machines. Faulty machine detection is critical when an unexpected fault occurs in a machine. From our experience, a training task can encounter two faults per day on average, possibly leading to a halt for hours. To address the drawbacks of the time-consuming and labor-intensive manual scrutiny, we propose Minder, an automatic faulty machine detector for distributed training tasks. The key idea of Minder is to automatically and efficiently detect faulty distinctive monitoring metric patterns, which could last for a period before the entire training task comes to a halt. Minder has been deployed in our production environment for over one year, monitoring daily distributed training tasks where each involves up to thousands of machines. In our real-world fault detection scenarios, Minder can accurately and efficiently react to faults within 3.6 seconds on average, with a precision of 0.904 and F1-score of 0.893.

1 Introduction

Recent years have witnessed a rapid increase in dataset sizes and the number of parameters in models, especially in Large Language Models (LLMs). The GPT-4 model [18], an instance of the Mixture-of-Experts (MoE) paradigm, demonstrates this growth with its 1.8T parameters. Other latest models also exhibit this trend, with parameter counts exceeding 500 billion [24, 69]. The feasibility of training such extensive models efficiently has been realized through large-scale machines and GPUs [39, 40]. It has also been accompanied by advancements in distributed model training [49, 68], high-performance collective communication [48, 65], and fault-tolerant techniques [36]. A system of such vast size and complexity involves a huge amount of computation, communication, and storage resources as well as software support for a task. Consequently, the potential for faults is high, leading to the possibility of task failure.

Faulty machine detection thus has become a significant bottleneck in the maintenance of distributed tasks. In our production environment, an accidental hardware or software fault occurs twice a day on average. The entire task may be forced to stop for hours or days until fixed for retraining. The economic loss for a customer can reach more than $1700 in a 128-machine task for 40 minutes (case in 2.1). Training a GPT-2 model with 1.5 billion parameters and 40GB dataset [67], for instance, takes 200 days utilizing an NVIDIA V100 GPU [10] (or 12 days for a DGX-2H server). If the training process is frequently interrupted by such faults, operating expenses and time costs will increase significantly.

However, the current manual diagnosis method is unsatisfactory. Once a halted task notification is received, the engineer needs to check the training parameters. Meanwhile, engineers from the training, physical networking, storage, and hardware teams, are also involved in diagnosis, since a fault can occur in any machine component. Examining machine logs and conducting offline performance tests on relevant hardware devices are required until the fault is detected (usually for hours). Delayed notifications, incomplete log content, and the complicated process of manual diagnosis amplify the unpredictability of time and labor costs.

It's necessary to design effective and accurate faulty machine detection methods that can quickly react to faults at runtime, not only providing better reliability but also eliminating manual efforts. Achieving such goals is challenging because a machine can fail due to various types of faults. These faults can occur in any possible component, including hardware and software, and can be intra-host or inter-host. Besides, the abnormal pattern of monitoring metrics varies from task to task, making the traditional supervised anomaly detection methods impractical, because even the same behavior might be abnormal in a task with a different workload and machine scale. Additionally, there isn't an individual monitoring metric that necessarily signals a fault. For instance, CPU or GPU usage is the most sensitive metric for fault in-

The first two authors contributed equally to this paper. Zhuo Jiang and Minlan Yu are the corresponding authors. This work was done while Yangtao Deng and Xingjian Zhang were doing a joint research project at ByteDance.

dication, based on our observation from real production data. However, neither one is guaranteed to identify the faulty machine for Error Correction Code (ECC) errors. If noises exist in monitoring data, the detection may be even misguided. Therefore, faulty machine detection for distributed training is challenging.

Instead of creating a monolithic predictor with available monitoring data, we developed Minder by leveraging the ideas of similarity (3.1), continuity (3.2), training individual models for each monitoring metric (3.3), and metric prioritization (3.4). Minder resolves the challenges by recognizing that a machine with a fault displays an abnormal pattern in certain metric data that differs from other machines and lasts for a duration. We also train individual models for data denoising. We then track the dissimilarity between the denoised data for each machine and monitor its duration. By repeating this to individual metrics, the faulty machine is detected. To further expedite detection, we prioritize metrics to identify the most sensitive ones when a fault occurs.

We designed, implemented, and deployed Minder for all the distributed training tasks. Minder operates without interrupting the running of the training machines, only requiring the pulling of monitoring data from the Data APIs for backend run-time analysis. Host metrics used by Minder cover the aspects of computation, communication, and storage. Manual labors are released from the debugging process since Minder can react within 3.6 seconds (6.1), reducing over 99% of the time of manual debugging (shorter time by 500 ×). Minder has an overall precision and F1-score of 0.904 and 0.893.

We make the following contributions.

- An investigation of the fault types and their correlations with various monitoring metrics (2.3). We empirically explain why some metrics are more sensitive to faults and outline the challenges for the detection (2.4).
- The ideas of similarity, continuity, denoising models, and metric prioritization for the design of Minder (3, 4). Our thorough evaluation of Minder's implementation (5) and and ablation experiments highlight its fast reaction, high accuracy, and proper design choices (6).
- Lessons we encountered when deploying Minder in practical (7). We also point out future directions.

2 Motivation

2.1 Negative Impacts of Faults in Real-World Production Environments

Once an unexpected hardware or software fault occurs in a machine, it does harm to the entire distributed training task at the machine level.

Faults are frequent due to the long training duration and large scale. The relationship between a task's machine scale and the daily number of faults is illustrated in Figure 1. Based on our seven-month statistics, the machine scale could significantly exceed 1024, training on more than 10000 GPUs. The occurrence of unexpected faults is highly correlated with the task scale, with an average of two faults a day. If we cannot react quickly to these incidents, such a frequent fault is a burden for training large models.

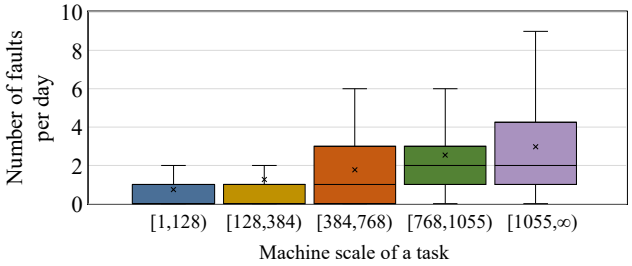


Figure 1: Fault frequency of tasks with different machine scale sizes.

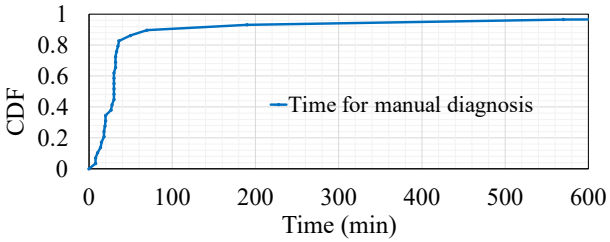


Figure 2: Time for task diagnosis in seven months.

A single fault can cause a large-scale task halt. Based on empirical observations, a fault during the training process often originates as a host problem (*e.g.,* CUDA error, NVLink error [5]). However, this issue can eventually cause a cascade effect, leading to the entire task being interrupted or a noticeable slowdown. For example, a hardware ECC error during communication will make the distributed systems interrupt (*e.g.,* when running in PS [48] or All-reduce [65]) due to NCCL timeout [12] or network disconnections, thereby could resulting in more than half thousand of machines to stop and be in idle state in the production environment.

Faults take a long time to diagnose, increasing labor and resource costs. Based on seven-month data in 2023, the time until the faulty machine can be manually diagnosed is shown in Figure 2. The time lasts over half an hour on average and can be days. It ultimately results in GPU and NIC resources being left idle, leading to additional time costs upon resolution. More delays can result in significantly increased costs and time requirements for users.

An example: PCIe downgrading. We highlight a real-world instance where a 128-machine training was forced to slow down severely for 40 minutes due to a PCIe degradation (PCIe is an intra-host, high-speed serial bus that connects a variety of devices, such as graphics cards, NICs, and solid-state drives). The faulty machine encountered PCIe degradation from 6.4Gbps to 4Gbps. The congested communication ultimately results in the underutilization of computational resources, where thousands of V100 GPUs were compelled to

remain slowed down for 40 minutes. Such cost wastes for customers can be $650 for 40 minutes (given the public renting price of $2.48 per GPU for an NVIDIA V100 [6]).

2.2 Today's Solution and Drawbacks

A straightforward approach after an unexpected fault occurs is inspecting the existing hardware and software logs on the machines, as well as `dmesg` [2], `netstat`, `top` commands. However, this process involves the following three limitations.

The notification of when to trigger a diagnosis is not timely. Firstly, engineers are only alerted once the task has stopped entirely. Certain faults will slow down the training speed in Model FLOPs Utilization (MFU) but are not severe enough to stop the task. The current approach fails to detect degradation in performance as long as the task continues to run. The task will continue running with deteriorated performance for a period in the PCIe downgrading example.

Scrutinized content is incomplete or redundant. Upon a fault notification, the logs or counters recorded during training will be reviewed. The logs are typically maintained in plain text, including built-in software-layer logs (*e.g.,* NCCL and CUDA logs), hardware-layer logs, and network logs. Previous knowledge and experience guide the decision on which logs to include and check, resulting in some log content being overlooked. Meanwhile, logs do not include monitoring metrics like GPU power, temperature, and NVLink bandwidth. In the PCIe degradation case (2.1), identifying the faulty machine was hard as critical monitoring data, like Priority-based Flow Control (PFC) packet rates, was not promptly inspected from the logs. Besides, the log content frequently includes redundant data, such as environmental parameters and warnings. The detection time will be lengthened.

The diagnosis analysis is a complicated and time-consuming process. Engineers from multiple teams are involved in the diagnosing process. Figure 2 presents the time-consuming diagnosis process. It could take as long as several days to detect the faulty machine.

Manual diagnosis procedure and fault propagation for the PCIe downgrading case (2.1). The detection took 40 minutes in total and involved multiple teams. The engineer in charge scrutinized model-related information, parallelism settings [40], dependencies, environmental, and framework (*e.g.,* Megatron-LM [68]) parameters. Meanwhile, the network team scrutinized intra-host throughput, Remote Direct Memory Access (RDMA) traffic, packet loss/randomness, congestion indicators, drivers, and routing. The storage and hardware teams inspected HDFS&SSD usage, GPU&CPU usage, NIC health, and machine scheduling.

The fault propagation initiated from PCIe downgrading to PFC surge. The NIC buffer of the faulty machine was filled after the PCIe degraded. The consequent bottleneck inter-host communication caused a PFC Tx packet surge. The congestion also raised both Explicit Congestion Notifications (ECN) received and Congestion Notification Packets (CNP) sent [17]. As a result, the NIC throughput across all machines dropped from 6.5Gbps to 4.9Gbps. Reduced computation data led to declined GPU tensor core usage [7]. Thus, the training performance was downgraded.

2.3 Real-world Faulty Case Studies

To address the drawbacks of the manual log analysis, we first conducted an in-depth review of the fault types that occurred over seven months. Table 1 shows the common types of faults, their frequencies, and the proportion of instances for each fault type that a metric could indicate. The proportions are determined empirically by examining the available instances and quantifying the number that exhibited abnormal patterns in the monitoring data following a fault. The monitoring data contains multiple metrics and is sampled per second.

We come up with the following observations. Firstly, hardware faults make up the majority of faults (55.8%), in which ECC errors constitute a large proportion (38.9%). Errors that happened in CUDA or GPU also make up a large proportion. Unfortunately, these errors are hard to predict or avoid.

Moreover, each metric displays varying probabilities of indicating a type of fault. There isn't a single metric that effectively signals all of them. ECC error, NIC dropout, GPU card drop, NVLink error, CUDA, and GPU execution error strongly correlate with CPU or GPU metrics. Likewise, PCIe downgrading, NIC dropout, and machine unreachable are most relevant to network metrics, such as PFC and throughput. We explain these observations from our experience. For the CPU metric, a fault on a machine will cease the CPU process, reducing its CPU usage. However, the other machines retain their processes for a while, waiting for data synchronization, due to the Kubernetes management, NCCL timeout setting, and heartbeat mechanism setting with a predefined time window. Thus, these machines maintain normal CPU usage before the task halts. For the GPU metric, a GPU drop or process kill during computation leads to low GPU usage. Conversely, other machines carry on running their CUDA kernels before the timeout and heartbeat check, maintaining normal GPU usage. For the PFC metric, PFC signals on the faulty machine surge abruptly when the NIC buffer is filled, due to congestion-related network errors. Other machines exhibit a low number of PFC packets. The same trend applies to memory usage. Nonetheless, disk usage does not exhibit significant fluctuations based on our experience. Therefore, most fault types strongly correlate with CPU, GPU, Memory, and PCIe metrics. As an exception, AOC errors happen on the switch or the machine-side cables. If a switch AOC error occurs, machines connected to this switch port will be affected instantly. Such a large scale of affected machines can quickly propagate adverse effects to others. It is hard to capture abnormal patterns with second-level monitoring data.

Table 1: Fault types and the proportion of instances for each fault type being indicated by a metric.

Fault type		Frequency of each fault type	Metrics					
			CPU	GPU	PFC	Throughput	Disk	Memory
Intra-host hardware faults (55.8%)	**ECC error**	**38.9%**	80.0%	65.7%	8.6%	45.7%	11.4%	57.1%
	PCIe downgrading	**6.6%**	0.0%	8.3%	100%	33.3%	8.3%	0.0%
	NIC dropout	**5.7%**	100%	100%	0.0%	100%	0.0%	100%
	GPU card drop	**2.0%**	75.0%	70.0%	5.0%	50.0%	20.0%	55.0%
	NVLink error	**1.7%**	83.3%	50.0%	16.7%	50.0%	0.0%	66.7%
	AOC error	**0.9%**	25.0%	25.0%	0.0%	25.0%	25.0%	25.0%
Intra-host software faults (28.0%)	**CUDA execution error**	**14.6%**	61.9%	57.1%	19.0%	33.3%	14.3%	61.9%
	GPU execution error	**7.7%**	50.0%	71.4%	14.3%	42.9%	21.4%	42.8%
	HDFS error	**5.7%**	57.1%	57.1%	0.0%	14.3%	0%	14.3%
Inter-host network faults (6.0%)	**Machine unreachable**	**6.0%**	47.4%	63.2%	0.0%	53.6%	26.3%	15.8%
Others (10.3%)	-	**10.3%**	-	-	-	-	-	-

Note: the introduction of each fault type is in Appendix A.

2.4 Challenges

Based on the observations, we discovered that the following challenges must be addressed:

Challenge 1: Any machine could fail in various ways. Advanced machines, such as Nvidia DGX-A100 [11], incorporate as many as 8 Nvidia A100 GPUs and 4 Mellanox 200 Gb/s RDMA NICs (RNIC), all of which are potential fault points. As presented in Table 1, a fault could happen from intra-host computation, and communication, to inter-host networks, or from hardware (GPU, CPU, PCIe, NVLink, RNIC, memory, and disk) to software communication libraries (*e.g.,* NCCL), and training frameworks (*e.g.,* CUDA). It is hard to detect a faulty machine under all the unknown circumstances.

Challenge 2: The normal state of a monitoring metric is task-dependent. Training tasks have various machine scales, data sizes, models, and training frameworks. Thus, a monitoring metric may have different normal states for various tasks. For instance, a GPU temperature of 70 degrees Celsius is abnormal where GPU clock frequency is 1350MHz, but is regarded as normal in a task where GPUs work with a high clock frequency of 1800MHz. Traditional supervised anomaly detection is inappropriate for differentiating between normal and faulty machine states because the same input monitoring data can be labeled as either normal or abnormal.

Challenge 3: The correlation between fault types and monitoring metrics is not necessarily one-to-one. On one hand, a metric anomaly may be caused by various fault types. For example, a decrease in CPU usage could be caused by ECC error, NVLink error, and other faults. On the other hand, for a fault type, there isn't a single metric that necessarily signals it. As an example in Table 1, ECC errors could potentially be noticed by either CPU or GPU usage of the faulty machine, yet neither metric guarantees nor necessarily both. Thus, a single fault type can be manifested via multiple abnormal metrics, with no solitary metric providing a guaranteed indication. Instead, the metrics exhibit an "or" correlation for a fault type. Consequently, we cannot simply use a model that incorporates the data from all metrics. Rather, we use individual per-metric models for detection (analysis in 6.3).

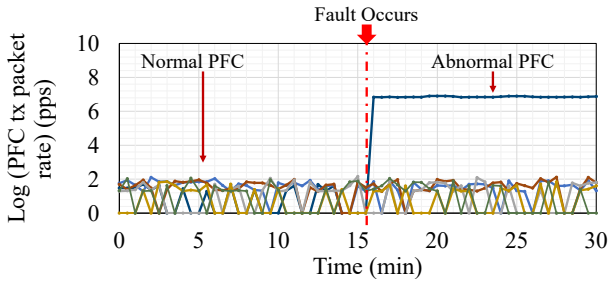


Figure 3: PFC tx packet rate pattern for each machine before and after a fault occurs.

Challenge 4: Noises exist in time series monitoring data. The monitoring data inevitably consists of noises due to jitters, inaccurate sensors, temperature, timestamp misalignment, network interruptions, or other issues. Short-term noises will mislead us to regard a machine as the faulty one, resulting in extra time and labor burden. Thus, the raw data cannot be used directly for detection (analysis in 6.3).

3 Design Overview

In this section, we introduce four design choices to address the challenges, revolving around Minder: an automatic, responsive, and accurate watcher to detect the machine with an unpredicted fault during distributed training that leads to a task halt. We examine the dissimilarity (3.1) among machines. Then we evaluate the continuity (3.2) of the detected faulty machine candidate for filtering out bursty jitters. Since metric data contains noises, we use models to denoise and reconstruct the raw data. Specifically, individual models corresponding to various monitoring metrics are trained, instead of integrating them into a single model (3.3). We also prioritize the metrics (3.4). The faulty machine can be detected swiftly by the top-prioritized metrics and corresponding models.

3.1 Machine-level Similarity

To address challenges 1 and 2, we notice that machine-level metric data exhibit similar behavior in parallel distributed training [68] at the second level.

Data parallelism (DP) operates by splitting data equally among GPUs that store duplicate model parameters and optimizer states. Pipeline parallelism (PP) assigns model layers to multiple GPUs in a pipelined manner. In tensor parallelism (TP), each GPU executes a portion of a computation process to improve hardware utilization in parallel.

By integrating these techniques into a 3D parallelism framework, large-scale model training, such as LLM training [40] or multi-modal training, can be effectively facilitated. TP is typically constrained within a single machine, whereas DP or PP groups involve inter-host communications. The computation, storage, and communication loads are evenly balanced across machines at the second level. Therefore, similar monitoring data fluctuations will be observed across machines. In the PCIe downgrading example, the initial `PFC Tx Packet Rate` patterns are notably uniform for all the machines in Figure 3. However, if a machine undergoes a fault, its monitoring data will display distinctive differences, offering an opportunity for detection. This principle can be extended to other metrics, as demonstrated by the possibilities offered in Table 1. Hence, the denoised metric data from each machine is used for calculating the similarity with others. The machine identified with the longest "distance" from others could be assumed to be faulty, regardless of the fault type.

Why not use a supervised learning model for faulty machine detection? Unlike many supervised learning-based anomaly detection approaches [21, 27, 45, 53], Minder uses unsupervised learning and similarity-based distance check, due to the different problem contexts. Firstly, labeling the data as normal or abnormal is impractical. As mentioned in challenge 2, the abnormal pattern is task-dependent. Different tasks present different normal ranges for the same monitoring metric under varying working conditions. Secondly, our objective is to identify which machine is to be blamed for the unexpected fault. This is not merely a classification problem of distinguishing normal or abnormal cases. Thus, developing a universal model via supervised training is challenging.

3.2 Machine-level Continuity

To further tackle challenge 2, we utilize the notion of abnormality continuity. Typically, abnormal performance persists for a few minutes upon a fault, but jitters typically last for a short duration. As illustrated in Figure 3, the machine with PCIe degradation experienced significantly higher `PFC Tx Packet Rate` for a period compared to other machines. By inspecting fault instances from seven-month data in 2023, the duration of abnormal performance after a fault occurs is depicted in Figure 4. Most abnormal patterns last for over five

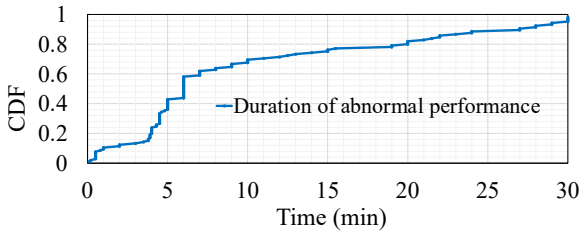


Figure 4: Duration of abnormal performance following a fault.

minutes. Thus, if we recognize a machine displaying such dissimilarity continuously for a period, the machine may be faulty. In the case of raw data containing bursty noises, they will be filtered out (as analyzed in 6.4).

3.3 Individual Learning-Based Denoising Models for Each Monitoring Metric

For challenge 4, we utilize simple learning models for data denoising and reconstruction, in addition to basic data alignment and normalization. Variational autoencoder (VAE) and other generative probabilistic models are recognized for learning time-series data patterns and features [52, 70]. They are also known for learning embedding schemes that can infer the generation factors for most of the training data. This makes unsupervised learning particularly suited for modeling normal behavior in an anomaly detection task. As such, raw data is denoised and reconstructed by our learning-based models before being fed into further detection (as analyzed in 6.3).

Meanwhile, based on challenge 3, no single metric provides a guaranteed indication for a specific fault type. The indication probability by a metric varies across different types of faults. For this reason, we opt for training individual models for each monitoring metric. These models and their corresponding metric data are used independently for denoising, similarity, and continuity detection.

We do not merge all potential metrics into a single model for two main reasons. First, the time series of multiple metrics do not fluctuate in the same manner when a fault arises. Moreover, metrics' indication capacity differs even for a particular fault type, and one metric's capability varies across different types. As a result, integrating them into one model could lead to the model being misdirected or confused by the array of metrics (as analyzed in 6.3).

3.4 Prioritized Metric Sequence

To expedite detection, we prioritize metrics and only use the models of the top ones, since plenty of metrics (in Appendix B) could be collected and each could be trained with a model. This ensures we use the metrics with higher indication probabilities earlier. The faulty machine could therefore be detected sooner. If a model and its associated metric data cannot detect a faulty machine, we then move to the next metric

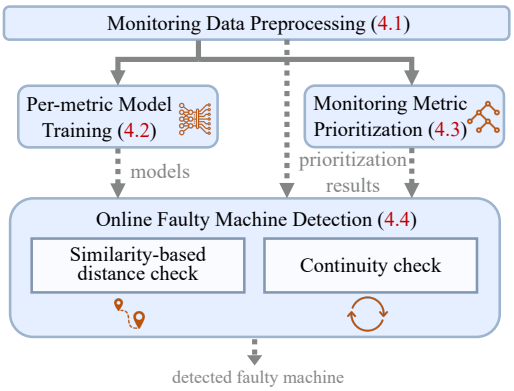


Figure 5: System architecture of Minder.

and its model, following the prioritization result. We repeat this process until a faulty machine is identified.

4 Minder Framework

The architecture of Minder is shown in Figure 5. Preprocessing (4.1) is required for raw data from each machine. Per-metric Model Training (4.2) and Monitoring Metric Prioritization (4.3) will train models and prioritize the metrics in their sensitivity to faults respectively, as two independent processes. The models and prioritization results are then used in Online Faulty Machine Detection (4.4) for run-time detection.

4.1 Preprocessing

Given a stream of monitoring data from each machine, Minder needs to aggregate them into a series of time windows. Within each window, Minder can do data denoising and machine-level similarity check on a set of metrics. By checking the anomaly continuity from consecutive time windows, Minder detects the faulty machine.

Thus, Minder preprocesses the collected monitoring data if it lacks alignment among certain metrics. Minder first aligns the sampling points across all machines based on the corresponding sampling timestamps. If sample points are missed, Minder uses data from the nearest sampling time for padding.

Normalization is adopted to ensure that the multi-dimensional monitoring data is integrated into an even distribution. Minder normalizes the monitoring data based on the upper and lower limits of each metric, using the Min-Max normalization technique.

4.2 Per-metric Model Training

As specified in 3.3, we train models for the learning-based denoising and reconstruction for subsequent detection. Since no single metric provides a guaranteed indication and a model incorporating various metrics might be misdirected, individual models should be trained for each metric.

The preprocessed per-machine data within a time window is used as input instances to train an unsupervised model. For

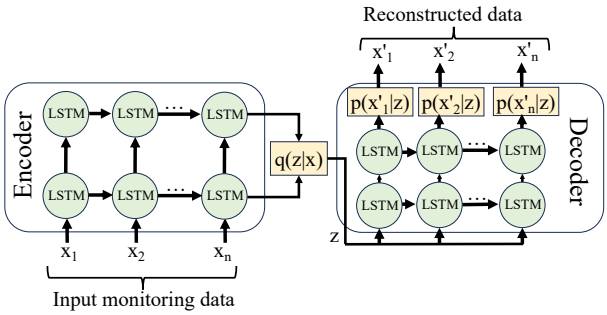


Figure 6: The LSTM-VAE structure for Minder.

example, to train a model for `CPU Usage`, we use `CPU Usage` sample data within a time window with a length of w (*e.g.,* 8) and a stride of 1 from each machine of the task. Multiple $1 \times w$ vectors are fed into the model respectively for training. Models for `CPU Usage`, `PFC Packet Rates`, and so on are individually trained. The parameters in the model include *hidden_size* (*e.g.,* 4), *latent_size* (*e.g.,* 8), and *lstm_layer* (*e.g.,* 1). With a model, the time series input of a monitoring metric from a machine could be reconstructed as a denoised vector for this machine.

Training choice: LSTM-VAE. Specifically, VAE models are trained for Minder. VAE is widely used for denoising [70] and compression of high-dimensional features [71], where normal vectors will be reconstructed into similar embeddings while abnormal ones will be reshaped into a more distinctive outlier. This highly effective unsupervised DL technique can reconstruct time series inputs into an arbitrary latent representation without sacrificing original characteristics. Consequently, VAE can enhance the accuracy and robustness of anomaly detection where labeling is absent [78]. When the input consists mainly of normal training state vectors and only a small proportion corresponds to a faulty period, the VAE learns the vector distribution and denoises the jitters.

As depicted in Figure 6, the VAE comprises an encoder and a decoder. The encoder extracts temporal features into a latent space embedding z. Subsequently, the decoder utilizes z to restore the data to a new dimension output as a reconstruction of the distribution. Various statistical or machine learning techniques can be employed in the encoder and decoder to determine the optimal distributions. Given that our data is temporal time series, we utilize LSTM as both the encoder and decoder to extract temporal characteristics [52]. LSTM considers both forward and backward information of a time series to obtain complete correspondence information. As such, LSTM is an ideal choice for VAE.

4.3 Monitoring Metric Prioritization

As introduced in 3.4, we aim to use only the most representative metrics and their models for quick faulty machine detection. Given a large number of metrics, pinpointing the metrics more sensitive to faults is critical. By using the top prioritized metrics and their associated models first, the faulty

machine will be detected more quickly.

Consequently, following the steps below, Minder will generate a prioritized list of metrics by their sensitivity to faults. The prioritization results can then be used in run-time detection, specifying which metrics and their models should be used first. Note that this process runs in parallel with the model training process in 4.2.

Step 1: Z-score calculation for evaluating metric sensitivity to faults. To identify the most sensitive metrics, Minder utilizes the Z-score [23], because it depicts the dispersion of data distribution. A metric with a higher Z-score relates to an imbalanced distribution, where a faulty machine shows a dissimilar pattern from others. For a monitoring metric, the Z-score is computed for each machine at a sampling data point from the preprocessed data. For the j-th monitoring metric:

$$Z_{ij} = \frac{x_{ij} - \bar{x_j}}{s_j}$$

where Z_{ij} is the Z-score of the i-th machine, x_{ij} is the sample value of the i-th machine, while $\bar{x_j}$ and s_j are the average value and the standard deviation of all machines on the j-th metric. When a fault occurs, the affected machine exhibits abnormal behavior, leading to outlier samples with high Z-scores.

For a time window of a training task, we use $max(Z_{ij})$ across all the machines for the j-th monitoring metric, indicating the extent of the dispersion among machines.

Step 2: Prioritization of the monitoring metrics. Based on the maximum Z-score for each monitoring metric, Minder uses a decision tree [32, 60] to prioritize the sensitivity of each metric in identifying the faulty machine. We resort to a decision tree for two primary reasons. Firstly, the logical structure of decision trees bears a resemblance to rule-based policies that are common in networking monitoring systems. For example, certain monitors utilize simple threshold rules, such as when `CPU Usage` drops to nearly zero [83]. Secondly, decision trees offer high expressiveness and faithfulness, attributed to their lack of parameters and the ability to represent complex decision-making [22].

To construct a decision tree, Minder gathers the maximum Z-score for each metric from step 1 as an individual instance for the time window of the training task. The instance is labeled manually as normal or abnormal depending on whether a faulty machine exists within this window. Instances across multiple time windows and multiple training tasks are used together to train a decision tree.

As shown in Figure 7, monitoring metrics are prioritized based on their sensitivity to faults. The decision tree employs a step-by-step approach to classify the instances by analyzing the Z-scores of each metric. Nodes located closer to the root of the tree indicate that the corresponding monitoring metrics are more sensitive to the occurrence of a faulty machine. PFC, CPU, GPU, and NVLink-related metrics are identified as the most informative ones. Specifically, `CPU Usage` is relevant to running process states, while the four GPU metrics relate

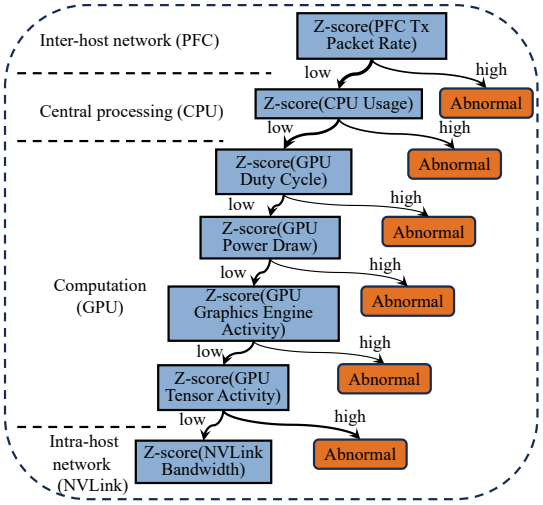


Figure 7: Top 7 layers of the decision tree for prioritization.

to the computation states. Additionally, `NVLink Bandwidth` and `PFC Packet Rates` are indicative of intra-host and inter-host network quality. The outcome aligns with Table 1, where CPU and GPU enjoy the highest priority.

4.4 Online Faulty Machine Detection

During run-time detection, Minder leverages the concepts of machine-level similarity (3.1) and continuity (3.2), since the faulty machine tends to exhibit an abnormal pattern over a period. Given the data from each machine, Minder follows the order of the decision tree prioritization results and selects a metric. The metric data is then denoised by the correlated model. Minder then runs step 1&2 introduced below for detection. If none is detected, Minder picks the next metric and repeats step 1&2 until one is detected. Fortunately, if no one is detected after passing all the models, Minder assumes no anomaly occurs up to this time.

Step 1: Similarity-based distance check per time window. To detect a faulty machine during a time window, Minder compares the similarity (3.1) among all the machines for the same metric. In the presence of a faulty machine, its denoised data from LSTM-VAE is distinguishable from others.

To initiate the detection for a time window, the monitoring data from each machine (*e.g.,* a $1 \times w$ vector for a machine) is fed into the corresponding model successively. The reconstructed embedding for the i-th machine is captured by Minder for the following distance calculation. Specifically, Minder calculates the pairwise Euclidean distances of embeddings between every two machines, as it expresses distinct differences between normal and abnormal samples and provides characterizations of various record types [76]. For each machine, Minder calculates the sum of the distances to other machines, representing its dissimilarity. Since the distance magnitude shifts with machine scales, we calculate the normal score for each sum value of the machines to normalize.

The machine with the maximum normal score is probably the faulty one. If the maximum normal score is higher than a *similarity threshold*, the machine is assumed as a candidate of the time window.

Step 2: Continuity check for consecutive time windows. The detected candidate of a time window might be a false alarm due to instant bursts or temporary counter noises, so the idea of continuity (3.2) is essential. This is because faults often lead to deteriorated performance for a period. Minder shifts the time window with a stride of one to detect the potentially faulty machine for new windows. If the same machine is detected with consecutive times that exceed a *continuity threshold*, it is considered a truly faulty machine. A proper *continuity threshold* can be set as 4 minutes as it is adequate to filter out short-term noises and will not exceed the typical lasting time of a deteriorated performance in Figure 3.

5 Implementation

Minder has been deployed in our ML system for a year. It runs on a dedicated machine with two dual-port ConnextX-6 25G-RNICs [9], 128 Intel Xeon Platinum 8336C CPUs, 512G memory, and 1.6T disk. The high-speed RNICs ensure the fast transmission of monitoring data, while computation and storage resources are adequate for real-time detection.

Minder monitors all the ongoing training tasks throughout their life cycles in our production environment. For a task, Minder is called at pre-determined intervals (*e.g.,* every 8 minutes). Upon a call, Minder pulls 15-minute data for the metrics listed in Appendix B from a database for all machines associated with the task. The metrics cover aspects of computation, communication, and storage. The database updates monitoring data per second from all the machines. If Minder identifies a faulty machine, an alert is triggered to a driver and relevant engineers. After the driver submits the machine IP to be blocked and the Pod information to Kubernetes, the faulty machine will be evicted and replaced by a new one, before a fast recovery from recent checkpoints [40]. Importantly, the running of Minder will not interfere with online distributed training tasks, as Minder works as a backend service.

Task workload. The monitored training tasks are distributed across four to over 1000 machines, with GPU numbers up to more than 10000 for a task. Concurrent tasks could be monitored by Minder. Each task is running on machines of homogeneous GPU and RNIC architectures on rail-optimized topology with up to three layers of switches. TP for computation, PP for gradient calculation and propagation, and DP that performs gradient synchronization are efficiently used for our LLM pre-training. These 3D parallelism strategies [40, 68] (3.1) facilitate balanced computation (GPU usage, power, temperature *etc.*), storage (memory usage *etc.*), and communication (intra-host and inter-host throughput) across machines. Models trained in our ML system include the Transformer [74] and so on. The model sizes range from under 32B to over 500B.

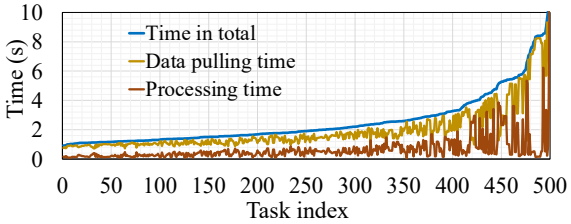


Figure 8: The total data processing time for a call of Minder.

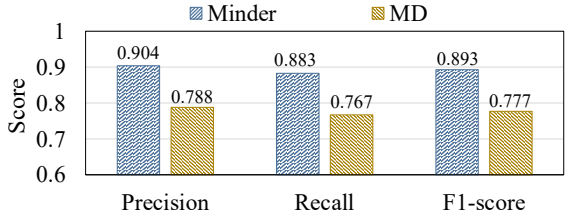


Figure 9: Comparison with a baseline algorithm MD [30, 46].

6 Evaluation

Dataset. Our dataset includes 150 run-time fault instances from online distributed training tasks with 3D parallelism [40]. The dataset includes instances over nine months. Each task involves 4 to over 1500 machines (up to 10,000 NVIDIA Ampere GPUs), covering all the scale groups in Figure 1. 30% of the tasks involve a minimum of 600 machines. All fault types listed in Table 1 are covered. The dominant ones are ECC error (25.7%), CUDA execution error (15%), GPU execution error (10%), and PCIe downgrading (8.6%). Our dataset focuses on faults in an individual machine, as they account for 99% of all faulty cases in the production environment (6.6 for concurrent faulty machine evaluation). The monitoring metrics in Appendix B were collected at the second-level granularity. Due to the fast eviction and recovery process, verifying whether all evicted machines are faulty is challenging. Consequently, our dataset encompasses the run-time instances where the actual faulty machine could be manually confirmed via offline log-checking, nccl-tests, or hardware tests (*e.g.,* [55]). For LSTM-VAE training, we use data from the first three months and the rest for evaluation.

Metrics. For a task, we denote true positives (TP) as the correct machine detection following a fault, and false negatives (FN) as errors in machine detection or missed detections during a fault. True negatives (TN) refer to the correct approvals when machines are running normally, while false positives (FP) refer to false detections when there is no fault. Then we calculate Precision, Recall, and F1-score as our metrics.

6.1 Overall Performance

Total data processing time. In Figure 8, a call of Minder takes 3.6 seconds on average to make an alert. It includes data pulling time (fetching the pertinent monitoring data from

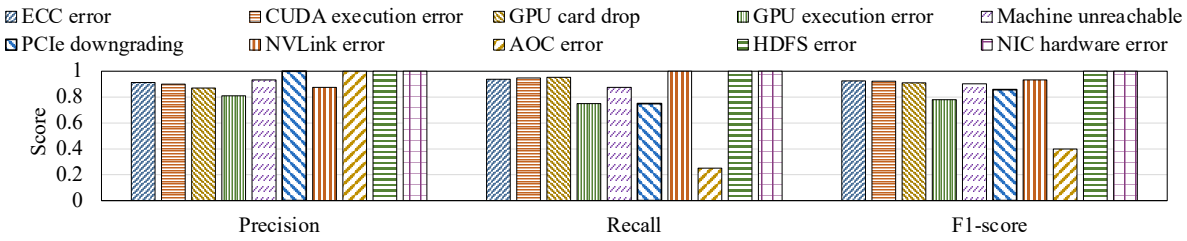


Figure 10: Accuracy for various fault types.

Data APIs) and processing time (preprocessing, and running inference for faulty machine detection). Due to Minder's deployment on an exclusive machine, it reduces the time by 99% (shorter time by 500 × compared with Figure 2) if engineers otherwise manually inspect machine information one by one.

Comparison with the baseline. The baseline for comparison is Mahalanobis Distance (MD) [30, 46, 57]. MD is widely used in identifying outliers. It considers the variable correlations in multi-dimensional data and calculates features like mean, variance, skewness, and kurtosis before applying principle component analysis (PCA) and computing the pairwise distances. We keep other processes the same for comparison.

In Figure 9, the precisions are 0.904&0.788, with recalls of 0.883&0.767, leading to F-1 scores of 0.893&0.777 for Minder and MD. These findings demonstrate that Minder effectively detects actual faults and lowers the rate of false alarms. MD detects the outlier machine from the statistical perspective but exhibits lower accuracy, indicating that jitters and noises interfere with statistical features. Minder outperforms MD by using LSTM-VAE for denoising and extracting the data patterns for a better distance calculation.

Performance breakdown with fault types. By considering individual fault types in Table 1, the results slightly differ. In Figure 10, Minder effectively handles faults like ECC error, CUDA execution error, GPU card drop, machine unreachable, NVLink error, HDFS error, and NIC hardware error. These faults are relative to CPU, GPU states, and networking performances that Minder monitors.

GPU execution error and PCIe downgrading present a lower recall, since some faulty instances have concurrent faulty GPUs or PCIe links within a machine. Owing to the 3D parallel topology, faults swiftly impact multiple machines in both the DP and PP groups, leading to an instant group effect. Thus, the time-series granularity at the second level is insufficient for Minder. Other fault types, such as AOC errors, are partially missed due to the current absence of optical cable-related counters for capturing useful monitoring metrics. We will continue to have more hardware counters in the future. Overall, AOC errors occupy only a small portion, so the overall accuracy is still high.

Upon examining the errors made by Minder, most of them were not entirely incorrect. In many error cases, the machine incorrectly detected showed short-term metric fluctuations or performance jitters before being normalized afterward. These detected fluctuations, however, were not the root cause of the task halt. Nonetheless, these errors should not be ignored as they led to short-term jitters in the training performance.

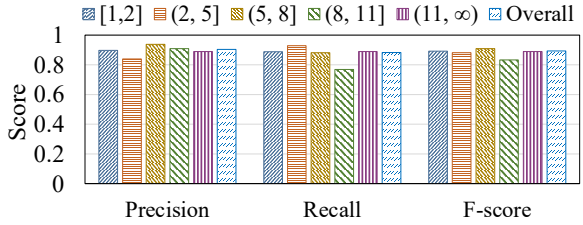


Figure 11: Accuracy for tasks with varied lifecycle fault occurrences.

Performance breakdown by fault occurrences of a task's lifetime. Training tasks with various workloads experience distinct occurrences of faults. Figure 11 categorizes the performance relative to the fault numbers throughout a task's lifetime, which may span up to months for extensive workloads. In our dataset, 70% of the tasks display no more than five faults, whereas over 15% face more than eight faults throughout their lifetime. Given the lack of interdependence among individual faults, the accuracy is not tied to the fault occurrences. Since faults are random and a faulty machine will be promptly auto-replaced in the production environment, the fault occurrences do not affect performance. Despite the lower recall for the (8, 11] group, it primarily originates from the limited task number of this group.

6.2 Analysis of Monitoring Metric Selection

To validate the proper selection of monitoring metrics in (4.3), we conduct experiments to show that fewer or more metrics do not improve accuracy. Notably, most of the remaining metrics not used by Minder are GPU-related, including `GPU Temperature`, `GPU Clocks`, `GPU Memory Bandwidth Usage`, and `GPU FP Engine Activity`. Thus, we use only `GPU Duty Cycle` to train a GPU model with fewer metrics, while adding these unused GPU-related monitoring metrics to train a GPU model with more metrics. We keep the other settings unchanged for the comparison.

Figure 12 reveals that including more monitoring metrics achieves a higher recall, but its precision is lower. More metrics may introduce mutual interference, since different metrics may indicate different patterns that will obfuscate the detection. On the other hand, using fewer metrics undermines outlier detection capacity due to the exclusion of key metrics.

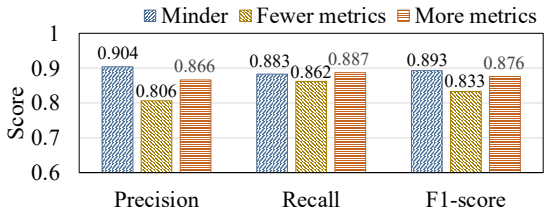


Figure 12: Comparison with different metric selections.

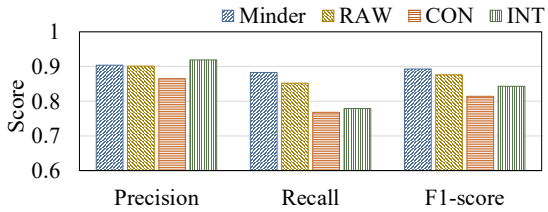


Figure 13: Comparison with different model selections.

Our optimal selection of metrics achieves the best precision, meaning that Minder alerts far fewer false alarms than others. Based on the metric priority result (4.3), the top metrics are sufficient to cover all components that might malfunction.

6.3 Analysis of Model Selection

We contrast Minder against other statistical methods or model variations to show the proper choice of LSTM-VAE. A simple approach is calculating the Euclidean Distances of the preprocessed raw data (RAW) without using VAE. Variants of LSTM-VAE include concatenating the embeddings of all the models as a whole for distance calculation (CON) or training an integrated LSTM-VAE model with all the monitoring metrics (INT).

In Figure 13, Minder outperforms others in recall and F1-score. Their recalls vary. Worse recall generated by RAW illustrates the significance of eliminating noises. Minder, in contrast, reconstructs for denoising. CON and INT show worse recall because they consider multiple metrics with equal significance by calculating distances from evenly concatenated embeddings or regarding all the metrics as a whole for input. However, not all monitoring metrics have an equal sensitivity to faults. Mutual interference exacerbates the performance of CON and INT. Minder, on the other hand, leverages the VAE for denoising and separates metrics to enhance the performance. Besides, comparing the input and reconstructed data of LSTM-VAE yields a Mean Squared Error (MSE) lower than 0.0001, demonstrating effective reconstruction.

6.4 Analysis of Continuity and Threshold

To verify the feasibility of continuous detection, we compare Minder without the application of continuity (3.2). Minder ensures the same machine is detected multiple times. Without continuity, an alert will be made immediately upon a fault detection during the time window. The results in Figure 14 imply that the overall performance is worse without continuity.

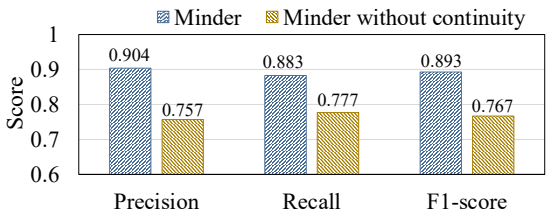


Figure 14: Accuracy with/without continuity.

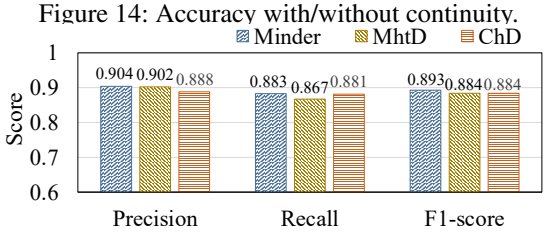


Figure 15: Comparison with different distance measures.

More false alarms are triggered due to occasional short-term jitters. The continuity emphasizes the degraded performance for a period caused by a fault's gradual impact on other machines. Thus, Minder filters discrete bugs, noises, or jitters.

Note that we set the *continuity threshold* (4.4) as four minutes to reduce false alarms caused by jitters or noises. That means Minder only alerts when a detected machine endures dissimilarity for four minutes. The threshold is chosen empirically based on the fault duration in Figure 4. Most of the faults last longer than four minutes before the halt. A shorter continuity threshold introduces more false alarms, while a longer one excludes more actual faulty detection results.

6.5 Choice of Distance Measures

We compare Minder 's pair-wise Euclidean Distance algorithm with Manhattan Distance (MhtD) and Chebyshev Distance (ChD). MhtD adds up the absolute distances from each dimension of the embeddings while ChD uses the largest difference in their coordinates. We replace the distance algorithm to rerun the experiments.

In Figure 15, Minder achieves similar performance to others, suggesting that the embeddings from LSTM-VAE are already representative and the outlier is distinctive for any distance calculation method. The comparison with MhtD implies that spatial distribution is a solid representation because both methods use the distances of multiple dimensions. ChD's worse precision suggests that a single spatial difference is insufficient for comparing dissimilarity. Minder considers the overall distance from the outlier to other normal ones and the dissimilarity will be intensified by summing the distances.

6.6 Performance with Multiple Concurrent Faulty Machines

Analyzing Minder's ability to handle multiple faults is intuitive. Such detection capability largely depends on the faulty machine scale ratio and the granularity of monitoring data.

Given the 3D parallelism [40], a machine participates in multiple DP and PP groups. More faulty machines impact more groups, inducing faster negative propagation across the entire cluster. The data granularity determines the visibility of the propagation process. With a brief duration of less than 100 milliseconds per interaction, the dissimilar pattern may be overlooked due to coarse-grained monitoring. Yet, millisecond-level monitoring is not widely deployed due to the high overhead.

In our production environment, concurrent faulty machine instances only occur due to automatic switch reboots or switch-related AOC errors. A switch reboots in response to high temperatures, extreme port congestion, and so on. Thirty-two connected machines will be forced to go offline out of a total of 600 machines in our environment. However, Minder hardly distinguishes the faulty outliers. Firstly, the fault ratio is relatively high. Given our rail-optimized topology and 3D parallelism mechanism, communication among 32 machines contains at most 256 DP groups, quickly propagating across other machines. Moreover, the current second-level time granularity monitoring limits Minder's ability. The rapid spread is only observable at the millisecond level following several training interactions. However, the rate of such multiple faulty machine instances is less than 1%, sometimes zero a month. Should a switch reboot, the switch monitoring system will automatically alert the engineers.

To demonstrate Minder's ability to detect multiple faulty machines, we injected PCIe downgrading into two of four machines simultaneously, with customized millisecond-level monitoring. Thus, the ratio of faulty machines is higher and the data granularity is finer than the switch-reboot instance. In the experiment, each machine was equipped with eight NVIDIA Ampere GPUs, running Reduce-Scatter collectively. Two PCIe links on two machines were purposely degraded. With the millisecond-level data from the NICs, Minder could detect the two NICs connected to the faulty PCIe links. These two NICs presented the largest outlier distances during Reduce-Scatter. Figure 16 shows the millisecond-level monitoring, where normal NICs demonstrate a high throughput at the beginning of each Reduce-Scatter step to transmit their data to the next node. In contrast, the NICs with downgraded PCIe links exhibit steady and low throughput. Thereafter, normal ones drop to zero, after transmitting their data, waiting for the slow NICs for synchronization. Minder can capture this dissimilarity, thanks to the granularity of milliseconds. Our injection experiments also show Minder's ability to detect other concurrent faults, such as GPU degradation and NIC throughput downgrading.

7 Discussion

In this section, we discuss the lessons we learned during implementation and potential future works.

Minder and other monitoring tools. Large-scale model training involves the cooperative efforts of multiple teams.

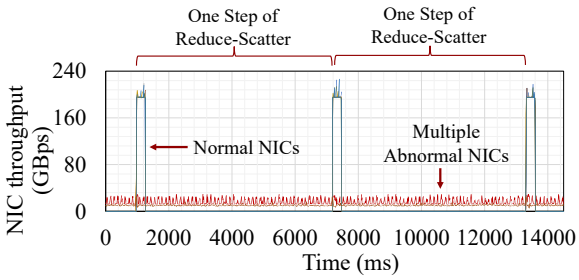


Figure 16: Millisecond-level NIC throughput for all machines after injection of PCIe downgrading on two NICs.

Other monitoring tools used along with Minder include switch state monitoring, periodic heartbeat messages (IP, hardware states, Pod names *etc.*), RDMA traffic down-limit alerts, R-Pingmesh [54] (a pingmesh-like [33] connection testing), and automatic text analysis for GPU error detection. As described in 5, a detected machine will be replaced before restarting the task. These approaches primarily target network connection and jitters or GPU states, whereas Minder provides metrics covering computation, storage, and communication resources. The combined use of them enhances the detection efficacy. Meanwhile, offline testing tools, such as DCGM [1], EUD [3], are used for intra-host bottlenecks diagnosis, though not feasible for run-time fault identification.

Similar systems and tools are introduced. For example, SuperBench [77] is a proactive system to ensure cloud AI infrastructure reliability. It runs model training and component benchmarks to identify incremental performance degradation on defective machines. The first difference is that it works proactively because incremental performance degradation exists due to hardware redundancy. However, such gradual degradation process before a fault occurs is hardly observed in our distributed training. Second, SuperBench mainly improves the hardware-side reliability. Real-time software errors also occur frequently during model training. Besides, SuperBench performs offline validation by running benchmarks. Minder otherwise monitors the tasks throughout their life cycles. Thus, the integration of such a proactive system with Minder is a more promising debugging solution. By executing benchmarks on the machines in a job and monitoring their running status, faults might be largely prevented and rapidly detected.

Minder's Generality. Minder can be extended in data granularity and the spectrum of available metrics. Second-level monitoring is currently deployed and used by Minder. Minder has demonstrated improved detection capability for concurrent faulty machines with millisecond-level monitoring data in 6.6. With finer-grained data, the annihilated rapid propagation from the straggler will be revealed, as a training iteration only lasts tens of milliseconds. Besides, the currently available metrics for Minder are out-of-band hardware counters. Other metrics (*e.g.,* AOC counters) and in-band traces (Torch Profiler [16], the Megatron-LM timer, or CUDA event timer [40]) could also be utilized by Minder. These new traces offer

fine-grained training and collective communication operation information for comprehensive performance monitoring.

Minder's robustness of other faults. Notably, Minder is allowed to detect new or rare fault types not covered in training, as long as the monitoring data presents discernible dissimilarities. For concurrent faulty machines, provided multiple machines present a distinctive dissimilarity, Minder can detect them simultaneously.

Machine-level similarity. Large models such as those in LLM training and multi-modal tasks often employ 3D parallelism. This results in a trend of consistency across machines for computation, communication, and storage, allowing Minder to detect faulty machines. As a result, we focus on machine-level detection instead of finer granularity. We also consider Minder in other workloads, such as large-scale inference and fine-tuning. As long as these workloads satisfy the requirements of inter-machine metric similarity and fault continuity, Minder could be applied. Future work will explore Minder's effectiveness in other workloads.

Not all failed tasks have the right label. Although the labeled machine is the root cause, the Minder-detected machine may also have temporary performance fluctuations. It is necessary to inform engineers of such performance jitters.

Root cause analysis. Minder detects faults at the machine level. The root cause for a fault indicated by a metric is uncertain. Extra labor is employed for further network jitter and short-term straggler analysis. In the future, we plan to design fine-grained run-time monitoring for root cause identification.

8 Related Work

In this section, we briefly introduce related works on anomaly diagnosis [20, 28, 29, 31, 37, 38, 41, 59, 61, 72, 84].

Intra-host diagnosis. Run-time diagnosis detects anomalies without disturbing the running tasks. Leveraging the existing counters [8, 14] from the machine is a direct method [70, 71, 78]. For example, BRCA [63], MonitorRank [43], and FChain [62] construct a dependency graph based on historical monitoring data or traces for anomaly alerts. These DL algorithms have been proven to be automatic, robust, and flexible. Another naive but effective way is to check logs or `dmesg` [2] using natural language processing (NLP)-based methods [19]. However, log-based approaches are limited to their log content and information processing abilities.

Offline diagnosis requires specific tools to detect possible intra-host bottlenecks when the machine is not running tasks. Liu *et al.* [55] and Martinasso *et al.* [58] implement tools to test if there is NVLink [13] or PCIe link [15] degradation or congestion. Collie [44] is a "fuzzing"-like implementation to help uncover potential performance bottlenecks in RNICs. Deepview [82] is designed for virtual hard disk failure localization. These approaches are useful ahead of running tasks, so they cannot be directly used during the large-scale training process.

Inter-host diagnosis. NetBouncer [73] leverages the IP-in-IP technique to actively localize failure devices or links among millions of servers in a data center network. Similarly, SNAP [79] monitors TCP statistics and socket logs for network diagnosis. Pingmesh [33], Haecki [34], and Fathom [66] monitor end-to-end latencies between arbitrary servers or detailed RPC performance in data centers. Nonetheless, they only detect failures along the routes instead of intra-host hardware failures that degrade the training speed. Cloud system diagnosis [26, 35, 47, 81] are based on contextual data patterns and associations of microservices on multiple machines. Such patterns and dependencies are eliminated in distributed training, where machines exhibit similar workloads.

Algorithms for anomaly detection and diagnosis. The first type is statistics-based methods. Apart from Euclidean distance, Pearson Correlation [25], Kendall's tau [42], and Spearman Correlation [80] also quantify the similarity between two vectors and discover the deviating anomalies. Setting parameters as thresholds from experienced operators is usually required [56].

Supervised algorithms are widely used for anomaly detection [45, 53]. EGADS in Yahoo [45] leverages diverse machine learning methods for large-scale univariate time series anomaly detection. Machine learning algorithms like random forest [64] are used for incident routing [27], VM compromise detection [21]. However, supervised learning does not fit in our context, where the goal is to detect the faulty machine instead of classification.

Unsupervised learning [50, 51, 56, 70, 71, 75, 78] identifies outliers from monitoring data for root cause detection and machine state detection, or enables dialogue-based diagnosis chatting. Apart from VAE (4.2), clustering [50, 51, 56] is commonly used to cluster the machines with similar monitoring data change patterns or detect anomalies for time series.

9 Conclusion

This paper presents Minder, addressing the problem of faulty machine detection in distributed training tasks. Minder leverages the concept of similarity among machines and the continuity of a fault during training. Minder has been deployed in our production environment for a year to assist engineers in training diagnosis. Evaluation results demonstrate the reduced time required by Minder and the effectiveness of its design choices for training tasks.

This work does not raise any ethical issues.

Acknowledgments

We thank all the anonymous NSDI reviewers for their feedback that greatly improved the paper. We thank the broader ByteDance High-speed Network team and Applied Machine Learning team for their support throughout this project.

References

[1] DCGM Diagnostics. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/dcgm-diagnostics.html.

[2] dmesg. https://linuxhint.com/dmesg_tutorial/.

[3] Extended Utility Diagnostics (EUD). https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/dcgm-eud.html.

[4] GPU Memory and Duty Cycle. https://cloud.google.com/kubernetes-engine/docs/concepts/gpus.

[5] GPU Metrics. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html#profiling-metrics.

[6] GPU Pricing. https://cloud.google.com/compute/gpus-pricing.

[7] GPU tensor core utility. https://docs.nvidia.com/datacenter/dcgm/latest/user-guide/feature-overview.html#profiling-metrics.

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

[9] NVIDIA ConnectX-6 Dx Network Adapters. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/.

[10] Nvidia DGX-2H. https://docs.nvidia.com/dgx/pdf/dgx2-user-guide.pdf.

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

[12] Nvidia NCCL timeout. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html.

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

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

[15] PCIe. https://docs.nvidia.com/certification-programs/pdf/nvidia-certified-configuration-guide.pdf.

[16] torch profiler. https://pytorch.org/docs/stable/profiler.html.

[17] *Infiniband Trade Association. Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE).* 2014.

[18] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[19] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370, 2006.

[20] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, G. Outhred, and B. T. Loo. Closing the network diagnostics gap with vigil. In *Proceedings of the SIGCOMM Posters and Demos*, pages 40–42. 2017.

[21] B. Arzani, S. Ciraci, S. Saroiu, A. Wolman, J. Stokes, G. Outhred, and L. Diwu. {PrivateEye}: Scalable and {Privacy-Preserving} compromise detection in the cloud. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 797–815, 2020.

[22] H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2):285–297, 1998.

[23] C. Cheadle, M. P. Vawter, W. J. Freed, and K. G. Becker. Analysis of microarray data using z score transformation. *The Journal of molecular diagnostics*, 5(2):73–81, 2003.

[24] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[25] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen. Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4, 2009.

[26] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[27] J. Gao, N. Yaseen, R. MacDavid, F. V. Frujeri, V. Liu, R. Bianchini, R. Aditya, X. Wang, H. Lee, D. Maltz, et al. Scouts: Improving the diagnosis process through domain-customized incident routing. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 253–269, 2020.

[28] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. {SIMON}: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019.

[29] D. Ghita, K. Argyraki, and P. Thiran. Toward accurate and practical network tomography. *ACM SIGOPS Operating Systems Review*, 47(1):22–26, 2013.

[30] H. Ghorbani. Mahalanobis distance and its application for detecting multivariate outliers. *Facta Universitatis, Series: Mathematics and Informatics*, pages 583–595, 2019.

[31] J. Gong, Y. Li, B. Anwer, A. Shaikh, and M. Yu. Microscope: Queue-based performance diagnosis for network functions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 390–403, 2020.

[32] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, 51(5):1–42, 2018.

[33] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[34] R. Haecki, R. N. Mysore, L. Suresh, G. Zellweger, B. Gan, T. Merrifield, S. Banerjee, and T. Roscoe. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 861–877, 2022.

[35] V. Harsh, W. Zhou, S. Ashok, R. N. Mysore, B. Godfrey, and S. Banerjee. Murphy: Performance diagnosis of distributed cloud applications. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 438–451, 2023.

[36] T. He, X. Li, Z. Wang, K. Qian, J. Xu, W. Yu, and J. Zhou. Unicron: Economizing self-healing llm training at scale. *arXiv preprint arXiv:2401.00134*, 2023.

[37] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, et al. Leveraging sdn layering to systematically troubleshoot networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 37–42, 2013.

[38] H. Herodotou, B. Ding, S. Balakrishnan, G. Outhred, and P. Fitter. Scalable near real-time failure localization of data center networks. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1689–1698, 2014.

[39] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, and T. Zhang. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, Santa Clara, CA, Apr. 2024. USENIX Association.

[40] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[41] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. E. Anderson, and A. Krishnamurthy. Reverse traceroute. In *NSDI*, volume 10, pages 219–234, 2010.

[42] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[43] M. Kim, R. Sumbaly, and S. Shah. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):93–104, 2013.

[44] X. Kong, Y. Zhu, H. Zhou, Z. Jiang, J. Ye, C. Guo, and D. Zhuo. Collie: Finding performance anomalies in {RDMA} subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, 2022.

[45] N. Laptev, S. Amizadeh, and I. Flint. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1939–1947, 2015.

[46] C. Leys, O. Klein, Y. Dominicy, and C. Ley. Detecting multivariate outliers: Use a robust variant of the mahalanobis distance. *Journal of experimental social psychology*, 74:150–156, 2018.

[47] L. Li, X. Zhang, S. He, Y. Kang, H. Zhang, M. Ma, Y. Dang, Z. Xu, S. Rajmohan, Q. Lin, et al. Conan: Diagnosing batch failures for cloud systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 138–149. IEEE, 2023.

[48] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.

[49] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[50] X. Li, J. Lin, and L. Zhao. Linear time complexity time series clustering with symbolic pattern forest. In *IJCAI*, pages 2930–2936, 2019.

[51] Z. Li, Y. Zhao, R. Liu, and D. Pei. Robust and rapid clustering of kpis for large-scale anomaly detection. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2018.

[52] S. Lin, R. Clark, R. Birke, S. Schönborn, N. Trigoni, and S. Roberts. Anomaly detection for time series using vae-lstm hybrid model. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4322–4326. Ieee, 2020.

[53] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng. Opprentice: Towards practical and automatic anomaly detection through machine learning. In *Proceedings of the 2015 internet measurement conference*, pages 211–224, 2015.

[54] K. Liu, Z. Jiang, J. Zhang, S. Guo, X. Zhang, Y. Bai, Y. Dong, F. Luo, Z. Zhang, L. Wang, X. Shi, H. Xu, Y. Bai, D. Song, H. Wei, B. Li, Y. Pan, T. Pan, and T. Huang. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 554–567. Association for Computing Machinery, 2024.

[55] K. Liu, Z. Jiang, J. Zhang, H. Wei, X. Zhong, L. Tan, T. Pan, and T. Huang. Hostping: Diagnosing intra-host network bottlenecks in {RDMA} servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 15–29, 2023.

[56] P. Liu, Y. Chen, X. Nie, J. Zhu, S. Zhang, K. Sui, M. Zhang, and D. Pei. Fluxrank: A widely-deployable framework to automatically localizing root cause machines for software service failure mitigation. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 35–46. IEEE, 2019.

[57] P. C. Mahalanobis. On the generalized distance in statistics. *Sankhyā: The Indian Journal of Statistics, Series A (2008-)*, 80:S1–S7, 2018.

[58] M. Martinasso, G. Kwasniewski, S. R. Alam, T. C. Schulthess, and T. Hoefler. A pcie congestion-aware performance model for densely populated accelerator servers. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 739–749. IEEE, 2016.

[59] D. S. Matteson and N. A. James. A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505):334–345, 2014.

[60] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

[61] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese. Gestalt: Fast,{Unified} fault localization for networked systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 255–267, 2014.

[62] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward black-box online fault localization for cloud systems. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 21–30. IEEE, 2013.

[63] X. Nie, Y. Zhao, K. Sui, D. Pei, Y. Chen, and X. Qu. Mining causality graph for automatic web-based service diagnosis. In *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2016.

[64] A. Palczewska, J. Palczewski, R. M. Robinson, and D. Neagu. Interpreting random forest models using a feature contribution method. In *2013 IEEE 14th International Conference on Information Reuse & Integration (IRI)*, pages 112–119. IEEE, 2013.

[65] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[66] M. A. Qureshi, J. Yan, Y. Cheng, S. H. Yeganeh, Y. Seung, N. Cardwell, W. De Bruijn, V. Jacobson, J. Kaur, D. Wetherall, et al. Fathom: Understanding datacenter application network performance. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 394–405, 2023.

[67] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[68] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[69] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[70] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2828–2837, 2019.

[71] M. Sun, Y. Su, S. Zhang, Y. Cao, Y. Liu, D. Pei, W. Wu, Y. Zhang, X. Liu, and J. Tang. Ctf: Anomaly detection in high-dimensional time series with coarse-to-fine model transfer. In *IEEE INFOCOM 2021-IEEE conference on computer communications*, pages 1–10. IEEE, 2021.

[72] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with {PathDump}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, 2016.

[73] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. {NetBouncer}: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[74] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[75] H. Wang, A. Abhashkumar, C. Lin, T. Zhang, X. Gu, N. Ma, C. Wu, S. Liu, W. Zhou, Y. Dong, W. Jiang, and Y. Wang. NetAssistant: Dialogue based network diagnosis in data center networks. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 2011–2024, Santa Clara, CA, Apr. 2024. USENIX Association.

[76] D. J. Weller-Fahy, B. J. Borghetti, and A. A. Sodemann. A survey of distance and similarity measures used within network intrusion anomaly detection. *IEEE Communications Surveys & Tutorials*, 17(1):70–91, 2014.

[77] Y. Xiong, Y. Jiang, Z. Yang, L. Qu, G. Zhao, S. Liu, D. Zhong, B. Pinzur, J. Zhang, Y. Wang, J. Jose, H. Pourreza, J. Baxter, K. Datta, P. Ram, L. Melton, J. Chau, P. Cheng, Y. Xiong, and L. Zhou. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 835–850, Santa Clara, CA, 2024. USENIX Association.

[78] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, et al. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 world wide web conference*, pages 187–196, 2018.

[79] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[80] J. H. Zar. Spearman rank correlation. *Encyclopedia of biostatistics*, 7, 2005.

[81] Z. Zeng, Y. Zhang, Y. Xu, M. Ma, B. Qiao, W. Zou, Q. Chen, M. Zhang, X. Zhang, H. Zhang, et al. Traceark: Towards actionable performance anomaly alerting for online service systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 258–269. IEEE, 2023.

[82] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for azure. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 519–532, 2018.

[83] P. Zhao, M. Kurihara, J. Tanaka, T. Noda, S. Chikuma, and T. Suzuki. Advanced correlation-based anomaly detection method for predictive maintenance. In *2017 IEEE International Conference on Prognostics and Health Management (ICPHM)*, pages 78–83. IEEE, 2017.

[84] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

Table 2: Monitoring metrics collected by Minder.

Monitoring Metrics	Description
CPU Usage	Percentage of CPU time being used.
PFC Tx Packet Rate	Periodic counts of PFC packets sent by RDMA-enabled devices.
Memory Usage	Percentage of memory being used.
Disk Usage	Percentage of storage space being used on a disk.
TCP Throughput	Periodic counts of the amount of TCP data being transmitted by a NIC.
TCP+RDMA Throughput	Periodic counts of the amount of TCP and RDMA data being transmitted by an NIC.
GPU Memory Used [4]	The amount of GPU memory being used by processes.
GPU Duty Cycle [4]	Percentage of time over the past sample period when the accelerator is active.
GPU Power Draw	Periodic counts of the GPU power consumption.
GPU Temperature	The temperature of a GPU while it is operating, measured in degrees Celsius.
GPU SM Activity [5]	Averaged percentage of time when at least one warp is active on a multiprocessor.
GPU Clocks	The clock speed of a GPU, reflecting the frequency of the GPU's processor.
GPU Tensor Core Activity [5]	Percentage of cycles when the tensor (HMMA / IMMA) pipe is active.
GPU Graphics Engine Activity [5]	Percentage of time when any portion of the graphics or compute engines are active.
GPU FP Engine Activity [5]	Percentage of cycles when the FP pipe is active.
GPU Memory Bandwidth Utilization [5]	Percentage of cycles when data is sent to or received from the device memory.
PCIe Bandwidth [5]	The rate of data transmitted/received over the PCIe bus.
PCIe Usage [5]	Percentage of the bandwidth being used on the PCIe bus.
GPU NVLink Bandwidth [5]	The rate of data transmitted/received over an NVLink.
ECN Packet Rate	Periodic counts of ECN packets transmitted/received by a NIC.
CNP Packet Rate	Periodic counts of CNP packets transmitted/received by a NIC.

Appendices are supporting material that has not been peer-reviewed.

A Fault Types

The fault types are listed in Table 1.ECC error: caused by corrupted or lost data in (GPU) memory. PCIe downgrading: a link fault leading to a slow PCIe sending/receiving rate. NIC dropout: a NIC is missing from the OS. GPU Card drop: a disconnected GPU card. NVLink error: a link fault between two Nvidia GPUs. AOC error: an error in high-speed active optical cables (AOC) on either the host network card or the switch side. CUDA execution error: an unexpected overflow or configuration leading to a failed CUDA program. GPU execution error: unexpected page-fault, out-of-memory, and other incorrect processing leading to GPU hang or other results. HDFS error: HDFS connection timeout, io error, and so on when loading or saving checkpoints. Machine unreachable: mostly due to malfunctioning SSH services or virtual machine services. Others: illegal memory access, failed scheduling, no disk storage, low resource usage, switch reboot, and so on.

B Collected Monitoring Metrics

Table 2 contains the monitoring metrics that we choose to collect in our production environment, though only a portion of them are used for training and detection. Other available host metrics could also be used by Minder.

Holmes: Localizing Irregularities in LLM Training with Mega-scale GPU Clusters

Zhiyi Yao[1,2*], Pengbo Hu[1,2*], Congcong Miao[2*], Xuya Jia[2], Zuning Liang[1], Yuedong Xu[1§], Chunzhi He[2], Hao Lu[2], Mingzhuo Chen[2], Xiang Li [2], Zekun He[2], Yachen Wang [2], Xianneng Zou[2], Juncheng Jiang[3]
[1]*Fudan University,* [2]*Tencent,* [3]*University of Chicago*

Abstract

Training Large Language Models (LLMs) on large-scale GPU clusters requires numerous iterations over several months. Existing works mainly focus on addressing failures that interrupt the iterative training process to improve the utilization of GPU clusters. However, our large-scale measurements over tens of thousands of GPUs show that the training process exhibits an unstable state with some *irregular* iterations taking even more than twice the time of a normal iteration. Surprisingly, we find that these irregular iterations greatly extend the time of LLM training, which is even more severe than the impact of failures. Meanwhile, the irregular phenomenon is silent, making it challenging to be accurately localized. In this paper, we propose a first-of-its-kind system called Holmes, leveraging communication operators to accurately localize these irregularities in real-time. The core of Holmes's approach is to employ an enhanced abnormal operator detection model and a novel communication operator graph to perform efficient irregularity localization. Furthermore, Holmes conducts cross-iteration analysis to improve localization accuracy. We evaluate Holmes using large-scale trace-driven simulations and a production-level prototype. Large-scale simulation results demonstrate that Holmes achieves irregularity localization accuracy of 97.21%. Production-level prototype evaluation results show Holmes can localize irregularity within 30.3 seconds, achieving a speedup of 6.52× as compared to traditional approaches.

1 Introduction

Large language models (LLMs) [7] are propelling the rapid evolution of efficient and reliable AI infrastructure. Since the model size explosively increased to hundreds of billions of parameters such as GPT-4 [42] and PaLM [10], the training clusters have scaled out to encompass thousands to even 10,000 GPUs [25]. Training LLMs over large-scale GPU clusters always requires hundreds of thousands of iterations. Each *iteration* is a cycle of parameter update and synchronization, consisting of a mixture of massive compute- and communication- intensive operators. Executing these iterations in the GPU cluster can last for months, making investment in LLM training entail significant costs. Efficient utilization of expensive GPUs is crucial since even small improvements can save millions of dollars at such a scale [5].

Unfortunately, LLM training is *fragile* in the sense that even a small failure in the GPU clusters can explicitly interrupt the training process. These failures will finally extend the training time, resulting in a considerable waste of hardware resources. Therefore, there are several works focusing on failure detection and diagnosis to improve the system reliability [18, 21, 23, 32, 35, 43, 54, 62]. In these works, once a failure is diagnosed, the anomalous device is isolated to recover the training process.

Generally, the time of each iteration in the LLM training is expected to be stable and regular because the computing and communication patterns are predetermined and repeated in each iteration. However, our production measurements over tens of thousands of GPUs indicate that the training process exhibits an unstable state with the time of each iteration varying. Some *irregular* iterations take even more than twice the time as a normal iteration. Here, the term "*irregularity*" describes the time of an iteration exceeding the reference value by a certain margin, yet without interrupting the training process. We then accumulate additional training time introduced by irregularity. Surprisingly, we observe that the additional time introduced by the irregularity is even more severe than the impact of failures. Our manual post hoc analyses show that the root causes of these irregularities encompass anomalous GPUs, network link issues, unstable switches, etc. (§ 3.1).

However, efficient irregularity localization in a large GPU cluster is very challenging. Firstly, these irregularities are *silent* and do not interrupt the training process, making traditional failure diagnosis approaches [21, 43, 54, 62] relying on error logs ineffective. As there will be no failure logs during the irregular iteration, traditional systems will even not react

*The authors contributed equally to the paper.
§Corresponding author: Yuedong Xu (ydxu@fudan.edu.cn).

to begin the diagnosis process. Secondly, the anomaly in the device causing longer training iteration will propagate to impact other GPUs. Specifically, since there are thousands of collective communications across GPUs in the cluster, once an anomaly occurs in a device, all the GPUs connected to this device will be impacted to exhibit a longer training time in one iteration. When there are a lot of irregular iteration logs among these GPUs, it is hard to identify the anomalous device. Thirdly, the irregularity localization approach should be accurate. When the normal device is mistakenly identified as an anomaly, the isolation process of the device will lead to the waste of precious computing resources.

Holmes architecture. To address the aforementioned issues, we present a first-of-its-kind system, called Holmes, to accurately localize these irregularities for LLM training on mega-scale GPU clusters. Holmes is a lightweight approach that only relies on the data from the communication operator, thereby introducing little overhead during the LLM training. Specifically, to explicitly capture the silent irregularity, we introduce an enhanced abnormal operator detection model. We first leverage a machine learning method (i.e., random forest) to analyze communication tracing logs to detect abnormal communication operators. As the time distribution of iteration may shift over a long-term training process, we further introduce an auto-tuning approach for the machine learning method to adapt to the changes (§ 4.2). To identify the anomalous device using a set of irregular iteration logs from GPUs, we build a novel communication operator graph and design a combination of breadth-first and depth-first search methods to localize the irregularity in the graph. Note that Holmes only collects logs from these GPUs which are traversed by the search methods to reduce collected data volume and localization time (§ 4.3). To further improve the localization accuracy, we conduct cross-iteration analysis among all the devices including GPUs, links, and switches in the training clusters, and utilize Maximum A Posteriori estimation to find the most possible anomalous devices (§ 4.4).

Contributions. Our main contributions are summarized as follows:

- We identify the irregularity phenomenon and observe that these irregularities greatly extend the time of LLM training, which is even more severe than the impact of failures. To the best of our knowledge, we are the first to study the irregularity phenomenon in large-scale LLM training (§ 3).
- We design Holmes that leverages communication operators to localize irregularity over a 10,000 GPU cluster in real-time. Holmes constructs a communication operator graph combined with a domain-specific search method (§ 4).
- Large-scale simulation results demonstrate that Holmes achieves an overall localization accuracy of 97.21%. Production-level prototype evaluation results show that Holmes can localize irregularity within 30.3 seconds, which is 6.52× faster than the baseline (§ 5).

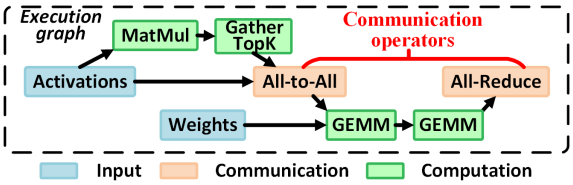


Figure 1: Compute- and communication- operators.

2 Background

Training Large Language Models (LLMs) relies on a large number of GPU machines and auxiliary resources to accelerate the execution of hundreds of thousands of iterations [21,60]. Each iteration is a cycle of parameter update and synchronization, consisting of a mixture of massive compute- and communication- intensive operators. These operators are repetitively executed following a fixed order, which can be represented in the form of an execution graph. Figure 1 depicts an example execution graph of a typical LLM training, where `Activations` and `Weights` are the input for the operators. The computation operators such as `MatMul` are executed on a single GPU without interference with other training nodes. In contrast, communication operators such as `All-to-All` involve a set of GPUs and will be executed only after all the GPUs have completed the execution of their respective dependent operators. Each GPU has its own execution graph and executes independently, and the communication operators will conduct synchronization among GPUs.

Communication operators. The communication operators serve for various parallel training strategies, which aim to better utilize GPUs and accelerate the training. These communication operators can be classified into two categories [37]: collective communication (CC) and point-to-point (P2P) communication. Collective communication including `All-Reduce`, `All-Gather`, `All-to-All` and `Reduce-Scatter`, is widely used in strategies like data parallelism (DP) [11,46], tensor parallelism (TP) [41,51] and expert parallelism (EP) [14,28]. During the training process, collective communication brings tight synchronization to massive GPUs involved, as it forces all the GPUs to initiate data transmission simultaneously. Different from collective communication, P2P communications like `Send` and `Recv`, do not need synchronization among massive GPUs. They are adopted in the pipeline parallelism (PP) [22,40], used to transmit the activations between the sub-models. Figure 2 depicts an example of LLM distributed training. The black arrows represent the P2P communication of GPUs, which communication only involves one sender and one receiver. Bidirectional arrows are collective communication, in which GPUs involved transmit data to each other.

3 Observation & Motivation

In this section, we first describe the overall performance degradation caused by irregularities in our production cluster and classify them into three categories (§ 3.1). We then elaborate

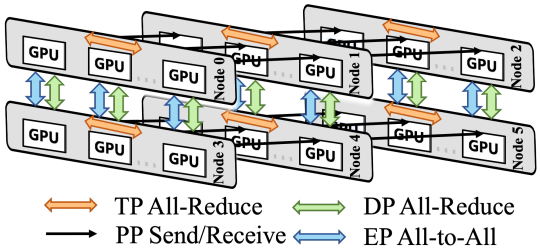


Figure 2: Communication of LLM distributed training.

on the challenges of localizing the anomalous devices, which will cause the irregularity (§ 3.2).

3.1 Irregularity in LLM Training

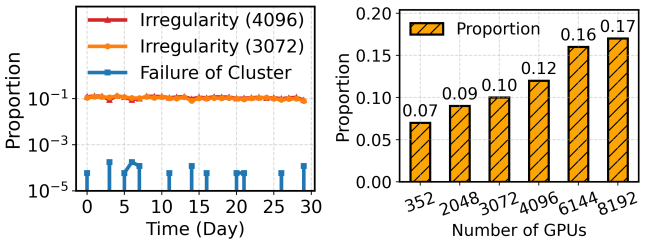
We define a novel metric to measure the irregularity in LLM training as below.

Definition 1 δ−*Irregularity refers to the phenomenon where training continues* ***uninterrupted****, but the duration of an iteration, i.e. time of every training step, is* δ *times longer than the average value within a given time window.*

The parameter δ (where $\delta > 1$) is tunable to account for subjective assessments of performance irregularities (by default δ is set to 1.1). The production cluster in our study comprises over 10,000 NVIDIA H800 GPUs. We keep tracing LLM training using a total number of GPUs ranging from 352 to 8192 in three months. The models of LLM training include LLaMA [56], LLaMA2 [57] and GPT3 [7] which vary in size. Their parallel training strategies employ hybrid parallelism incorporating tensor- [41], data- [11, 46], pipeline- [22, 40], sequence- [27], and expert- parallelism [28].

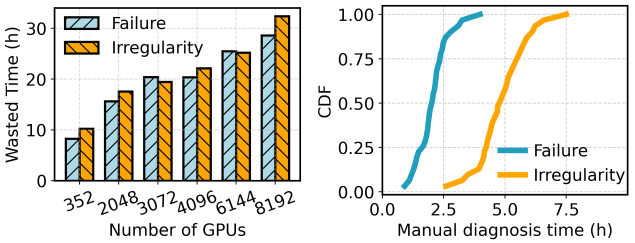
Irregularity occurs frequently. In Figure 3(a), we compare the numbers of irregularities and failures during the training of LLaMA+MoE using 3072 and 4096 GPUs for a month, respectively. The frequency is defined as the irregular iterations compared to the total iterations per day. On average, irregularities occur in 778 and 1047 iterations daily, in contrast to 6646 and 10165 total iterations per day. No more than three failures occur in one day throughout the entire training process, and only 40% of the days in a month encounter failure with a failure probability of less than 10^{-3}. It is evident that the frequency of failures is several orders of magnitude lower than that of irregularities. In Figure 3(b), we perform a statistical analysis of the frequencies of irregularities across various training. The frequency of irregularities increases as more GPUs are used, since more devices in training can lead to more potentially anomalous devices. Specifically, when training with 8192 GPUs, the irregularity frequency is 0.17, nearly three times that of training with 352 GPUs. More details and analysis of Figure 3(b) can be referred to appendix A.

Irregularity extends the training time. We highlight the *time wastage* caused by potential irregularities compared to training failures for a month in Figure 4(a). The wasted time due to failures is defined as the time interval between the



(a) Frequency across time (b) Frequency across training

Figure 3: Frequency of irregularity.



(a) Wasted time of irregularity (b) CDF of irregularity diagnosis time

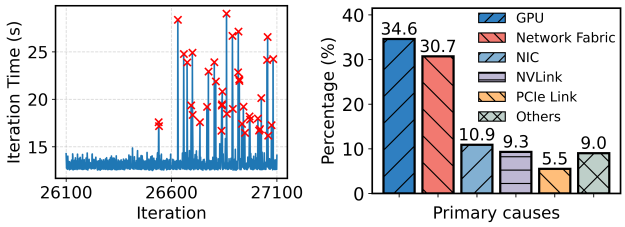
Figure 4: Impact of irregularity.

pause and resumption of training. As reported in [21], a training failure state usually lasts no more than three hours. The wasted time due to irregularities is calculated as the difference between the total actual training time and the hypothetical training time, where each irregular iteration is replaced with the time it would have taken without irregularities. An observation is that the time wastage of irregularities is comparable to or even greater than that of failures, with wasted time increasing as the number of GPUs scales up. For instance, when training with 8192 GPUs, the wasted time due to irregularities is as long as 32.38 hours, leading to a significant wastage of valuable GPU resources. This motivates us to explore the key behaviors of irregularities and identify the anomalous devices that induce such irregularities.

Irregularity localization is time-consuming. We then compare the manual irregularity localization time with the failure diagnosis time. As shown in Figure 4(b), around 86.2% of the irregularity localization time is larger than 4 hours. This is much longer than most of the time consumption on failure diagnosis, where approximately 84.7% of the diagnosis time is within 2.5 hours. This is attributed to that failure diagnosis has been extensively studied and standardized. In contrast, irregularity localization highly depends on human experience, leading to uncertainties in the time required for resolution. This motivates us to build an efficient system for automatic irregularity localization.

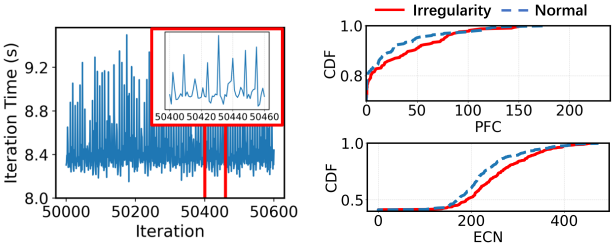
3.1.1 Abnormal Spikes

In this paper, we classify the irregularities into three categories according to their duration and frequency. The first type of irregularity, namely *abnormal spike*, occurs randomly and less frequently but significantly extends per-iteration training time. We study the pattern of abnormal spikes while training



(a) Iterations with abnormal spike (b) Causes of abnormal spike

Figure 5: Irregularity: abnormal spike.



(a) Iterations with fluctuation (b) CDF of PFC and ECN

Figure 6: Irregularity: performance fluctuation.

a LLaMA+MoE model using 3072 GPUs. The per-iteration times are shown in Figure 5(a), with a total of 1000 iterations recorded. Within this time window, abnormal spikes, marked with crosses, occur in 39 iterations. The iteration time with a spike is much longer than the normal value. For instance, at the 26803-rd iteration, the training time is 1.96× longer than the average. Over the entire training process for a month, the wasted time due to spikes amounts to 8.93 hours.

Manual diagnosis. When a *serious* abnormal spike occurs, our engineers will examine the states of training resources such as GPU and CPU utilization, ECN marks. Through comparison with empirical normal values, the suspicious anomalous devices will be isolated with further task-specific stress tests. In general, the entire process takes nearly 3-7 hours to resolve the abnormal spikes, while they continue to occur randomly until being resolved. We illustrate the primary causes of abnormal spikes and their respective percentages in Figure 5(b). Among them, GPU execution ranks first and is responsible for 34.6% of all abnormal spikes. The second major cause is the inter-machine networking issue, comprising network fabric problems, link flapping, network link failures, and NIC anomalies. Together, they account for 41.6% of the spikes, highlighting the severity of networking faults. This suggests that monitoring inter-machine transmission could be the key to identifying these faults.

3.1.2 Performance Fluctuation

Another type of irregularity is called *performance fluctuation* that is usually ignored due to their inconspicuous magnitudes of iteration time changes. Figure 6(a) shows a sequence of 600 iteration times of training with 4096 GPUs, where δ in this case is set to 1.05. The iteration times at the top may reach 9.45 seconds and those at the bottom are around 8.18 seconds, and the average iteration time is 8.40 seconds. As a result, there are 85 iterations with abnormal fluctuation among total 600 iterations. Even though the impact of performance fluctuation is smaller than abnormal spike on an individual iteration, the former is much more frequent than the latter (see appendix B). Overall, the training in Figure 6(a) is extended by performance fluctuation for 9.32 hours over a month, which is even more severe than abnormal spike. We also study the performance of network fabric when irregularities occur. Figure 6(b) shows the CDF of packets with ECN mark and pause frames within an iteration on all the ToR switches. The irregular iteration shows a clear shift to more ECN marks and PFCs, compared to the normal iteration. This phenomenon indicates that irregularities could affect the network traffic, which also inspires us to monitor irregularities through the network.

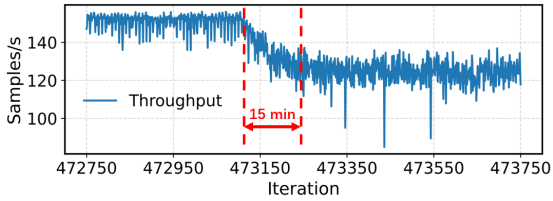


Figure 7: Irregularity: persistent degradation.

Manual diagnosis. The issues that could lead to fluctuations include link flapping and shared network fabrics for multiple jobs (e.g., periodic fluctuation occurs when jobs with different iteration times share spine or ToR switches [45]). Other causes that may also make the training process unstable with small and frequent fluctuations include load imbalance issues of MoE training [20, 31] and congestion [17]. However, this kind of analysis highly relies on the experience of engineers, leading to high and uncertain analytic costs.

3.1.3 Persistent Degradation

Different from the above irregularities, the third type of irregularity degrades the training efficiency persistently, which we call *persistent degradation*. Figure 7 shows the training throughput of 1000 iterations within a 4096 GPUs training. An observation is that the throughput decreases dramatically from around 150 samples per second to 125 samples per second, which means a persistent degradation occurs and results in a 16.72% training throughput degradation. Although the decreasing process of throughput lasts for only 15 minutes, the training will execute at a low efficiency if the irregularity is not addressed in time.

Manual diagnosis. Most of the root causes of persistent degradation are training-independent, such as GPU overheating [21] and abnormal CPU occupation of back-end services. Engineers have to analyze massive resource states and even operation logs of non-training systems to find the causes. For example, after several hours of analysis for the irregularity in Figure 7, we find that the CPU utilization in part of the machines is abnormally high. By tracking the operation logs,

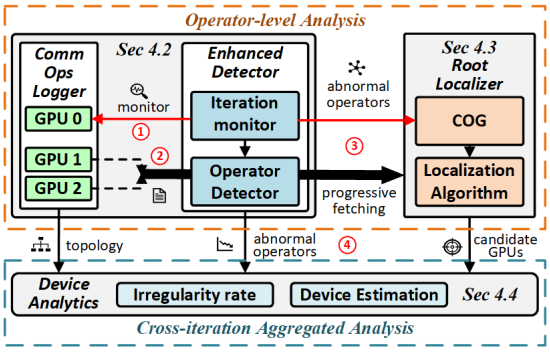


Figure 8: System overview.

the final root cause we identified is that the latest release of our network monitor system frequently scans all the NCCL ports abnormally, competing for the CPU cores. In our training cluster, each persistent degradation could result in 18.2% of training throughput decrease on average.

3.2 Challenges

Our goal is to design a system to accurately localize the above irregularities for LLM training. However, designing such a system in large GPU clusters is quite challenging.

Irregularity behaves silently. Since irregularities do not interrupt training, no error logs are generated. Network monitoring systems track real-time link throughput, which does not reflect the computing anomalies and the traffic interdependency across machines. GPU monitoring tools provide only coarse and noisy runtime states that limit their range of application. The training debugging system usually logs the execution process, while the logging granularity remains unresolved. For example, coarsely logging the iteration time [41,47] is insufficient. Fine-grained logging such as the torch profiler [3] may assist in localization. However, they record all the operators, which considerably slows down the training process [25].

Irregularities's impact propagates among GPUs. The irregularity tends to *propagate* its impact. For instance, if one GPU computes slowly before a DP `All-Reduce`, the other GPUs participating in the same `All-Reduce` must wait for it. As a result, similar irregular phenomena, i.e. extended iteration time, are observed across multiple GPUs, which can be confusing to diagnose. The irregularity can propagate *more than one hop*, further complicating its localization. Independently analyzing all the GPUs will introduce a lot of effort and is time-consuming since a lot of irregular training logs will be detected and hard to localize the anomalous device.

Misidentify the anomalous device is costly. The irregularity localization needs to be accurate, as isolating potentially anomalous devices is a costly countermeasure. While the irregularity occurs frequently, misidentifying anomalous devices causes the waste of precious computing resources, and leaves the irregularity unresolved. It is crucial to ensure high system accuracy while minimizing misidentification.

4 System Design

In this section, we first introduce the design principle and overall design of Holmes (§ 4.1). Then, we elaborate on the three key aspects of Holmes: enhanced abnormal operators detection model (§ 4.2), graph-based root causes localization (§ 4.3), and cross-iteration analysis (§ 4.4).

4.1 Overall Design

The goal of this paper is to accurately localize the anomalous devices that lead to the irregularity during LLM training.

Design principles. Our system design should follow these basic principles.

• *Accuracy.* High accuracy in pinpointing the irregularities and diagnose the root causes.

• *Real-time detection.* Capable of detecting and localizing irregularity in real time.

• *Low overhead.* Acceptable overhead brought by monitoring and localization, which has little impact on the LLM training.

• *Flexibility.* Capable of adapting to various parallel training strategies without any modifications.

System Overview. Following the above design goals, we design Holmes, a first-of-its-kind system to accurately localize the anomalous devices in a large-scale GPU cluster. Holmes is a lightweight approach that only relies on data from communication operators. More concretely, to explicitly observe silent irregularity, we propose an enhanced abnormal operator detection model. This model firstly leverages a communication operator logger to log the detailed process of communication operators (§ 4.2.1). Upon observing the irregularity, we propose an enhanced random forest model (RF) to analyze communication operator logs and capture the specific operator that causes the irregularity. Since the time distribution of iteration may shift over a long-term training and result in the RF performance degradation, we further introduce an auto-tuning approach to adapt to the changes (§ 4.2.2). To localize the anomalous device according to abnormal operators among GPUs, we construct a novel communication operator graph (COG), and design an effective algorithm that combines breadth-first search and depth-first search for traversing the collective communication and point-to-point operators respectively. Holmes only collects logs from the GPUs traversed by the search algorithm, minimizing the amount of data transmitted and localization time (§ 4.3). To ensure high localization accuracy, we introduce cross-iteration analysis among all devices within the training clusters, including GPUs, links, and switches, and employ Maximum A Posteriori estimation to identify the most possible anomalous device (§ 4.4).

Workflow. As shown in Figure 8, Holmes firstly embeds a communication operator logger for each GPU in the training cluster to record the data locally. Then we deploy our system modules including enhanced detector, root localizer and device analytics in an independent CPU server. ① The iteration

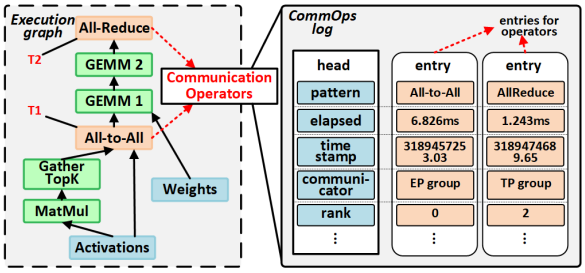
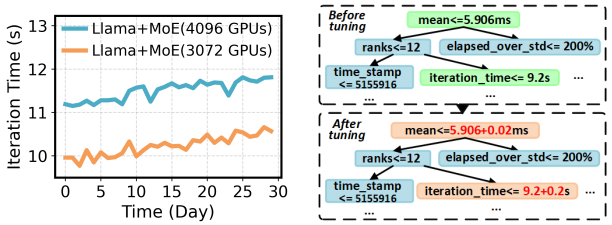


Figure 9: Communication operators and logging scheme.



(a) Distribution drift (b) Auto-tuning against mean shift

Figure 10: (a) Distribution drift and (b) Holmes's auto-tuning.

monitor keeps tracing the log of one pivot GPU for analysis in real time. ② Once the iteration monitor observes an irregularity, the operator detector will perform enhanced abnormal operator detection on the logs. ③ Then root localizer module searches the GPUs in COG and collects the log data of these GPUs that are correlated to the irregularity. ④ The device analytics module calculates the probability of anomaly for every device and identifies the top-K anomalous devices. Finally, the on-call technicians can further confirm and replace the anomalous devices.

4.2 Abnormal Operators Detection

4.2.1 Communication Operator Tracing

Since the irregularity does not generate any error logs, we are motivated to implement a logger to capture this phenomenon. However, logging all information during LLM training can significantly interfere with the training process. Therefore, it is challenging to develop a lightweight approach with minimal overhead on LLM training.

CommOps log. Figure 9 shows a typical execution graph of a single iteration when training LLMs. There are two communications, i.e., `All-to-All` ending at T_1 and `All-Reduce` starting at T_2. The computation operators are executed during T_1 and T_2. We find that both the computation issues and communication issues can be observed by monitoring communication operators. For instance, if the `GEMM` operator executes abnormally with longer computation time, the start time of the `All-Reduce` operator will be later than T_2. Even when computation and communication overlap during training, interdependent communication operators remain vulnerable to cascading delays. Furthermore, communication operators indicate data transmission among GPUs, which we can use to trace the propagation of irregularity's impact. However, popular communication libraries such as NCCL [1] do not provide the interface to record the communication operators directly. In Holmes, we introduce a logging scheme called CommOps log to keep tracing the communication operators as shown in the right part of Figure 9. Each entry of the log records the features of a specific communication operator, which includes the communication pattern, the elapsed time, time stamps, communicator, GPU rank, etc. Since the number of communication operators is much less than that of computation operators in the typical LLM training, CommOps log could trace the fine-grained training process with almost negligible overhead.

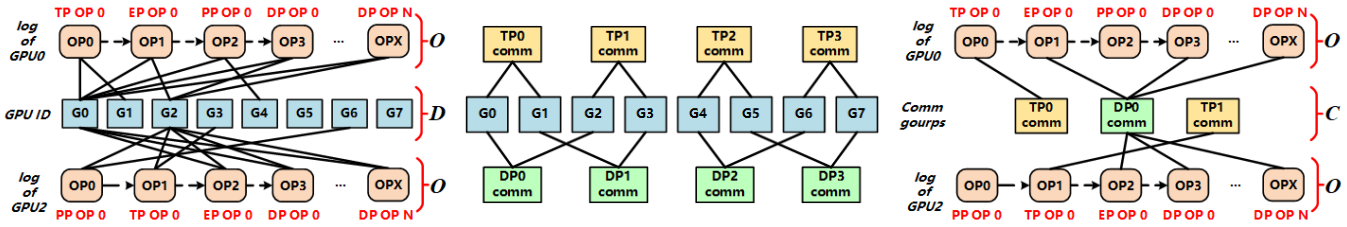
4.2.2 Automatic Operator Detection

Iteration monitor. As we present in § 3.1, the irregularities have apparent characteristics in end-to-end measurement, e.g. prolonged iteration time. For simplicity, we perform irregular iteration detection strictly according to our definition of δ-irregularity.

Unified operator detection. Typically, an iteration always contains tens of thousands of operators. Once an irregular iteration is detected, we need to trace the relevant operators. However, due to the operators' execution time varying dramatically, we can hardly leverage traditional anomaly detection methods such as the 3-sigma rule to detect them. For example, if the duration of an abnormal iteration increases from 7 seconds to 9 seconds, potentially caused by two abnormal operators. The elapsed time of one operator increases from 0.02 ms to 1 ms (by 50×), while the elapsed time of the other operator increases from 1000 ms to 3000 ms (by 3×). In this case, if we only consider the increasing ratio, the former operator will be detected as an abnormal operator rather than the latter. In Holmes, we choose to use Random Forest (RF) to train a prediction model for abnormal operator detection. We consider each entry in the operator log as an individual sample of the RF, while the features are the records of the entry. Since RF can effectively capture multiple features, we also incorporate global information, such as iteration time and average operator elapsed time within a time window into the features. In this way, Holmes can detect abnormal operators with the consideration of the importance and variation of operators.

Detection features. Each operator has specific features in a given iteration, however, detection requires analysis of global information. Below is a list of features we construct:

- *Mean and standard deviation (std).* The average value and standard deviation of operator execution time.
- *Z-Score.* For each point X in the time window, Z-Score is calculated as: $Z = (X - mean)/std$.
- *Quartiles.* The first quartile ($Q1$), the second quartile ($Q2$, i.e., median), and the third quartile ($Q3$) of feature.
- *IQR.* The interquartile range (IQR), calculated as $Q3 - Q1$.
- *Fixed features.* Ranks, data size, communication pattern.



(a) Original COG, (b) Communication grouping. (c) Reconstructed COG.

Figure 11: Communication operator graph.

Distribution drift adaptation. The RF can predict the abnormal operators based on training samples. However, due to the non-stationarity inherent in production, the trained RF model is prone to *distribution drift* [13, 19, 63], which subsequently degrades its forecasting performance over time. For example, we depict the average iteration times per day of two LLM training instances in Figure 10(a). An observation reveals that the average iteration time exhibits slight daily variations. Both iteration times increased by 5.92% and 7.03% respectively within a month, indicating a drift in the distribution of features (e.g., iteration time). Periodic tree model retraining mitigates drift, but RF training requires costly manual data labeling. Holmes counters this drift by enhancing tree-based predictions with an automatic tuning approach. In tree-based models such as RF, there are numerous interconnected tree nodes. Each node is capable of classifying data samples by utilizing trained attribute values associated with a specific feature. We directly tune the attribute values in the trained model periodically. For instance, as shown in Figure 10(b), a trained RF model contains nodes with specific features. We detect that the mean of iteration time right shifts for 0.2 seconds (from 9.0 seconds to 9.2 seconds). To cope with this mean shift, we adjust all attribute values associated with the shifted features, e.g. adding 0.2 seconds to the attribute value 9.0s in the node "iteration_time". This tuning approach allows Holmes to adapt to distribution drift during training without the need for additional efforts such as incremental training.

4.3 Graph-based Causes Localization

During the training, an anomaly in the device causing longer training iteration will propagate to impact other GPUs, appearing longer iteration time. When there are a lot of irregular iteration logs among these GPUs, it is challenging to identify the anomalous device. To address this, Holmes constructs a novel communication operator graph (COG), and then designs an effective algorithm to localize the irregularity.

4.3.1 Communication Operator Graph

Constructing the COG. Holmes designs a novel graph called the communication operator graph (COG) $G = (O, D, E)$, where O and D are the sets of operators and GPUs respectively, and E denotes the edges between O and D. Specifically, Figure 11(a) illustrates each $o \in O$ represents a communication operator in a log and each $g \in D$ represents an individual GPU. An edge $e = (o, g) \in E$ exists between o and g if g is participated in o. In addition, the operators in the same log are connected in the form of a linked list to represent the execution order, as the dash lines in Figure 11(a) show. However, this naive format of COG may result in an unacceptable complexity. For example, if an LLM training includes 4096 GPUs ($|D| = 4096$) and 4000 operators in each log ($|O| = 4000 * 4096$), the number of connections $|E| = |O| * |D|$ can be 6.71×10^{10}, which is prohibitively complicated.

Reconstructing the COG using communication grouping. To address the curse of dimensionality, Holmes leverages the grouping feature of communication operators to reduce the connections. In Figure 11(b), we take 8 GPUs training as an example, where the degrees of data, pipeline, and tensor parallelism are 2. With the communication pattern defined, all the communication operators are executed on fixed groups of GPUs. For instance, all the communication of data parallelism in GPU G0 are executed with G2, then G0 and G2 are called a DP group (labeled as "DP0 comm"). This group conception is used in various parallelism implementations and communication libraries, such as the *communicator* in NCCL [1]. In Holmes, we maintain a mapping of groups such as "DP comm" and their corresponding GPU sets. By substituting GPUs with groups, we reconstruct COG $\hat{G} = (O, C, E)$, where C represents the set of GPU groups. As shown in Figure 11(c)[1], the number of edges in the $\hat{G}$ is much lower than in G. Furthermore, since all the connections of O are sparse (only one to C), Holmes could use sparse storage for the edges, reducing the memory and processing overhead.

4.3.2 Graph-based Localization Algorithm

To localize the irregularity, we design a recursive algorithm (Algorithm 1) to identify the GPU or GPU sets where the root cause occurs. When a set of abnormal operators *OPs* is detected in the pivot GPU, we call the recursive `locateNeighbor` function for each operator (line 2). Given the low probability that multiple types of irregularities occur in the same operator, our algorithm treats input abnormal

[1] Typically, the GPUs performing DP communication will always perform the same EP communication. We group them together to further reduce the COG complexity. Furthermore, we do not group GPUs in PP communication as it is only executed among 2 GPUs.

Algorithm 1: Root cause localization

```
   Input: Ĝ = (O,C,E), OPs
   Output: OPERATORS, CAUSES
1  OPERATORS ← ∅, CAUSES ← ∅;
2  for all o of OPs do locateNeighbor(o) ;
3  Function locateNeighbor(o):
4      if o.type ∈ CC_OP then
5          ▷ Breadth First for CC
6          for all neighbors group of o do
7              if any neighbor operator n of group is normal then
8                  ▷ Track operators of the GPU g where n belongs to
9                  for all neighbors operators po of g before n do
10                     if any po is abnormal then
11                         locateNeighbor(po), break
12                     else
13                         OPERATORS.add(po),
                             CAUSES.add(COMPUTE), return;
14                     end
15                 end
16             else
17                 OPERATORS.add(o), CAUSES.add(NETWORK)
                     return;
18             end
19         end
20     else
21         ▷ Depth First for P2P
22         if o.peer = ∅ or o.peer operator is abnormal then
23             OPERATORS.add(o), CAUSES.add(NETWORK);
24             return;
25         end
26         ▷ Track operators of peer GPU
27         for all neighbors operator po of o.peer GPU do
28             if po before o.peer operator is abnormal then
                   locateNeighbor(po);
29         end
30     end
31 end
```

operators separately. For each communication operator, we use breadth-first search (BFS) and depth-first search (DFS) to traverse the operators in collective communication (CC) and point-to-point communication (P2P), respectively. For each GPU, we track the operators belonging to it and detect whether they are abnormal. We initialize the set of root cause operators and types of causes to $\emptyset$ (line 1).

BFS for Collective Communication. We perform a breadth-first search to traverse all the GPUs within a collective communication group. For an abnormal collective communication operator o, we identify its corresponding group c and verify all associated operators across group GPUs. These operators represent the same communication but are recorded from different GPUs. If all the operators are reported as abnormal, we confirm the communication itself as the root cause and terminate the analysis (line 17). Otherwise, there exists an operator n that is executed normally, i.e. normal execution duration. However, since the collective communication requires all the GPUs to be ready, this indicates that the abnormal operators in the rest of the GPUs take too much time waiting for the "normal" operator n to start. We can infer that irregularity occurs and delays n. In this case, we will track all the previous

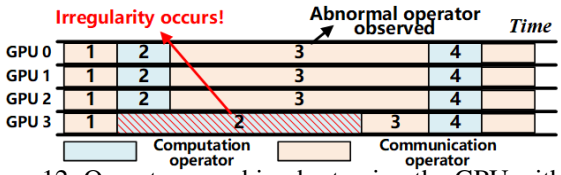


Figure 12: Operator searching by tracing the GPU with the normal collective operator.

operators before n (line 9). If any operator po is abnormal, we recursively analyze po (line 10), otherwise, we identify the causes as the computation prolongs and impacts n (line 13). Figure 12 demonstrates this principle through a 4-GPU scenario where GPU3's prolonged operator2 execution forces others to wait at operator3. Our algorithm correctly attributes the anomaly to GPU3's computational delay by observing abnormal operator3 executions in other GPUs versus GPU3's normal operator3.

DFS for P2P communication. In P2P communication, each operator involves two GPUs with peer relationships. We perform a depth-first search along with the pipeline stages. For each GPU, the other GPU is referred to as the peer GPU, and the corresponding operator in this peer GPU is termed the peer operator. For a specific P2P operator o, we directly detect its peer operator due to its inherent dependency. For example, if a `Recv` is abnormal, we can find its peer `Send` according to the COG. If both operator o and its peer operator are abnormal, then it can be confirmed that the cause of the irregularity is related to the abnormal P2P network communication (line 22). If only o is abnormal, its prolonged execution implies synchronization delays while awaiting the peer operator's readiness. In this case, we track the operators before the peer operator on the peer GPU and further locate the abnormal operators among them (lines 27-29). The readers can refer to appendix C for more details.

4.4 Cross-iteration Analysis

By combining the detection (§ 4.2) and localization (§ 4.3), we are able to find the root abnormal operators, involved GPUs and the cause type of an irregularity. However, there may be false positives if we localize the causes only based on a single iteration. Meanwhile, we can hardly identify multiple concurrent irregularities. To address these challenges, Holmes leverages a cross-iteration analysis in a time window T to find out the most possible anomalous devices.

Influence assessment of irregularities. Given detected irregularities $X = \{x\}$ within T, we first define a quality metric to evaluate the irregularity's influence on training. Since the irregularity prolongs the iteration time for δ times, Holmes calculates the impact of irregularity on the overall iteration time as a metric called *irregularity rate* $\mathcal{R}(x, t_s, T)$, where the time window starts at t_s. The impact of irregularity can be reflected in its corresponding abnormal operators. $\mathcal{R}(x, t_s, T)$ is calculated as the Pearson Correlation Coefficient [4, 59] of

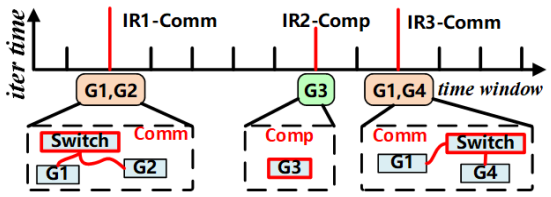


Figure 13: Cross-iteration analysis.

the abnormal operator duration and the iteration time:

$$\mathcal{R}(x,t_s,T) = \frac{Cov\left(\mathbb{T}^T_{t_s}(x), \mathbb{T}^T_{t_s}(I)\right)}{\sigma_{\mathbb{T}^T_{t_s}(x)}\sigma_{\mathbb{T}^T_{t_s}(I)}}$$

where $\mathbb{T}^T_{t_s}(x)$ denotes the series of the execution time of the abnormal operator involved in x, and $\mathbb{T}^T_{t_s}(I)$ denotes the series of the overall iteration time. The larger $\mathcal{R}$ an irregularity has, the more severe influence it has on training.

Anomalous device estimation. Any anomalous device in the cluster leads to the irregularities, we here to calculate the anomalous probability of each device. Noting that the abnormal operators may be different among iterations, they can be caused by the same anomalous device. We first define the device set $D = \{d\}$, including such as GPUs, links connecting GPU, NVLinks, and switches. Each device has an *accumulated index* $I(d)$ to indicate its possibility of being anomalous. With the irregularity rate $\mathcal{R}$ and involved GPUs of the irregularities, we can calculate all the $I(d)$ by accumulating $\mathcal{R}$. Specifically, for the irregularity that is identified as computation issues in a set of GPUs, we directly accumulate $\mathcal{R}$ into I of the GPUs. While for irregularity stemming from the network issues, we traverse all the network devices that lie on the communication paths among involved GPUs, and accumulate the $\mathcal{R}$ into their I. Given a device d and irregularities set X, we use the MAP estimation to perform the accumulation as described in appendix D, which exploits the prior probability of the devices being anomalous. Figure 13 depicts the communication irregularity "IR" involves GPU1 and GPU2. We accumulate the irregularity rate to all the communication devices between them, i.e. a switch and two links. While for the computation irregularity "IR" involving GPU3, we directly accumulate the irregularity rate to GPU3. Finally, Holmes ranks $I(d)$ and identifies the top K devices with the highest $I(d)$, which means that they are most likely to be anomalous and severely affect the training.

5 Evaluation

In this section, we present a series of experiments on Holmes. Experimental results manifest that Holmes can achieve the localization accuracy of 97.21% in the models (§ 5.1) trained in a mega-scale GPU cluster. Our production-level testbed simulations show that the irregularity localization can be completed within 30.3 seconds (§ 5.2). Finally, we study Holmes's localization details by a case from production (§ 5.3).

5.1 Large-scale Trace-driven Analysis

We collect CommOps logs over a three-month period from our production training cluster, which is equipped with over 10,000 GPUs, including NVIDIA H800. Each GPU is paired with an NVIDIA ConnectX-7 Dx NIC, providing a bandwidth of 400Gbps. The software systems include Megatron-LM [41], DeepSpeed [47], and the NCCL [1]. The training process adopts multi-dimensional parallelism strategies that encompass tensor- [41], data- [11,46], pipeline- [22,40], and expert- parallelism [28]. The LLMs are carefully designed models based on LLaMA [56], LLaMA2 [57] and GPT3 [7], with thousands of billions of parameters.

Dataset. We integrate our CommOps logger into our production cluster and generate the communication logs accordingly. Manual diagnosis is conducted to identify anomalous devices and corresponding irregularities. Firstly, we rely on expert knowledge to enumerate all the possible anomalous devices. Subsequent manual inspection evaluates device resource metrics (e.g., GPU usage). Suspected devices are isolated upon confirmation through stress testing.

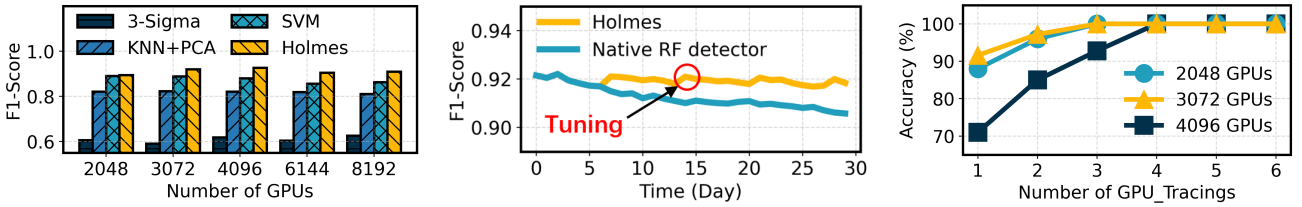
System configuration. The training set of the detection tree model is the CommOps log data of LLM training. Specifically, we train a distinct tree model for every LLM training using their CommOps logs. Each sample represents the feature of a specific communication operator. Most of the LLM training spans several months and we manually construct the training dataset using a few hours' logs. For cross-iteration analysis, given the low probability of concurrent failures in multiple devices, K is defaulted to 2. The irregularity rate calculation spans 100 iterations by default.

5.1.1 Modular study

We present the key modular study results below and other supplementary experiments can be found in appendix E.

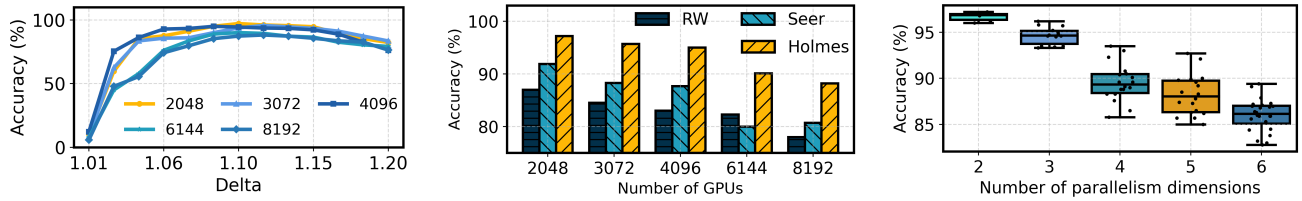
Abnormal operator detection accuracy. We first study the performance of abnormal operator detection in Holmes. Figure 14(a) compares our RF detector with three baseline methods, where higher F1-Score indicates better detection performance, the details of the setting can be referred to appendix F. For various scales of the training, the F1-Score of RF exceeds 0.89. Specifically, when training with 4096 GPUs, the F1-Score of RF reaches 0.93, surpassing the scores of the 3-sigma rule and K-Nearest Neighbor (KNN) by 0.31 and 0.11, respectively. Even when compared to SVM, Holmes still achieves a higher F1-Score by 0.07. These phenomena manifest that our RF detector could detect abnormal operators correctly even in large-scale training.

Distribution drift adaptation. We then study the auto-tuning of Holmes against the distribution drift. For a 3072 GPUs training lasting one month, we set the auto-tuning interval to one week and train the same RF model for Holmes and a naive RF detector without tuning. Figure 14(b) illustrates how the F1-Scores evolve over time. Starting from the second



(a) F1-Score of different detection methods (b) Distribution drift adaption (c) Localization accuracy

Figure 14: Performance of detection.



(a) Localization accuracy of δ (b) Scalability to number of GPUs (c) Scalability to number of parallel dimensions

Figure 15: End-to-end localization accuracy.

week, Holmes consistently outperforms the native RF. The F1-Score of Holmes is always above 0.917, while the score of native RF declines from 0.922 on the first day to 0.906 by the end of the month. Meanwhile, an increase in the F1-Score is clearly observed on each auto-tuning date. Holmes automatically collects the feature shift such as iteration time shift and tunes the attributed value for these features weekly, keeping high detection performance. Furthermore, the tuning frequency can be adapted according to the training.

Localization algorithm accuracy. To evaluate the localization algorithm, we present per-iteration localization accuracy on three training with thousands of GPUs. As shown in Figure 14(c), we run the localization algorithm several times independently, beginning with 1 to 6 randomly selected monitored GPUs. The GPUs that have the earliest executed abnormal operators are then chosen as the localization results. When monitoring a single GPU, Holmes achieves localization accuracy of 91.6%, 88.0%, and 71.1% with 2048, 3072 and 4096 GPUs, respectively. As the training involves more GPUs, GPUs are more likely to interfere with each other, which limits the localization accuracy. However, since the localization algorithm starts from few GPUs, we can monitor more GPUs to improve the localization accuracy in production. For example, in training with 4096 GPUs, the accuracy increases from 71.1% to 100% as the monitored GPUs increase.

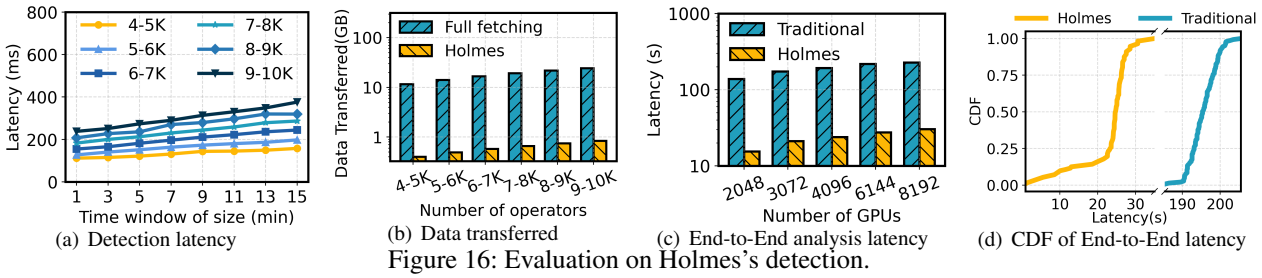
5.1.2 End-to-end Performance

Baselines. Since there are no existing systems for irregularity localization during training, we adapt the idea of two anomaly diagnosis counterparts [16, 59] from the field of (micro-)service computing to irregularity localization. "Seer" is a neural network based method [16] that directly localizes the potential anomalous device using the CommOps logs. We further replace our graph-search algorithm with the ranking and random walk method [59], called "RW".

Accuracy of delta. The irregularity refers to the training iteration time prolongs for δ times. Hereby we evaluate Holmes's performance when δ varies. As shown in Figure 15(a), Holmes can detect the irregularities in most of the training when δ varies from 1.06 to 1.12. When the δ is less than 1.04, Holmes can not tell the normal iterations and irregular iterations since the training iteration time fluctuates. While δ is larger than 1.12, Holmes can not detect slight irregularities like performance fluctuations. We empirically set our default δ of the detection system as 1.1.

Scalability to GPU number. Figure 15(b) shows the end-to-end localization accuracy when training LLMs at various GPU scales. With 2048 GPUs, Holmes achieves a localization accuracy of 97.2%. Even in the extreme scenario of 8192 GPUs, Holmes maintains an accuracy of 88.2%, while RW and Seer attain only 78.0% and 80.7%, respectively. RW's random walk on COG effectively localizes anomalous *communication* devices but cannot detect computational anomalies, as CommOps logs do not explicitly provide their information. Seer exhibits rapid accuracy degradation with GPU scaling, as its CNN+LSTM architecture struggles to model inter-operator dependencies at extreme scales. Holmes explicitly models the operator relationships, thereby achieving superior localization accuracy and interpretability.

Scalability to parallelism dimension. We assess the localization accuracy of Holmes when the training parallelism varies. The parallelization strategies include the aforementioned strategies and customized approaches. The LLMs in our experiments implement various combinations of these strategies, so each test case is designated by the number of parallelism dimensions rather than the specific combinations, as shown in Figure 15(c). For the commonly used 3D and 4D parallelisms, Holmes achieves a median localization accuracy of 94.6% and 89.6%, with a first quartile of 93.6% and 88.3%, respectively. The more complex parallelism introduces the more complicated communication patterns, leading to an accuracy decline. However, even with 6D parallelism, Holmes's median localization accuracy reaches 86.0%, with a



(a) Detection latency (b) Data transferred (c) End-to-End analysis latency (d) CDF of End-to-End latency

Figure 16: Evaluation on Holmes's detection.

first quartile of 85.2%, which shows that Holmes is adaptable to existing parallelization strategies.

5.2 Production-level Testbed

Localization latency is evaluated in our production network environments through a simulation framework that enables dynamic GPU configuration adjustments. Holmes's prototype operates on production network infrastructure, employing CPU-based log generation to emulate GPU training processes while preserving real collective communication operator logs. This approach maintains temporal fidelity by substituting GPU computation durations with simulated equivalents, with log transmission mechanisms mirroring production environments. Our testbed consists of nine machines: one acts as the Holmes analytics server, while the others serve as training machines. For example, to simulate a virtual cluster with 2048 GPUs, each training machine runs 256 log writers, acting as 256 GPU training processes. Each machine is equipped with an AMD EPYC 7K83 64-Core Processor and a 200 Gbps NVIDIA ConnectX-6 Dx NIC. Machines are connected to a Clos network using 10 TCS8400 switches to simulate the data plane network in production (Figure 22). We compare Holmes to the 3-sigma rule approach, labeled as "Traditional", which uses all GPUs' logs.

Detection latency. We study the detection latency by adjusting the detection time window size and the number of communication operators. A larger window size means that more iterations are processed in Holmes. As shown in 16(a), the overall detection latency for a single GPU log remains within 400 ms. When the window size increases from 1 minute to 15 minutes, the detection latency increases from 111.7 ms to 157.6 ms with 4-5K operators inside a log. The detection latency increases linearly as more operators are involved, since each operator needs to be processed by the RF model. An observation is that even when the time window extends to 15 minutes and the number of operators reaches 9-10K, the detection latency remains as low as 375.7 ms.

Localization with progressive fetching. To reduce collected data volume and localization time, Holmes implements a progressive fetching scheme that only collects logs from the GPUs that are traversed by our localization algorithm. Figure 16(b) shows the collected logs from the training machines to the analytics server for one localization process. While full log analysis is used as our baseline. For cases with 9-10K operators, the average per-iteration traffic load is 24.36 GB, whereas Holmes's load is only 0.84 GB, with a reduction of 96.6%. This manifests that our progressive fetching scheme effectively utilizes only the necessary data, preventing network overload due to excessive CommOps log traffic.

End-to-end localization latency. We evaluate the overall localization latency of Holmes when training with different cluster scales. As shown in Figure 16(c), Holmes significantly outperforms the traditional approach in all scenarios. For instance, the average latency in Holmes is 21.2 seconds when training with 3072 GPUs, exhibiting an 87.8% reduction compared to the baseline. As the number of GPUs scales from 2048 to 8192, the overall latency of Holmes increases from 15.5 seconds to 30.3 seconds. This increase is due to the prolonged detection time for irregular operators and the increased transmission time for CommOps logs. Figure 16(d) shows the cumulative distribution of the latency for the training using 3072 GPUs. Nearly 96.2% of the overall latency values fall below 30.0 seconds, while the 99th percentile is at 32.8 seconds. This implies the absence of a heavy tail, which is advantageous for Holmes's practical operation.

5.3 Case Study

We use an irregularity example from the production to illustrate the workflow of Holmes in Figure 17. The training in this case is executed under typical 3D parallelism, where the degrees of DP, PP and TP are 96, 8 and 4, respectively.

1) Irregularity detection. At the beginning, Holmes keeps monitoring the training in GPU 0 (DP:0, PP:0, TP:0). Upon the detection of an irregular iteration (whose time extends to 10.1s), we perform the abnormal operator detection on GPU 0, and find a `Send` operator with extended execution time.

2) DFS inside PP group 0. As the `Send` is a P2P operator in PP group 0, Holmes fetches the CommOps log in the peer GPU (DP:0, PP:1, TP:0) according to the COG. We find that the P2P operator `Receive` is normal while another `Send` operator is abnormal. We keep searching on this `Send`. After repeating multiple rounds, we transit to GPU 2304 (DP:0, PP:6, TP:0) whose `All-Reduce` operator is found abnormal.

3) BFS inside DP group 24. The `All-Reduce` belongs to DP group 24, we then detect all the GPUs in this DP group according to the COG. Only one GPU's (GPU 2572) corresponding `All-Reduce` is normal, while the other GPUs have the same abnormal DP `All-Reduce` operator. This indicates

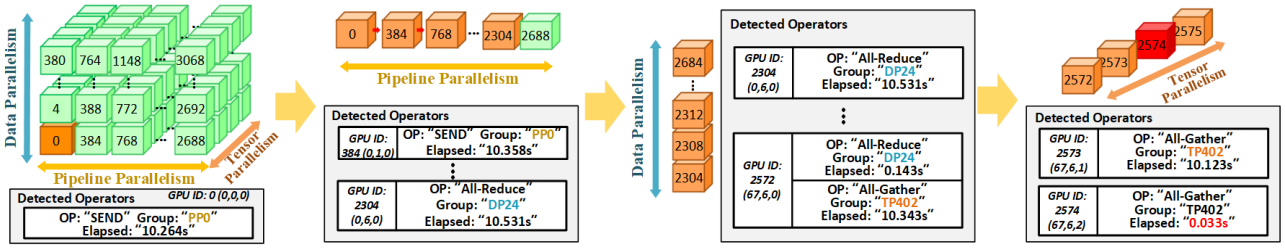


Figure 17: An example of irregularities detection with 3072 GPUs.

that GPU 2572 is the straggler and the rest of the GPUs start the `All-Reduce` ahead of GPU 2572 and wait till it is prepared. In GPU 2572, the abnormal operator we can detect is an `All-Gather` belonging to TP group 402.

4) BFS inside TP group 402. In this group, we find that the same `All-Gather` is normal only in GPU 2574, whereas no other abnormal operator in this GPU. We can infer that the irregularity is caused by the GPU 2574 computing slowly.

Summary. In this example, the number of GPUs we access is 106 (i.e., $6+96+4$), which is less than the total number of 3072. Combining with a long-term analysis during hundreds of iterations, Holmes suggests substituting the GPU 2574.

6 Discussion

Considering operator implementation. Communication operators have been implemented by various algorithms (e.g., All-Reduce variants [1, 50]) including ring [44, 55], recursive doubling [49, 55], and topology-aware optimizations [15, 34]. While operator-level behavioral discrepancies enable Holmes's straggler detection, integrating implementation-specific operator tracking could enhance network device localization. This refinement, however, risks excessive logging overhead during multi-hop transmission monitoring – a critical trade-off requiring future exploration between diagnostic precision and runtime efficiency.

Resource metric integration. Traditional hardware monitors provide coarse-grained metrics (GPU utilization, temperature) with inherent noise and limited LLM context awareness. Though augmenting Holmes with real-time resource traces could improve localization – for instance, validating GPU status during diagnostic – our current focus remains anomaly characterization and core detection framework development, reserving multi-source correlation for subsequent research.

Experience in using network telemetry. Initial experiments with RNIC queue-pair (QP) telemetry achieved full communication reconstruction via per-chunk logging, yet proved impractical for large clusters: 10,000 RNICs generated over 10TB/iteration of QP data, exceeding real-time processing capabilities. This scalability constraint motivated Holmes's communication library (CCL)-centric design, balancing observability with operational feasibility.

7 Related Work

Failure diagnosis in GPU cluster. Recent research on anomaly analysis in machine learning has focused on failure diagnosis. A series of works [21, 25, 43, 54, 62] employed error logs for diagnosis. For example, MegaScale [25] diagnosed the failure by visualizing the data collected with a custom CUDA event monitor. Different from them, Holmes localizes the irregularity according to the communication relationship between GPUs without error logs. Another failure diagnosis approach was made by resource monitoring, such as GPU utilization, memory usage, etc [2, 18, 23, 32]. However, they lacked awareness of the detailed training process, while the diagnosis performance highly depends on the granularity of resource monitoring. Holmes can learn the training process precisely at the operator level from the communication logs. Some works [8, 9, 48, 52, 53, 58] focus on develop performance debugging tools for high-performance computing (HPC). Holmes leverages domain-specific knowledge like parallel training strategies to detect irregularity.

Anomaly diagnosis in distributed applications. Research has focused on anomaly diagnosis in other distribution applications, which typically involve detection and localization. In anomaly detection, some work [24, 38, 39] directly analyzed the operational logs, while the others [12, 16, 26, 33, 36, 61] focused on leveraging machine learning. For example, Seer [16] trained a neural network to detect QoS violations. Holmes enhances the machine learning detection model to adapt to the distribution drift in LLM training. In anomaly localization, a series of research [6, 29, 30, 33, 59] proposed the idea of building a graph based on specific applications. MicroHECL [30] analyzed anomaly propagation chains based on a dynamically constructed service call graph. Holmes continues the idea of graph analysis and proposes a novel graph to model the communication relationship within LLM training.

8 Conclusion

In this paper, we systematically analyze the irregularity phenomenon in LLM training and propose a first-of-its-kind system called Holmes for irregularity localization. Holmes employs an enhanced abnormal operator detection model and a novel communication operator graph to perform efficient irregularity localization. Evaluation of production traces shows that Holmes achieves localization accuracy of 97.21%.

Acknowledgements

We thank our shepherd Fang Zhou and the anonymous reviewers for their constructive feedback. This work was supported in part by the Natural Science Foundation of China under Grant 62072117, the Shanghai Natural Science Foundation under the Grant 22ZR1407000, and the Tencent Research Program RBFR20230724.

References

[1] Nvidia NCCL, 2024. https://github.com/nvidia/nccl.

[2] Nvidia Nsight Systems, 2024. https://developer.nvidia.com/nsight-systems.

[3] Torch Profiler, 2024. https://pytorch.org/docs/stable/profiler.html.

[4] H. Abe and S. Tsumoto. Analyzing behavior of objective rule evaluation indices based on a correlation coefficient. In *Knowledge-Based Intelligent Information and Engineering Systems, 12th International Conference, KES 2008, Zagreb, Croatia, September 3-5, 2008, Proceedings, Part II*, volume 5178 of *Lecture Notes in Computer Science*, pages 758–765. Springer, 2008.

[5] L. A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[6] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero. Graph-based root cause analysis for service-oriented and microservice architectures. *J. Syst. Softw.*, 159, 2020.

[7] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

[8] M. Burtscher, B. Kim, J. R. Diamond, J. D. McCalpin, L. Koesterke, and J. C. Browne. Perfexpert: An easy-to-use performance diagnosis tool for HPC applications. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pages 1–11. IEEE, 2010.

[9] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development*, 50(2.3):239–248, 2006.

[10] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023.

[11] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1232–1240, 2012.

[12] Q. Du, T. Xie, and Y. He. Anomaly detection and diagnosis for container-based microservices with performance monitoring. In *Algorithms and Architectures for Parallel Processing - 18th International Conference, ICA3PP 2018, Guangzhou, China, November 15-17, 2018, Proceedings, Part IV*, volume 11337 of *Lecture Notes in Computer Science*, pages 560–572. Springer, 2018.

[13] W. Fan, P. Wang, D. Wang, D. Wang, Y. Zhou, and Y. Fu. Dish-ts: A general paradigm for alleviating distribution shift in time series forecasting. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 7522–7529. AAAI Press, 2023.

[14] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.*, 23:120:1–120:39, 2022.

[15] G. Feng, D. Dong, and Y. Lu. Optimized MPI collective algorithms for dragonfly topology. In L. Rauchwerger, K. W. Cameron, D. S. Nikolopoulos, and D. N. Pnevmatikatos, editors, *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*, pages 14:1–14:11. ACM, 2022.

[16] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 19–33. ACM, 2019.

[17] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang, S. Zhang, M. J. Fernandez, S. Gandham, and H. Zeng. RDMA over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*, pages 57–70. ACM, 2024.

[18] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 350–361. ACM, 2011.

[19] H. He, Q. Zhang, K. Yi, K. Shi, Z. Niu, and L. Cao. Distributional drift adaptation with temporal conditional variational autoencoder for multivariate time series forecasting. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–15, 2024.

[20] J. He, J. Zhai, T. Antunes, H. Wang, F. Luo, S. Shi, and Q. Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 120–134. ACM, 2022.

[21] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, and T. Zhang. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 709–729. USENIX Association, 2024.

[22] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. X. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[23] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 947–960. USENIX Association, 2019.

[24] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, pages 25–32. IEEE, 2017.

[25] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, Y. Jia, S. He, H. Chen, Z. Bai, Q. Hou, S. Yan, D. Zhou, Y. Sheng, Z. Jiang, H. Xu, H. Wei, Z. Zhang, P. Nie, L. Zou, S. Zhao, L. Xiang, Z. Liu, Z. Li, X. Jia, J. Ye, X. Jin, and X. Liu. Megascale: Scaling large language model training to more than 10, 000 gpus. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 745–760. USENIX Association, 2024.

[26] M. Jin, A. Lv, Y. Zhu, Z. Wen, Y. Zhong, Z. Zhao, J. Wu, H. Li, H. He, and F. Chen. An anomaly detection algorithm for microservice architecture based on robust principal component analysis. *IEEE Access*, 8:226397–226408, 2020.

[27] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro. Reducing activation recomputation in large transformer models. *CoRR*, abs/2205.05198, 2022.

[28] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[29] Y. Li, J. Tan, B. Wu, X. He, and F. Li. Shapleyiq: Influence quantification by shapley values for performance debugging of microservices. In T. M. Aamodt, M. M. Swift, and N. D. E. Jerger, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 287–323. ACM, 2023.

[30] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. Wu. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 338–347. IEEE, 2021.

[31] J. Liu, J. H. Wang, and Y. Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 486–498. ACM, 2023.

[32] K. Liu, Z. Jiang, J. Zhang, H. Wei, X. Zhong, L. Tan, T. Pan, and T. Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 15–29. USENIX Association, 2023.

[33] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, pages 48–58. IEEE, 2020.

[34] J. Ma, D. Dong, C. Li, K. Wu, and L. Xiao. PAARD: proximity-aware all-reduce communication for dragonfly networks. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), New York City, NY, USA, September 30 - Oct. 3, 2021*, pages 255–262. IEEE, 2021.

[35] K. Maeng, S. Bharuka, I. Gao, M. C. Jeffrey, V. Saraph, B. Su, C. Trippel, J. Yang, M. Rabbat, B. Lucia, and C. Wu. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[36] L. Mariani, C. Monni, M. Pezzè, O. Riganelli, and R. Xin. Localizing faults in cloud systems. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 262–273. IEEE Computer Society, 2018.

[37] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*, Nov. 2023.

[38] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Trans. Parallel Distributed Syst.*, 24(6):1245–1255, 2013.

[39] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 215–224. ACM, 2016.

[40] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 1–15. ACM, 2019.

[41] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 58. ACM, 2021.

[42] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[43] G. Ostrouchov, D. Maxwell, R. A. Ashraf, C. Engelmann, M. Shankar, and J. H. Rogers. GPU lifetimes on titan supercomputer: survival analysis and reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 41. IEEE/ACM, 2020.

[44] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distributed Comput.*, 69(2):117–124, 2009.

[45] S. Rajasekaran, M. Ghobadi, and A. Akella. CASSINI: network-aware job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 2024.

[46] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 20. IEEE/ACM, 2020.

[47] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. ACM, 2020.

[48] S. Roy, A. C. König, I. Dvorkin, and M. Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1167–1178. IEEE Computer Society, 2015.

[49] P. Sack and W. Gropp. Collective algorithms for multiported torus networks. *ACM Trans. Parallel Comput.*, 1(2):12:1–12:33, 2015.

[50] D. D. Sensi, T. Bonato, D. Saam, and T. Hoefler. Swing: Short-cutting rings for higher bandwidth allreduce. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 2024.

[51] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10435–10444, 2018.

[52] D. Skinner and W. Kramer. Understanding the causes of performance variability in hpc workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 137–149, 2005.

[53] H. Su, M. Billingsley, and A. D. George. Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–8. IEEE, 2008.

[54] A. Taherin, T. Patel, G. Georgakoudis, I. Laguna, and D. Tiwari. Examining failures and repairs on supercomputers with multi-gpu compute nodes. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 305–313. IEEE, 2021.

[55] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, 2005.

[56] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[57] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[58] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. M. Brandt, V. J. Leung, M. Egele, and A. K. Coskun. Online diagnosis of performance variation in HPC systems using machine learning. *IEEE Trans. Parallel Distributed Syst.*, 30(4):883–896, 2019.

[59] J. Weng, J. H. Wang, J. Yang, and Y. Yang. Root cause analysis of anomalies of multitier services in public clouds. *IEEE/ACM Trans. Netw.*, 26(4):1646–1659, 2018.

[60] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 945–960. USENIX Association, 2022.

[61] L. Wu, J. Tordsson, E. Elmroth, and O. Kao. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*, pages 1–9. IEEE, 2020.

[62] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang. An empirical study on program failures of deep learning jobs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1159–1170. ACM, 2020.

[63] Y. Zhou, C. Liang, N. Li, C. Yang, S. Zhu, and R. Jin. Robust online matching with user arrival distribution drift. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 459–466. AAAI Press, 2019.

Appendix

A Irregularity statistics

During the three months of tracing LLM training, we manually diagnosed a total of 126 anomalous devices which led to irregularities. As shown in Table 1, the majority were primarily caused by anomalous GPUs, accounting for 58.7%. Anomalous NICs were responsible for 23 occurrences, making up to 18.3%. Additionally, there were other anomalous devices involved, such as links, switches and CPUs. For Figure 3(b), additionally, this increase of irregularity frequency is sub-linear with the increasing number of GPUs. This is attributed to the fact that although the expectation of the number of anomalous devices increases sharply, one irregularity in overall training can be induced by multiple anomalous devices when a large number of GPUs are used.

Table 1: Anomalous devices of irregularity during a three month of LLM training.

Anomalous Device	Count	Percentage
GPUs	74	58.7%
NICs	23	18.3%
Links	12	9.5%
Switches	6	4.8%
CPUs	4	3.2%
Others	7	5.5%

B Periodic feature of fluctuation

The performance fluctuations exhibit an interesting periodic feature. Recall in Figure 6(a), we zoom in the per-iteration times and scrutinize 60 iterations between 50400 and 50460. An observation is that performance fluctuation shows a periodicity, approximately occurs once every 6-9 iterations. At other times, the duration of iterations fluctuates within a small range between 8.18 seconds to 9.45 seconds. To investigate the fluctuation, we map the iteration time into the frequency domain by applying the Discrete Fourier Transform (DFT) and Figure 18 shows the corresponding amplitude spectrum. Significant frequency components can be observed at 0.144, 0.288, and 0.432, which corresponds to periods of 6.94, 3.47, and 2.31 iterations respectively. Although the periodicity may

not be strict and the impact of each performance fluctuation varies, this can indicate that performance fluctuation and abnormal spike are different irregularities.

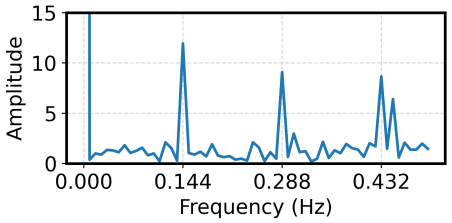


Figure 18: The frequency spectrum.

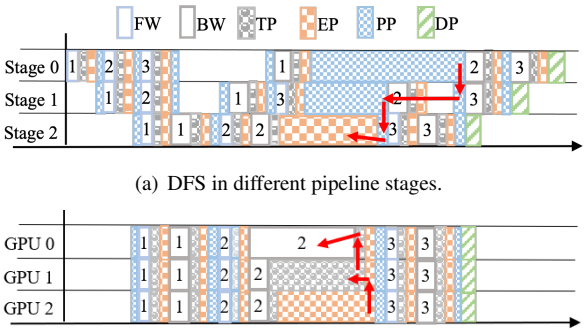
C Algorithm insights

We use two examples to explain our insights on the algorithm.

DFS for point-to-point communication. We illustrate an example in Figure 19(a), where an irregularity occurs in stage 2, leading to the overall training being prolonged. Suppose that the GPU at stage 0 is being monitored, and its "PP" operator has been identified as irregular. It is unclear whether this prolonged "PP" operator is due to the communication link anomaly at stage 0, or if it is influenced by the malfunctions on other GPUs. We then track the same "PP" operator on the downstream stage 1 and find that its execution time is normal, which excludes the inter-stage communication issue between stages 0 and 1. We further backtrack the communication operators on stage 1 and detect an anomalous "PP" operator. The reason for backtracking is intuitive, as the prolonged execution time of one operator is caused by anomalies in preceding operators or in the operator itself. By repeating the above procedure, we move to stage 2 and observe an abnormal "EP" operator, which is different from the previous "PP" operators. The DFS and the corresponding backtracking trajectories are illustrated as the arrow lines in Figure 19(a).

BFS for collective communication. As shown in Figure 19(b), where GPU 0 and GPU 1 are in the same TP group, GPU 1 and GPU 2 are in the same EP group. If the monitored anomalous operator in Holmes is the "EP" operator in GPU 2, we traverse the GPUs in the same EP group and locate the normal "EP" operator in GPU 1. To further diagnose the anomaly, we backtrack to the preceding communication operators and identify an anomalous "TP" operator. By traversing the other GPUs within the same TP group, we locate a normal "TP" operator in GPU 0. Since no other abnormal operators are detected in GPU 0, we can infer that the irregularity is likely caused by GPU 0 computing slowly. The BFS and the corresponding backtracking trajectories are illustrated as the arrow lines in Figure 19(b).

Correlation between operators and nodes Holmes localizes the irregularities with the correlations among operators and correlations between operators and nodes. To be specific, the correlations among operators help us to identify which operator belonging to a collective communication is the root



(a) DFS in different pipeline stages.

(b) BFS in collective communication groups.

Figure 19: Cause localization examples.

cause of the irregularity. The correlations between operators and nodes help us to recursively find the GPU where the irregularity takes place. However, deeper integration of these correlations and Holmes can be further taken into consideration. For example, investigating the local correlation of nodes and their operators. As this is not the main contribution of our paper, we leave this as our further research.

D Device-wise irregularity rate decay.

MAP-based $I(d)$ accumulation. The accumulation formula is shown as the following:

$$I(d) = \Big(\sum_{x \text{ uses } d} \mathcal{R}(x) + a(d)\Big) \Big/ \sum_{d} \sum_{x \text{ uses } d} \mathcal{R}(x) + a(d) + b(d)$$

where Beta distribution $I(d) \sim \mathrm{Be}(a(d), b(d))$ is used as the prior distribution. $a = \alpha \cdot p(d)$, $b = \alpha \cdot (1 - p(d))$, where $p(d)$ is the device irregularity ratio from our production experience and α is a scaling factor.

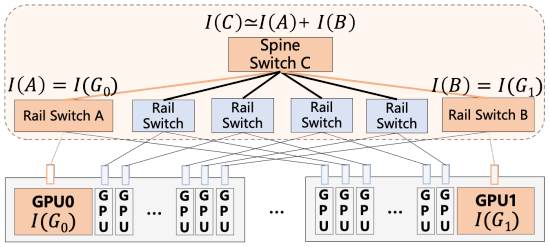
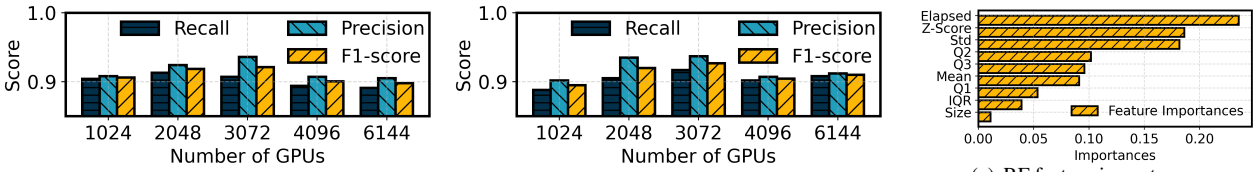


Figure 20: Topology-aware Irregularity rate decay.

In the estimation in § 4.4, we treat each device set equally. However, this could lead to unfairness since the connection of the network fabric is uneven. For example, assuming that irregularities are occurring in both rail switches A and B which slows down the upside transmission. If we traverse the network devices, we could detect irregularities that involve A, B and the spine switch C, and $I(C) \simeq I(A) + I(B)$, as shown in Figure 20. In this case, C will be identified as the most possible device in error. Holmes also designs a device-wise irregularity rate decay mechanism to address this issue. For a



(a) Accuracy with Gini Impurity (b) Accuracy with Entropy (c) RF feature importance

Figure 21: Evaluation on Holmes's detection.

network device d with connections $C(d)$ to downside NICs, we define its decay factor $H(d)$ as follows:

$$H(d) = \frac{\ln(C(d) + \varepsilon - 1)}{C(d)}$$

wherein $\varepsilon = 1e-6$. By substituting the irregularity rate as decayed irregularity rate, the final estimation of device anomalous probability is:

$$I(d) = \frac{\sum_{x \text{ uses } d} \mathcal{R}(x) \cdot H(d) + a(d)}{\sum_d \sum_{x \text{ uses } d} \mathcal{R}(x) \cdot H(d) + a(d) + b(d)}.$$

E Evaluation results

We hereby present the supplemental evaluation results of Holmes. The Network topology used in the testbed is shown in Figure 22.

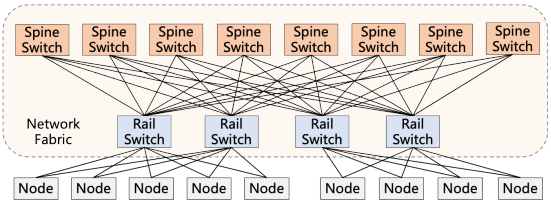


Figure 22: Network topology used in the testbed.

CommOps logger overhead. The communication logger's performance impact is evaluated across multiple training scales (Figure 23). Results show training speed degradation of only 0.064% and 0.083% for 1,024 and 3,072 GPU training when deploying CommOps logger, respectively. Even in the extreme scenario with 6144 GPUs, training time increases by 0.19%, with per-operator logging averaging under $2\mu s$. These metrics confirm the logger's ultra-lightweight design and negligible overhead across cluster scales.

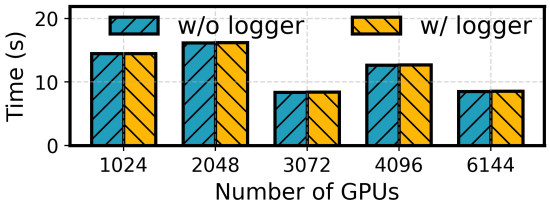


Figure 23: Impact on training with CommOps logger enabled.

Diagnosis of different types of irregularities. We present the diagnosis accuracy of the three types of irregularities across different training scales, as shown in Figure 24. The average diagnosis accuracy of Performance spike, Performance fluctuation, and Performance degradation is 90.7%, 93.0%, 93.2% respectively. Detecting Performance spikes is more challenging because they occur randomly over time, making it difficult to determine when a spike happening. Performance fluctuation and Performance degradation are easy to detect as they typically follow periodic or persistent patterns.

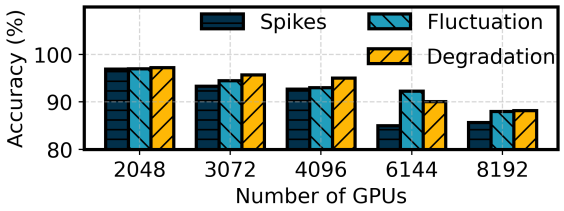


Figure 24: Accuracy for three types of irregularities.

Auto-tuning using different features. We compare Holmes's auto-tuning using different features including P75 (75th-percentile latency), P99 (99th-percentile latency), and mean operator execution time in Figure 25. Experimental results show that tail latency-based tuning (P75 and P99) performs worse than the mean-based strategy, due to their greater sensitivity to irregularities. For example, when training with 8192 GPUs, tuning the detection model using mean operator execution time achieves the highest F1-Score of 0.91, while all the others are less than 0.90. Notably, P99 performs slightly worse than P75, as it is more sensitive to extreme values.

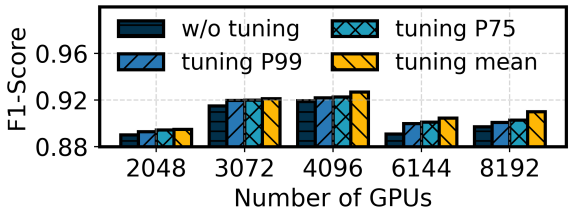


Figure 25: F1-Score of tail latency tuning.

MAP-based irregularity rate accumulation. We conducted comprehensive experiments to compare the accumulation approaches of $R(x)$ in cross-iteration device localization as shown in Figure26. To use the prior device irregularity ratio from our production experience, we employ a MAP-based irregularity rate accumulation approach, while we compare it with the naive linear accumulation without prior. In experiments of different scales, using the prior knowledge is proved

more effective than calculating the sum of the irregularity rate. The greatest improvement was observed in the setup with 8192 GPUs, where the accuracy was 3.3% higher.

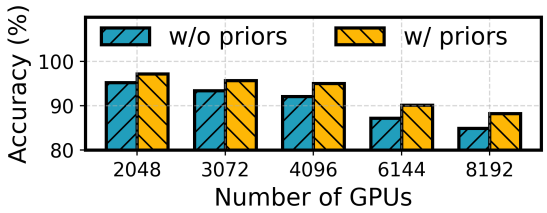


Figure 26: Impact of prior knowledge on localization.

Training throughput. In § 5.3 we illustrate an example of Holmes's localization and find the root cause of irregularity is that GPU 2574 computes slow. As shown in Figure 27. After we isolate the node containing GPU 2574, there are no spikes observed and the average training throughput increases from 85.43 samples/second to 90.62 samples/second.

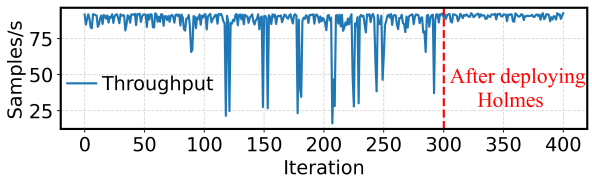


Figure 27: The example training throughput.

F Detailed evaluation of RF

Configuration of baselines of RF models. We compare our RF detection model with other machine learning models like 3-Sigma, KNN, KNN+PCA. The 3-sigma rule we used is performed over each individual operator's own time series. According to the 3-sigma rule, the operator elapsed time within the $(\mu-\sigma,\mu+\sigma)$ range accounts for 68.27% of all its own elapsed time series, and the others can be regarded as abnormal, where μ is the mean and σ is the standard deviation.

Model assessment. We also study the selection of RF models. Figure 21(a) and 21(b) show the Precision, Recall and F1-Score when we use Gini impurity and Entropy as the criterion of RF respectively. In all the evaluated training, the precision is higher than 0.9, while the Recall is higher than 0.85. In particular, we can observe that our RF detector obtains the best F1-Score in training with 3072 GPUs, which is 92.1% and 92.7% for Gini and Entropy. This means that our RF detector could detect the anomalous operators correctly even in large-scale training. Figure 21(c) further illustrates the feature importance distribution within the dataset, highlighting the key factors that contribute to the detection performance of the RF model.

SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale Large Language Model Training with Scalability and Precision

Xizheng Wang[1,2*], Qingxu Li[1*], Yichi Xu[1*], Gang Lu[1†], Dan Li[2], Li Chen[3], Heyang Zhou[1]
Linkang Zheng[1,4], Sen Zhang[1], Yikai Zhu[1], Yang Liu[1], Pengcheng Zhang[1], Kun Qian[1], Kunling He[1]
Jiaqi Gao[1], Ennan Zhai[1], Dennis Cai[1], Binzhang Fu[1]
[1]Alibaba Cloud, [2]Tsinghua University
[3]Zhongguancun Laboratory, [4]South China University of Technology

Abstract

The large number of GPUs required for a single LLM training significantly hinders the validation of new designs, tunings, and optimizations, calling for the occurrence of efficient simulators. Existing simulators, however, only target a specific granularity of the entire training, intrinsically leading to imprecision. This paper presents SimAI, a unified simulator aiming at precisely and efficiently simulating the LLM training procedure at scale. Through selective and high-fidelity integration of the training frameworks, the kernel computation, and the collective communication library into the simulating procedure, SimAI achieves high precision in simulations. SimAI further conducts multi-thread acceleration and implements lock-free global context-sharing to accelerate the execution speed. The effectiveness of SimAI is validated by its performance results, which show an average of 98.1% alignment to real-world results under various test scenarios and affirm its robustness and adaptability from small-scale labs to large-scale industrial environments. SimAI delivers meaningful guidelines for new host designs and parameter settings, directly benefiting in-production LLM training. We also share experiences and lessons learned during the evolution of SimAI. SimAI is open sourced at https://github.com/aliyun/SimAI.

1 Introduction

As the field of Artificial Intelligence (AI) rapidly advances, particularly with the rise of large language models like OpenAI's GPT-4 [41], the need for scaling AI infrastructure has grown significantly. For instance, training GPT-4 reportedly requires around 25,000 state-of-the-art GPUs [19]. This massive resource demand presents a significant barrier to entry for organizations looking to compete in this high-stakes domain.

In this context, simulators become essential both before and after infrastructure investments. In the planning phase, simulators help organizations estimate the scale and architecture required to achieve performance goals. In the operation phase, they help increase resource utilization, ensuring a return on investment. Thus, simulators are not just tools for improving efficiency but are strategic assets that maximize resource use and ensure that infrastructure investments deliver measurable results.

Currently, it is common practice to use separate simulators for capacity planning and performance tuning. For capacity planning, simulation is typically performed at the flow or job level, ignoring packet-level behavior [32]. In contrast, performance tuning relies on packet-level simulations to analyze network traffic patterns, latency, and packet loss—factors critical for optimizing communication and computation in AI model training and inference.

However, our experience shows that using multiple simulators with different levels of granularity presents three main challenges. First, this approach leads to inaccurate cost-performance analyses, making it difficult to predict load-balancing performance and assess system failures. Second, the poor performance of detailed simulators and the low accuracy of coarse-grained simulators limit the ability to optimize model training in large-scale deployments. Finally, this fragmented approach complicates development and testing, increasing the risk of discrepancies between simulated and real-world performance.

To overcome these challenges, we developed a unified simulator that handles both capacity planning and performance tuning within a single framework. However, this approach introduces several difficulties:

- Generating workloads with high precision to reflect realistic AI training behaviors.
- Simulating computation accurately across various GPU architectures.
- Precisely modeling communication to account for network traffic patterns and latencies.

* Equal Contribution
† Corresponding Author

- Scaling the simulator to support diverse, large-scale AI infrastructure configurations.

In this paper, we share the development and operation experience of SimAI, a unified simulator designed for scalable, high-precision simulations of large-scale LLM training. SimAI addresses these challenges by introducing high-fidelity models across the entire AI training stack. To generate precise workloads for LLM training at any scale, SimAI "hijacks" mainstream training frameworks, such as Megatron [52] and DeepSpeed [46], to run on a single host and create fine-grained workloads (§3.2). Different solutions ensure accurate simulation of both computation and communication. For computation, we break the workload into fine-grained kernels, measuring execution times on existing GPUs and mapping them to other GPU types (§3.3). For communication, we "hijack" the NVIDIA Collective Communications Library (NCCL) [36] to accurately simulate packet-level behavior for collective communication (§3.4). To improve the simulator's efficiency, we implement multi-threaded acceleration and lock-free global context sharing among threads (§3.5).

The performance results of our unified simulator are a testament to its efficacy. In terms of accuracy, it achieves an average deviation of only 1.9% compared to real-world results across various test scenarios. On the scalability front, the simulator handles simulations from small-scale lab environments to large-scale industrial deployments, proving its robustness and adaptability (§4).

Since incorporating this unified simulator into our AI development pipeline, we have served various teams and gained valuable insights. It has improved our infrastructure management and accelerated AI model development and deployment. We share the benefits and contributions of SimAI, including guidelines for new host designs and accurate assessments of scaling benefits (§5). These guidelines have been adopted by engineering teams and incorporated into production deployments. Additionally, we share lessons learned in transforming SimAI from a standalone simulator to a widely used simulation service (§6).

SimAI is a high-precision, full-stack simulator designed to benefit researchers across various domains involved in Large Language Model (LLM) training. This versatile tool caters to multiple levels of the LLM training ecosystem. At the framework level, SimAI enables the exploration of optimal parallel strategies and communication-computation overlap techniques, facilitating parameter tuning to reduce end-to-end training time. For collective communication research, it offers a platform to validate and quantify novel algorithms' performance gains. SimAI's system architecture design allows for experimentation with diverse intra-host and inter-host configurations, helping identify the most cost-effective solutions. From a networking perspective, the simulator supports investigations into the impact of various congestion control algorithms (such as DCQCN [64], HPCC [26], etc.), network protocols (such as RoCEv2 [17], Solar [33], etc.), and routing strategies—including adaptive routing—across different architectural setups. By providing the flexibility to customize and fine-tune different components, SimAI empowers users to conduct multifaceted research accelerating LLM training processes, and is an invaluable tool for scholars and practitioners throughout the LLM development pipeline.

2 Background and Motivation

2.1 AI Training Infrastructure

Large language models (LLMs) require specialized infrastructure, often involving dozens to thousands of GPUs working together to handle pretraining or fine-tuning tasks. For instance, training a GPT-3 model with 175 billion parameters demands 1,024 high-end GPUs running continuously for 34 days [35]. To optimize GPU usage, mainstream training frameworks like Megatron [52] and DeepSpeed [46] offer parallelization techniques such as Data Parallelism (DP), Pipeline Parallelism (PP), and Tensor Parallelism (TP). These methods enable the efficient distribution of training tasks across multiple GPUs. The process relies on collective communication libraries (CCLs) to manage data exchanges, such as using `AllReduce` for gradient synchronization or `AllGather` for parameter sharing. The CCL breaks each collective communication task into a series of peer-to-peer `Send` and `Receive` operations to carry out the necessary data transfers between GPUs.

In Alibaba's clusters, each server contains multiple GPUs. GPUs within the same server are connected through a high-bandwidth intra-host network, such as NVLink or NVLink Switch [40], and each GPU connects to the inter-host RDMA network via network interface cards (NICs). For example, in line with [23], our A100 servers are equipped with eight NVIDIA A100 GPUs [1] and four NVIDIA CX6Dx NICs, each providing 2×100Gbps [6] bandwidth. Each GPU is linked to other GPUs in the same server via a 600GB/s NVLink and to GPUs in other servers via a 100Gbps RDMA network. In production, Megatron and DeepSpeed are the two dominant frameworks, NCCL [36] is the dominant CCL.

2.2 Demands for a Unified Simulator

The rapid evolution of LLMs necessitates advancements in AI training infrastructure and optimization methods. To address these challenges, simulations are crucial for three primary goals:

Comprehensive evaluation of AI infrastructure. To ensure the effective deployment of new hardware and configurations, AI infrastructure must be evaluated from multiple perspectives:

GPU Selection: Before adopting new GPU models, cloud service providers (CSPs) need to evaluate their performance on AI workloads at scale. Simulations allow for a detailed

preview of how different GPUs perform in large-scale deployments, guiding purchasing decisions and scaling strategies.

Network Architecture Design: Once specific GPUs and intra-host interconnects are chosen, the next challenge is optimizing network architecture for scalability. Key questions, such as the ideal network bandwidth per GPU, require high-precision simulations since large-scale experiments are cost-prohibitive.

Host Architecture Design: Evaluating different host configurations for each type of GPU is essential to determine the optimal number of GPUs per host and the best intra-host interconnect. Simulations help answer these questions without the need to physically build and test expensive hardware prototypes.

Cost-effective validation of optimizations. In addition to hardware evaluations, simulations are indispensable for validating new optimization techniques during model development and system upgrades. This involves:

Parameter Tuning: Testing a variety of model parameters and training framework settings is critical to achieving optimal performance. Simulations allow for quick and inexpensive parameter tuning.

Evaluating New Mechanisms: As innovative enhancements—such as new training frameworks, collective communication methods, and network congestion control algorithms—are introduced, simulations provide a low-cost method for evaluating their effectiveness in realistic settings. This ensures that new mechanisms can be thoroughly tested without the expense of large-scale deployments.

Development of a unified simulation framework. Given the diverse needs for simulations across different components and layers of the AI infrastructure, a unified simulation framework is essential. Building multiple specialized simulators for individual parts of the system often results in inconsistent or incomplete insights. For example, a simulator focused solely on GPUs may overlook critical aspects of network performance, leading to inaccurate conclusions.

Our goal is to develop a unified simulator that addresses all these requirements in a single platform. This unified approach will enable consistent, high-precision simulations across different layers of the AI infrastructure, ensuring that teams can validate new designs and optimizations accurately and efficiently.

2.3 Our Goals

Generating workloads that reflect real-world training. To achieve accurate simulation results, realistic input sources—capturing the detailed behaviors of training frameworks—are essential. Simply estimating workload based on the required floating-point operations is too coarse-grained. Some approaches, like Chakra [53], improve this by using trace-driven methods to extract function-level data from PyTorch Execution Trace. However, this only works for LLMs with the same parameters and scale, limiting the ability to simulate new models or configurations.

Goal 1: We need a flexible and precise workload generator that can handle various models, parameters, and scales.

High-fidelity communication simulation. Classical network simulators, such as NS-3 [47] and OMNET++ [55], offer packet-level network behavior simulations but don't address the collective communication used in distributed LLM training. To maximize performance, collective communication libraries (e.g., NCCL) apply various optimizations that affect traffic patterns. Simulating these from scratch can lead to low fidelity.

Goal 2: We need a high-precision collective communication simulator that incorporates key optimizations and enhancements.

High-fidelity computation simulation. Current solutions like GPGPU-Sim [2] simulate GPU kernel computations at a detailed level but are too time-consuming for large-scale LLM simulations. Other approaches, such as ASTRA-sim [45], fail to support different GPUs or lack the necessary precision.

Goal 3: We need an efficient computation simulator that delivers both precision and scalability for large-scale simulations.

Fast simulation speed. Using a combination of current methods (*i.e.*, PyTorch trace generator with ASTRA-sim), simulating a single iteration of GPT-3 training with 128 GPUs can take an entire day, while the same task on real hardware takes just two seconds. Efficiency is critical to scale simulations for practical use.

Goal 4: The simulator must not only meet Goals 1-3 but also be scalable and capable of running large-scale LLM simulations efficiently.

3 The SimAI Simulator

3.1 SimAI Overview

Figure 1 illustrates the key components of SimAI. Each simulation request includes detailed information about the training process, such as the model itself and parameters, training framework configurations, CCL parameters, and the intra/inter-host network topology.

Workload Generator (SimAI-WG) generates realistic workloads for each simulation request (§3.2). The output, called a workload description file, outlines algorithm modules, collective communication operations, and their dependencies.

The workload file is then processed by the **Execution Engine**, which simulates the execution of both computation and communication operations as discrete events. We utilize the **Computation Simulator** (SimAI-CP) and the **Communication Simulator** (SimAI-CM) to simulate computation and

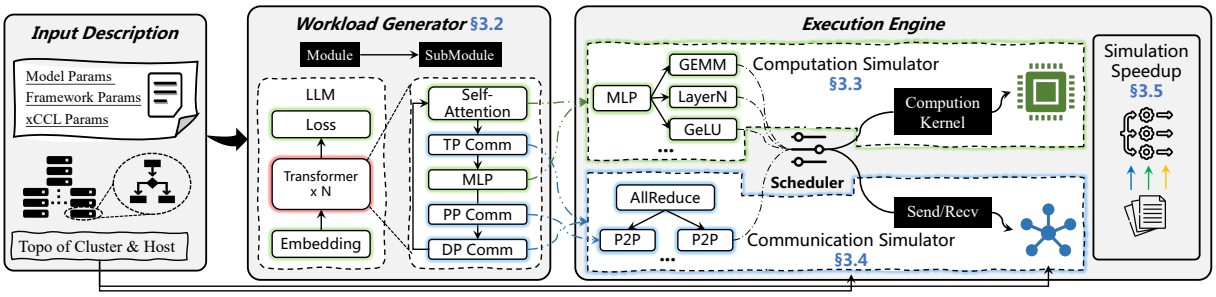


Figure 1: The SimAI architecture

communication tasks, respectively. SimAI-CP transforms submodules into detailed kernels, providing precise computation simulations using a self-built, fine-grained operation library (§3.3). SimAI-CM integrates parts of NCCL, breaking down each collective communication into peer-to-peer operations to deliver accurate communication simulation results (§3.4). Additionally, we implement multi-threaded acceleration and lock-free global context sharing to boost simulation speed further (§3.5).

3.2 Workload Generator

§3.2.1 outlines the process for generating precise workload files for a given model. To further maximize the benefits for real-world training scenarios, selecting a representative benchmark suite is crucial, as discussed in §3.2.2.

3.2.1 Generating workload files

Generating precise workload files is essential, as even small discrepancies between the workload file and actual training execution can lead to low simulation fidelity. While trace-based methods are highly accurate, they require physical clusters to run. SimAI-WG takes a different approach by "hijacking" existing training frameworks to produce workloads identical to real tasks. We have implemented this for Megatron and DeepSpeed, two of the most popular frameworks in our production environment.

Generating workloads using a single host. Without access to a large-scale GPU cluster, the key challenge for this "hijacking" approach is running it on a single host while considering the interactions of multiple peer hosts. To address this, we made two modifications to the framework and NCCL:

- The framework is tricked into believing it runs in a cluster with the target number of GPUs. The inter-host topology is also set to simulate a universal cluster configuration.
- All real communication in NCCL is skipped. For workloads involving pipeline parallelism, SimAI-WG must be configured with the appropriate rank number.

This allows the training framework to generate sequences of computation and communication operations, including algorithm submodules and collective or peer-to-peer communications. Table 1 provides examples of the typical operations used in LLMs. However, these operations lack dependency specifications, which need further specifications.

	Submodules	Kernels
Algorithm submodules & kernels	Grad_gather	Grad_gather
	Embedding	Embedding★
	Attention_forward	Layernorm★
		Attention_QKV★
		Core_attention△
		Attention_linear★
	Attention_backward	Attention_backward
	Mlp_forward	Layernorm★
		Mlp_linear_4h△
		GeLU/Swiglu★
		Mlp_linear_h△
	Mlp_backward	Mlp_backward
	Layernorm_post	Layernorm_post★
	Logits_parallel	Logits_parallel△
	Grad_param	Grad_param★
Collective & peer-to-peer communication operations	Allreduce	Write & write_with_imm operations
	Allgather	Write & write_with_imm
	Reducescatter	Write & write_with_imm
	Send	Send
	Recv	Recv

Table 1: List of computation and communication operations. ★ denotes memory-bandwidth-intensive operations and △ denotes computation-intensive operations.

Defining operation dependencies. Since computation and communication operations overlap during execution, we embed dependency information in the workload file to reflect this. Dependencies are determined based on the parallelization framework used and are validated using Nvidia Nsight Systems on a 1,024-GPU cluster. Figure 2 illustrates an example of dependencies when using TP, DP, and PP parallelisms. Solid arrows indicate a strict "happened-before" relationship, while dotted arrows represent overlapping operations where communication begins after the computation has started. The figure demonstrates how Megatron's attention and MLP backward phases benefit from overlapping optimizations. For example, the *MLPbackward* submodule starts only after the *Attentionbackward* submodule and the `AllReduce` operation are complete.

We decoupled the workload from the training framework for independent simulations. Instead of embedding simulation code directly into frameworks like Owl [9], we used SimAI-

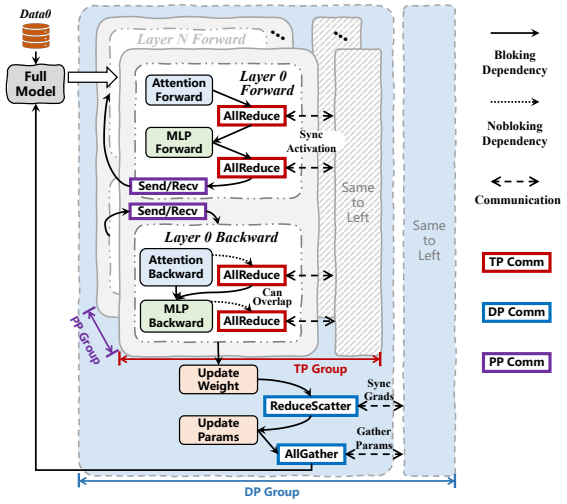


Figure 2: Demonstration of the dependencies among the algorithm submodules and the communication operations.

Size of model parameters	Percentage
100B ~ 500B	41.10%
65B ~ 75B	35.50%
7B ~ 34B	23.40%
Task scale (#GPUs)	**Percentage**
< 512	17.60%
512 ~ 1024	47.05%
> 1024	35.29%

Table 2: Proportions of tasks with different parameter sizes and scales.

WG to extract model workloads and simulate them within the SimAI execution engine. This approach avoids the substantial overhead of modifying training frameworks, which requires extensive changes to monitoring tools, data pipelines, and other critical components. By implementing core algorithms in SimAI-WG, we gain flexibility while understanding how different frameworks generate communication and computation operators.

Communication in large language models. In LLM training, metadata exchanges and barrier operation (under 1 KB) are negligible and not a focus of our simulations. Communication time is primarily introduced by various parallelism techniques, such as TP, PP, EP, and DP. TP, PP, and EP typically have fixed communication patterns and volumes, unaffected by cluster size, involving medium-sized messages ranging from tens to hundreds of megabytes. In contrast, the communication scope of DP expands with the increased cluster size, often involving gigabyte-scale collective communications (such as `AllGather` / `ReduceScatter`) across hundreds to thousands of nodes.

3.2.2 Determining the model-level benchmark suite

As a unified simulator designed to provide comprehensive evaluations of end-to-end performance in real-world LLM training scenarios, it's essential to establish a series of model-level benchmarks, referred to as the benchmark suite.

Model and framework selections. Table 2 shows the distribution of model parameters and task scales over the past six months. The models fall into three categories—small, medium, and large—each making up over 20% of the total. More than 94% of LLMs are variants of GPT-3 or LLaMA, with Megatron and DeepSpeed being the dominant frameworks. Although Megatron usage is on the rise, covering all models is impractical. Instead, we include a selection of open-source LLMs of varying sizes.

Parameter selections. We focus on the parameters that impact workload patterns:

- Model Parameters (e.g., hidden_size, num_layers, seq_len, etc.)
- Framework Parameters (e.g., world size, zero level, reduce_bucket_size/allgather_bucket_size, parallelization strategies like TP, PP, DP, and SP)

Based on the statistics in Table 2, we chose a minimal set of benchmarks that cover typical settings without exploring all possible combinations, as detailed in Table 3. Internal reports indicate that model architects frequently adjust framework parameters to optimize performance or precision. Users can also generate custom benchmarks with SimAI-WG as explained in §3.2.1.

We believe this benchmark suite accurately represents the real LLM training workloads commonly used by our customers. Unless otherwise specified, the benchmarks in this suite will be used for evaluation and discussion throughout the rest of the paper.

3.3 Precise Computation Simulation

Precisely simulating existing GPUs. As explained in § 3.2, by running the mocked framework on a single host, we obtain the detailed submodule workflow. For simulations targeting existing GPUs, SimAI-WG outputs the execution times for all submodules on a host with the corresponding GPU. Since each GPU in practical LLM training is dedicated to a single task, we can accurately simulate the entire computation procedure by following the workload file and filling in the execution times for each submodule. As shown in §4.3, the simulation precision ranges from 96.9% to 99.5%.

We maintain an operation database in SimAI-CP that records the execution times for all submodules in the benchmark suite. Table 4 lists common computation operators and their execution times on various GPUs and configurations. For workloads outside the benchmark suite, specific GPU tests are required to gather additional data.

Fine-grained kernel simulation. While submodule-level simulation works in many cases, it may not be suitable for all scenarios, especially when new parallelization strategies or optimizations reorganize or refine kernels for better performance. In such cases, fine-grained simulation is necessary.

Model		Model_hyperparameter				Framework	Parallel Parameter		Ds_config					
Name	Parameter size	Hidden size	Layers	Sequence length	FFN size	Name	TP	PP	Zero level	Reduce bucket size	Allgather bucket size	Prefetch bucket size	Max live parameters	Param persistence threshold
GPT-3 6.7B	6.7B	4096	32	2048	16384	Megatron	1	1	-	-	-	-	-	-
GPT-3 13B	13B	5120	40	2048	20480	Megatron	2	1	-	-	-	-	-	-
GPT-3 175B	175B	12288	96	2048	49152	Megatron	8	8	-	-	-	-	-	-
LLaMA 65B	65.2B	8192	80	4096	28672	Megatron	8	2	-	-	-	-	-	-
Llama3 405B	405B	16384	126	8192	53248	Megatron	8	16	-	-	-	-	-	-
LLaMA 7B	6.7B	4096	32	4096	11008	Deepspeed	1	-	2	1.00E+09	1.00E+09	-	-	-
LLaMA 65B	65.2B	8192	80	4096	28672	Deepspeed	1	-	3	1.00E+09	-	1.00E+09	6.00E+08	1.00E+06

Table 3: The SimAI benchmark suite v1.0

Operations / GPUs_ConfigParams	Embedding	LayerNorm	...	MLP-Linear
A100_Params1	3425(μs)	715(μs)	...	...
A100_Params2	3157(μs)	695(μs)	...	...
...	...	...	...	...

Table 4: SimAI-CP database format

We have designed a module-kernel converter to break down each submodule into smaller kernels, which are then tested on different GPUs. The third column in Table 1 shows an example of kernels used in different submodules. This further enriches the operation database, enabling precise simulation of advanced optimizations and new features.

Supporting unreleased GPUs. As a simulation service for CSPs, there is strong demand for simulating GPUs that have not yet been released. Decision-makers need to evaluate whether purchasing new GPUs is worth the investment, considering factors like budget, host architecture, and network architecture.

Although physical access to unreleased GPUs is not possible, we may have access to core specifications or a specification sheet. An initial approach might be to estimate computation times by scaling known values from existing GPUs, but this often results in significant inaccuracies—up to 25.1% deviation.

Our analysis shows that different kernels have different performance bottlenecks, typically falling into two categories: computation-intensive or memory-bandwidth-intensive. For example, the Gemm kernel, used for updating the KV cache, is memory-bandwidth-intensive, while flash attention is computation-intensive. Detailed classifications are listed in Table 1.

To improve accuracy, we propose using two equations tailored to these kernel types, based on data from our operation database and the Roofline performance model [58]. These equations, based on measured execution times of compute-bound and memory-bound kernels (subscript Comp_Known and Mem_Known, respectively) in the existing environment, allow us to calculate execution times for these kernels on new GPUs or configurations (subscript Comp_New and Mem_New, respectively):

- For computation-intensive kernels:

$$\text{Time}_{\text{Comp_New}} = \frac{\text{FLOPS}_{\text{Comp_New}}}{\text{FLOPS}_{\text{Comp_Known}}} \times \text{Time}_{\text{Comp_Known}}$$

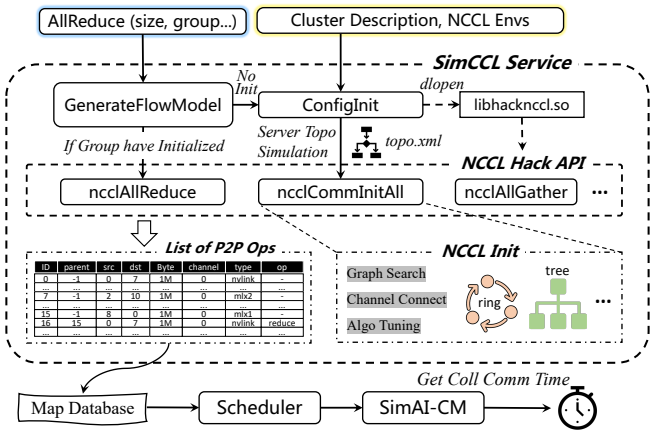


Figure 3: Illustration of how SimAI transforms a collective communication operation to a list of peer-to-peer communication operations.

- For memory-bandwidth-intensive kernels:

$$\text{Time}_{\text{Mem_New}} = \frac{\text{Bandwidth}_{\text{Mem_New}}}{\text{Bandwidth}_{\text{Mem_Known}}} \times \text{Time}_{\text{Mem_Known}}$$

To ensure accurate results, we recommend using GPUs with similar architectures as baselines, rather than generalizing across different vendors. In our experiments, we use the Nvidia A100 as the baseline for calculating kernel execution times on other Nvidia GPUs. We refer to this method as SimAI-CP-Model.

3.4 Precise Simulation of Communication

Training frameworks rely on collective communication libraries, with NCCL [36] being the most widely used in production. NCCL translates collective communication operations (e.g., `AllReduce`, `AllGather`, `ReduceScatter`) into network-level operations (e.g., `Send`, `Receive`). This complex process involves selecting optimal algorithms based on factors like the number of nodes, message size, and configured parameters. Even small mismatches between practical execution and simulation can cause significant deviations in results. To accurately reproduce these algorithm selections, SimAI integrates key procedures from NCCL directly.

Reproducing NCCL's key procedures. SimAI-CM uses a modified version of NCCL, called SimCCL, to intercept

key operations. SimCCL captures the initialization and core interfaces of collective communication to generate peer-to-peer communication lists.

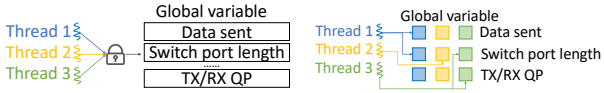
Since simulations typically run on a single host, SimCCL employs a "hijacking" technique to simulate the detailed peer-to-peer operations, similar to the approach discussed in §3.2.1, but at the collective communication level. Here's how SimCCL modifies NCCL's behavior:

1. *NCCL Initialization*: SimCCL intercepts the *NcclCommInitAll* function using the libhacknccl.so in Figure 3, creating virtual communicators for each GPU. This makes the system behave as though it's running in a real multi-GPU cluster, allowing for socket connections and data exchanges during the initialization phase. In the bootstrap and network initialization stages, multiple virtual communicators are created in the same communication group, while only one communicator is actually created.
2. *Topology Discovery*: Instead of searching actual PCIe devices, SimCCL reads a user-specified topology file that defines GPU, NIC, and PCIe switch configurations. Each virtual communicator processes the topology independently and no synchronization is required.
3. *Intra-Host Communication Channel Creation*: SimCCL sets up channels between virtual communicators within the host and stores the details in an isolated information table.
4. *Inter-Host Communication Channel Creation*: SimCCL bypasses gathering information from other GPUs using `AllGather` operations, as it already has information on all hosts. It creates inter-host channels directly.
5. *Collective Communication Transformation*: SimCCL intercepts collective communication calls, reconstructing the operations to trace lower-level communications. It skips actual data transfers and captures inter-GPU communication events, including data size, sender and receiver ranks, and routes, to simulate RDMA-layer behavior.

Supporting all NCCL parameters. SimCCL reflects the communication behavior for the vast majority of NCCL parameters [37]. SimAI-CM has been enhanced to support specific features like PCI × NVLink (PXN). PXN allows a GPU to use a non-local NIC (in the same node) through NVLINK connections for data transfers. This design is typically employed in rail-optimized network topologies, enabling cross-node network traffic to remain on the same rail (single-hop switch) to achieve message aggregation and network traffic optimization. By setting an appropriate NCCL_P2P_PXN_LEVEL, SimCCL can recognize these PXN traffic patterns and reflect them in the output FlowModel. For instance, if rank 1 sends data to rank 8 via rank 0 as an intermediary, the FlowModel represents this as two separate flows: 1->0 and 0->8. Subsequently, SimAI-CM extracts these patterns and simulates them accordingly.

Source file	Description	Lines of code
init.cc	The initialization procedures	364
topo.cc	The topology discovery, search, and synchronization	52
topo.xml	A custmoized topology file	~100
enqueue.cc	Chunk segmentation and task scheduling of collective communication	56

Table 5: Efforts of modifying NCCL.



(a) Multi-thread implementation. (b) Lock-free optimization

Figure 4: Multi-thread implementation and optimization in SimAI.

The SimCCL module is designed to emulate the communication operation processing workflow in NCCL or other CCL. It focuses on transforming collective communication operations into a set of easily interpretable point-to-point communications, without incorporating actual data verification or integrity checks. This approach generally does not affect the end-to-end training process, as it doesn't lead to exceptional workflows due to the absence of real data. In expert parallelism (EP), the gating module's token distribution is influenced by data values. In the simulation, we assume a balanced distribution, which has minimal impact on the simulation results.

Porting efforts. It took us over 10,000 lines of coding efforts for building the blocks shown in Figure 1. The majority of the efforts are intended for repeated use except for the SimCCL module. For example, to support a different version of NCCL or a new CCL, we need to re-adapt the SimCCL module. However, our design does not incur intrusive modifications on the original CCL codebase. As shown in Table 5, only 572 lines of codes (LOC) are essential based on the original NCCL codebase.

3.5 Large Scale Simulation Speedup

At the beginning of SimAI's development, simulating a single iteration of GPT-3 training with 128 GPUs took over 24 hours, compared to just two seconds on a physical GPU cluster. This challenge is similar to AstraSim, which also uses NS-3 for network simulation. To enable simulations of AI infrastructure with more than 1,000 GPUs, we implemented multi-threaded acceleration for SimAI-CM.

Several approaches have been proposed to speed up network simulations. For example, parallel discrete-event simulation (PDES) [10, 18] and UNISON [3] distribute network topologies across multiple logical processes, each running on separate CPU cores. We chose UNISON [3] for three key reasons: (1) It is open-source and builds on NS-3. (2) It automates the partitioning of network topologies and schedules each task to the appropriate thread. (3) It has superior scalability, as demonstrated in their evaluations.

Lock-free sharing of global variables. However, integrating UNISON into SimAI presented a significant challenge. As

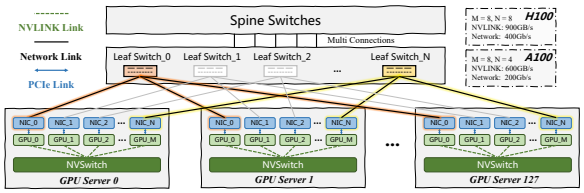


Figure 5: Multi-rail network topology of cluster H100. For cluster A100, two GPUs share a single NIC

the simulation scale increased, a large number of global configurations and contexts were shared across threads. Updating shared data structures with global locks, even atomic ones, caused performance bottlenecks. As shown in Figure 4(a), these global variables track metadata on inter-host communications, such as data volumes between nodes and queue lengths at switches. Atomic operations on these variables slowed down concurrent thread execution.

To solve this, we restructured SimAI to run without global locks. We observed that most global variables were accessed by specific threads. By managing these variables in a thread-independent way, we eliminated the need for global locks. Figure 4(b) illustrates how SimAI separates these variables, enabling lock-free operation. Since each node's simulation runs on a single thread, we isolate global variables in a node ID-indexed table. This allows threads to access the relevant data without locking, reducing the performance issues caused by concurrency. Our lock-free optimization resulted in a 23x speedup compared to the single-thread version and a 15% improvement over the multi-threaded version.

4 Evaluation

In this section, we evaluated SimAI based on the precision and scalability of simulations, using the open-source simulator ASTRA-sim for comparison. Furthermore, results from real-world clusters are used as the ground truth.

4.1 Testbeds and Benchmarks

We utilized two testbeds that share a similar multi-rail, fat-tree, inter-host RoCEv2 network, reminiscent of the Alibaba HPN architecture [44], as illustrated in Figure 5. Cluster A100 consisted of 128 GPU hosts, each equipped with eight NVIDIA A100 GPUs [1] and four Mellanox ConnectX-6 NICs (2×100 Gbps). Cluster H100 included 128 GPU servers, each with eight NVIDIA H100 GPUs [13] and eight Mellanox ConnectX-7 NICs (2×200 Gbps). The intra-host interconnect used NVLink, with bandwidths of 600 GBps for A100 and 900 GBps for H100. Both the simulations and physical clusters shared identical intra-host and inter-host network topologies.

The benchmarks were derived from the SimAI benchmark suite. We enhanced ASTRA-sim with a workload adapter that parsed workload files generated by SimAI-WG. For the physical GPU cluster, the benchmarks involved real LLM tasks that matched the model, framework, and NCCL parameters of the workload files. Our evaluation spanned cluster sizes of 128, 512, and 1024 GPUs. We executed all workloads, but due to space constraints, we only presented performance metrics for three representative models: GPT-3 13B, LLaMA 65B, and GPT-3 175B.

4.2 Precision of Communication Simulation

Figures 6 and 7 showed the performance of intra-host and inter-host collective communication operations across different message sizes (ranging from 8MB to 8GB) and cluster scales. In a multi-rail architecture [59], a collective communication group consists of the ranks on the same rail to minimize the latency across ranks and achieve optimal performance.

From Figure 6, we observed that SimAI was significantly more accurate than ASTRA-sim and closely aligned with the ground truth. The average deviations for SimAI and ASTRA-sim were 3.9% and 74.8% for A100, and 2.3% and 51.7% for H100, respectively. Notably, for `AllGather` and `ReduceScatter` operations on A100, ASTRA-sim achieved its best performance with 8GB messages, yet diverged from reality by over 21.8%, with error rates reaching up to 210.5% under adverse conditions. While there was slight improvement in the `AllGather` operation, ASTRA-sim still showed deviations of 204.3% and 35% for 8MB and 8GB messages, respectively. This discrepancy arose because small message performance is more susceptible to inaccuracies, as we did not simulate the runtime software (e.g., libibverbs) and the NIC pipelines. ASTRA-sim made significant simplifications in practical transmission procedures, leading to poor fidelity, particularly in small message scenarios.

Figure 7 presented the outcomes of collective communication between machines, reinforcing the finding that ASTRA-sim struggled to accurately simulate communications with small message sizes, while SimAI remained consistently close to the ground truth. Furthermore, as communication scaled up, the divergence between ASTRA-sim's simulations and the ground truth widened. For example, in an `AllGather` operation with 8MB messages on a cluster of 128 A100 GPUs, ASTRA-sim exhibited a simulation timing error of 45.9%. This error escalated to 530.2% when the scale increased to 512 GPUs. The results for `AllReduce` and `ReduceScatter` operations displayed similar trends as those observed with the `AllGather` operation.

4.3 Precision of Computation Simulation

We tested the precision of the computation simulation of SimAI and compared it with the ground truth of Nvidia A100, H100, and H20. As shown in Figure 8, SimAI-CP provided highly accurate estimations for all computation kernels. The gaps in total execution times were minimal (0.5%-3.1%) between SimAI-CP and the ground truth. However, for SimAI-CP-Model, the deviations were larger, ranging from 13%-15%, especially for the $Attention-QKV$ and $MLP-linear$ kernels. We recommend that users utilize SimAI-CP-Model only in scenarios where a GPU is not available.

Additionally, we were unable to run all computation kernels

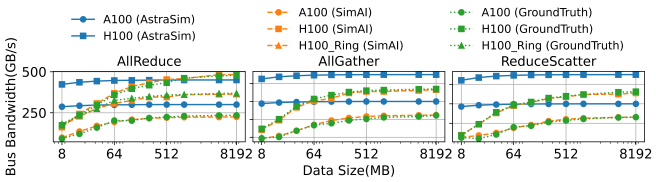


Figure 6: Bus bandwidths of intra-host collective communication operations vary with different message sizes and GPU types. Note that the performance is not related to the cluster scales.

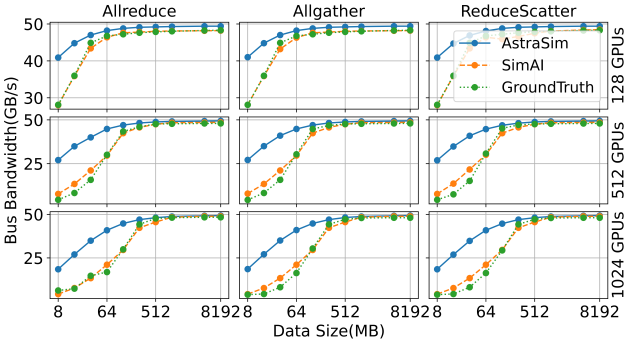


Figure 7: Bus bandwidths of inter-host collective communication operations with different message sizes and cluster scales.

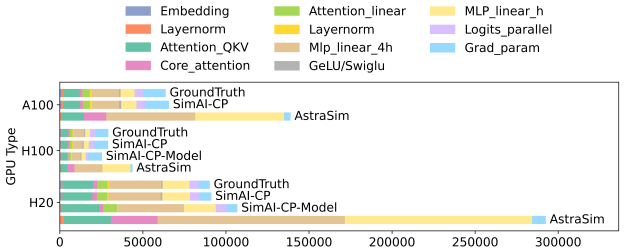


Figure 8: The execution time of the computation kernels of GPT-3 175B on different GPUs.

in SCALE-sim [50], which is used by ASTRA-sim. Only the matrix multiplication kernels ran successfully in SCALE-sim. The numbers for ASTRA-sim in Figure 8 included only these kernels, with execution times multiplied by the effective FLOPS of Nvidia GPUs divided by the effective FLOPS of TPUs. We observed that ASTRA-sim had a significant margin of error in simulating other GPUs, with total errors of 49.8% for H100, 117.9% for A100, and 224% for H20. As discussed in §3.3, the closed internal nature of the GPU computation pipeline caused these methods to fall short in precisely estimating computation times.

Moreover, we attempted to use another computation simulator, Accel-Sim [21], but ultimately failed to run the latest LLM tasks as it only supports PyTorch 0.4, which was released six years ago.

4.4 Precision of End-to-end Simulation

We validated the precision of SimAI in terms of the end-to-end performance on different cluster scales. Figure 9 shows

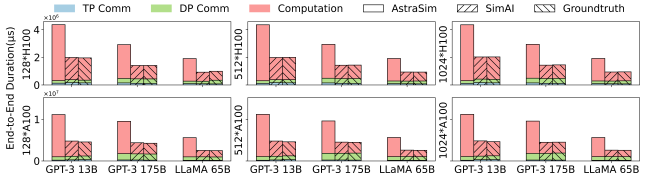


Figure 9: The iteration time of different workloads running on ASTRA-sim, SimAI, and real clusters.

that using SimAI the iteration time of all the workloads is quite close to that of real LLM tasks in real clusters. The gap is less than 4% even under the scale of 1024 GPUs. However, ASTRA-sim is imprecise in the iteration time for both A100 and H100 GPUs. The deviations between SimAI and ground truth are less than 3.9%, 36.1× better than ASTRA-sim. Furthermore, with the increase of the model size (i.e., from GPT-3 13B to GPT-3 175B), the precision of SimAI increases accordingly. It is because the message sizes of the collective communication operations of GPT-3 175B are larger, whose simulation is more precise as validated in §4.2.

5 In Production Benefits

5.1 Guiding Host Design for New GPUs

H100. In 2023, hosts equipped with A100 GPUs were designed to include eight A100 GPUs and four 2×100Gbps NICs (Mellanox-CX6), allowing each GPU to utilize 100Gbps of network bandwidth. With the release of the H100 at GTC 2023 [12], the performance of a single GPU significantly improved. This raises an important question: How much network bandwidth is needed to (1) fully utilize the enhanced performance of the H100 while (2) minimizing overall capital expenditure?

Answering this question is challenging because building multiple engineering prototype hosts with varying configurations is costly and time-consuming. Simply expanding network capacity in proportion to the increase in FLOPS is too simplistic to provide practical guidance.

To inform server design decisions, we utilized SimAI to simulate the performance of a training cluster with 1,024 H100 GPUs and varying network capacities. We tested network equipment configurations of 100Gbps, 200Gbps, and 400Gbps. As shown in Figure 10, the performance of the H100 cluster improves with increased network bandwidth, aligning with our expectations. Upgrading from 200Gbps to 400Gbps results in a 19% performance gain, which remains significant relative to the associated capital expenditure. These simulation results contributed to our final host design decision.

We did not simulate configurations with higher bandwidth than 400Gbps because the PCIe bandwidth of the H100 is 512Gbps (PCIe Gen5×16), indicating that the practical upper limit for a single H100 is approximately 400Gbps.

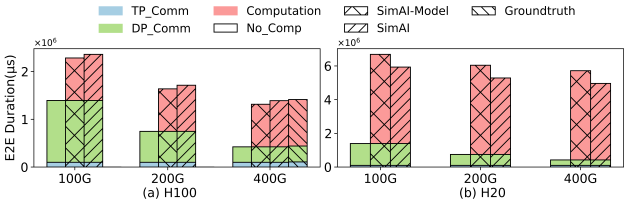


Figure 10: The iteration time of GPT-3 175B with different network bandwidths.

In production, hosts with H100 GPUs are designed to feature eight H100s and eight 2×200Gbps NICs (Mellanox-CX7). Comparatively, the simulation precision using SimAI-CP is over 98%, while SimAI-CP-model shows a precision that is more than 5% lower. Thus, we believe SimAI-CP-model could serve as a viable option when the target GPU type has not yet been released.

H20. With the release of H20 [14] in November 2023, we again used SimAI to guide host design. As shown in Figure 10, since the computational performance of H20 is lower than that of H100, the required network bandwidth also decreases. When network bandwidth is increased from 100Gbps to 200Gbps, end-to-end training performance improves by 11%. However, increasing from 200Gbps to 400Gbps yields only a 6% performance gain.

After thoroughly assessing the investment in networking equipment against performance improvements, we decided to equip each H20 with 200Gbps of network bandwidth (i.e., eight H20 GPUs and eight CX6 NICs per host). Additionally, using SimAI-CP-Model for computation simulation yielded similar conclusions, although with lower precision. This indicates that even for out-of-stock GPU devices where SimAI-CP cannot be used, SimAI can still provide meaningful results.

5.2 Quantifying Scaling-up Benefits

During the training of large language models (LLMs) in production, end-to-end performance is significantly influenced by model and framework parameters, particularly parallelization strategies. Using brute force to explore all parameter combinations in practical training clusters can result in considerable overhead. However, with SimAI, we can automatically identify optimal parameter settings.

In collaboration with the LLM team and the training optimization team, we conducted comprehensive simulations across various parameters. One surprising finding relates to the tensor parallelism (TP) configuration. Conventional wisdom suggests that using more GPUs in a single TP group can reduce the memory and computation load per GPU but may increase communication overhead. Typically, production practices set the TP group size to match the number of GPUs within a host connected via NVSwitch, yet this lacks quantitative analysis.

To understand the trade-offs associated with this parameter setting, we performed experiments within SimAI using GPU counts of 8, 16, 32, and 64, all interconnected with NVSwitch. For each configuration, we ran training tasks on three different models, aiming to identify performance trends and optimal setups. Figure 11 illustrates the following key points:

1. In an 8-GPU host, the optimal TP sizes for GPT-3 13B, LLaMA 65B, and GPT-3 175B are 4, 8, and 8, respectively.
2. For any specific TP size, performance improves with more GPUs in the host.
3. Even with a larger GPU count, performance may decline if the TP size is inappropriate.

These findings offer two valuable guidelines for practical LLM training:

1. The TP parameter should accommodate the entire model layer. However, increasing the TP size can reduce end-to-end throughput; a better strategy is to set up more data parallel (DP) groups for parallel training.
2. GPU host design must consider the evolving nature of LLMs. If a layer's maximum size is known, the number of GPUs within a host should be determined accordingly, prioritizing enhanced scale-out performance.

6 Experience

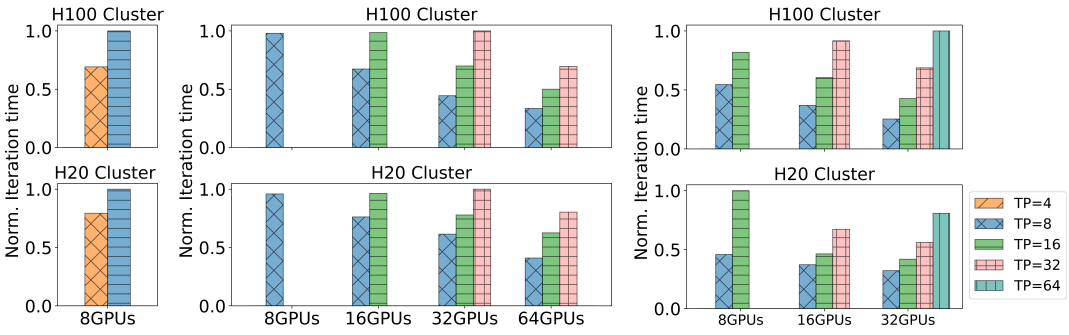
6.1 Evolution of Simulator

Maintaining precision from the very beginning. Initially, we used ASTRA-sim as the foundation for our simulator prototype. Our first challenge was to assess the training performance of emerging large language models (LLMs) like GPT-3 and LLaMA. However, suitable traces or workloads for simulating these models were unavailable, and the built-in DNN [56] workload in ASTRA-sim only provided a rough approximation, leading to significant discrepancies in simulations—ranging from 92.1% to 143.9% across different testbeds. This made it impractical to base our simulation model on the built-in workload.

To ensure accurate LLM workload simulations, we decided to build SimAI-WG from the ground up. We began by analyzing the open-source code of frameworks like Megatron [52] and DeepSpeed [46] to reconstruct their workflows in our simulator. While this improved precision, it had significant drawbacks: (1) It demanded extensive human resources, requiring our engineers to thoroughly understand each framework's components and execution details, which could take months. (2) Keeping our models updated proved challenging due to the rapid evolution of training frameworks, necessitating continuous engineering efforts without any automated update systems.

As a result, we abandoned this approach after our initial internal release and introduced the mock-framework method detailed in this paper. By accurately replicating all sub-modules and their relationships, we achieved substantial improvements in simulation precision.

Measurements from our cluster show that end-to-end training times fluctuate due to factors like network transmission



(a) GPT-3 13B. (b) GPT-3 175B. (c) LLaMA 65B.

Figure 11: Training performance of different hosts and different TP sizes. The x-axis denotes the number of GPUs in a single host. Because of the big model size and limited GPU memory, TP should be larger than 2 for GPT-3 13B and 4 for GPT-3 175B and LLaMA 65B. And we can not run TP = 64 or TP = 48 for GPT-3 175B because the TP size should be the factor of both the number of feedforward neural network (FNN) layers (96) and the world size (1024). We do not let TP exceed the number of GPUs in a single host as the RDMA network is the bottleneck.

and hardware status. Workflow tracking indicates these fluctuations are limited to 5%, with communication time accounting for only 30% of the total. As communication fluctuations contribute negligibly to overall variance, we input averaged parameter values into the SimAI-CP and SimAI-CM components to achieve more consistent performance estimates.

Enhancing computation simulation precision. With a precise workload as input, we still faced a considerable gap between practical execution and simulation results. Early attempts to derive execution times from theoretical models yielded low precision. Additionally, our GPU-acceleration team developed a fine-grained GPU simulator that provides detailed simulation timings for specific operations, but integrating it into our system resulted in excessive simulation latency.

To address this, we focused on the key factors influencing execution times and developed SimAI-CP. Our operation library construction unfolded in two phases. Initially, we extracted key sub-modules from LLMs and measured their execution times, achieving satisfactory precision. However, this approach was limited to specific GPU types. In the second phase, we split sub-modules into finer kernels categorized by computation and memory access intensity. This allowed us to estimate operation times on unavailable GPUs based on performance specifications, enhancing the simulator's applicability.

Improving collective communication simulation precision. We also encountered significant inaccuracies in collective communication simulations when using ASTRA-sim. A critical issue was the inconsistency between the collective communication algorithms in ASTRA-sim and those in NCCL. Even after implementing NCCL's algorithms, results remained unsatisfactory, prompting us to forgo optimizing ASTRA-sim and instead mock key NCCL behaviors from scratch.

The improvements in computation and communication simulations progressed concurrently. When both achieved optimal outcomes, we combined SimAI-CP and SimAI-CM, resulting in a simulator capable of simulating model training with over 93% precision.

Speeding up the entire simulator. After achieving high precision, our final challenge was ensuring efficient simulation execution. Built on NS-3, SimAI underwent a thorough review of existing acceleration solutions, ultimately integrating UNISON [3]. Although we encountered context-access conflicts and crashes during direct operation, implementing global locks to manage inter-thread context access reduced performance. We then analyzed context dependencies and implemented lock-free access, achieving up to a 23× performance improvement, making the simulator practically usable.

6.2 Simulation as a Service

The ultimate goal of developing this simulator is to meet the simulation needs of various teams, transforming it into an easy-to-use simulation service.

Running without GPUs. To achieve high simulation precision, we initially integrated original processing codes (e.g., PyTorch and NCCL) into SimAI. However, this required servers with specific GPUs and environments. To create a genuine simulation service that operates on any host, we decoupled SimAI from the GPU software stack, recognizing and hooking all CUDA-related functions.

Additionally, updating the operation library became an independent procedure. When a new GPU type is available, we conduct a full test of all defined operations on that GPU and update the library as needed.

Task management and scheduling. As a simulation service, SimAI must respond to concurrent simulation requests. To enhance system throughput, we maintain a cluster of hosts. Each request is assigned to an independent simulation task, scheduled on the host with the lightest workload. We use Kubernetes [29] for high scalability.

Parameter tuning. As mentioned in §5, adjusting the TP group size significantly affects end-to-end simulation results. In addition, SimAI allows fine-tuning various parameters influencing model training, including topology, model, and parallel strategy parameters. For example, modifying topology parameters alters the virtual topology within the simulation, impacting network behavior. Adjusting model parameters or parallel strategies affects the computation kernel execution time and communication volume, ultimately impacting training efficiency. Any input supported by SimAI can be fine-tuned to optimize overall performance.

Fine-grained monitoring. For each simulation, we not only generate reports detailing overall training time and throughput but also incorporate extensive monitoring statistics. We developed a front-end system to present the variations of different metrics during simulations.

7 Related work

AI infrastructure simulation. ASTRA-sim [45, 60] is the only comprehensive DNN training simulator designed to simulate the software and hardware co-design stack of distributed training systems. Dally [51] extends ASTRA-sim by incorporating simulations of modern networking hardware (e.g., NVIDIA Quantum [38] and Spectrum switches [39]). SimAI is inspired by ASTRA-sim but introduces significant improvements in workload generation, high-fidelity computation/communication simulation, and scalability.

GPU and computation simulation. Besides the roofline model [58], which provides coarse performance estimates for compute kernels or applications running on a GPU, other works offer instruction-level GPU simulations, such as GPGPU-Sim [2], Accel-Sim [22], SCALE-sim [50], and Gem5-GPU [43]. However, these simulations are very time-consuming for large-scale deployments.

Network simulators and scalability. Existing network simulators fall into three categories: discrete-event simulation (DES) (e.g., NS-3 [47], OMNeT++ [55], OPNET [28]), mathematical model estimation [24, 30, 42], and learning-based approximation (e.g., DeepQueueNet [61], MimicNet [63]). DES simulators offer high-fidelity packet-level simulation, and some works use parallel and distributed event simulation (PDES) approaches [10, 18] to accelerate these simulations. However, PDES often suffers from poor performance and complex configurations. DONS [11] employs a data-oriented design to reduce cache misses in simulation and can be automatically parallelized for intra-host and inter-host scenarios, though it currently lacks support for RDMA transports. UNISON [3] addresses this limitation with fine-grained partitioning and load-adaptive scheduling. SimAI's acceleration is based on UNISON, enhanced with lock-free sharing of global variables. In the future, SimAI will consider simulating network virtualization scenarios to evaluate various virtualization technologies (such as MasQ [15], etc.).

AI benchmarks. The recent surge in machine learning has led to the development of several benchmark suites, but many have limitations in LLM benchmarking. These suites typically include benchmarks for various application domains. MLPerf [31], ML-Bench [27], and AIBench [54] offer comprehensive machine learning benchmarks, while ParaDnn [57], DeepBench [7], and GNNMark [4] focus on deep learning and graph neural networks. However, these benchmarks only cover current applications, which may become obsolete as models evolve rapidly.

Learning-based simulator. Learning-based methods, commonly used as end-to-end performance estimators (EPE) [20, 25, 48, 49, 62, 63], improve scalability by predicting metrics like RTT and packet loss through deep learning models trained on real network traffic. However, these methods require extensive training data, creating significant overhead and limiting their applicability in large-scale simulations. In contrast, SimAI employs discrete event simulators (DES) to mimic the behavior of model training, enabling large-scale simulations and generating valuable data for training learning-based models.

Fault simulation. Fault simulation in model training is a feature worth exploring. Some existing works introduce various types of network faults in the NS-3 simulator to evaluate the impact on the network and the effectiveness of fault tolerance mechanisms [5, 8, 16, 34]. We plan to incorporate simulations for supporting model faults in future work.

8 Conclusion

In this paper, we introduce SimAI, a unified simulator designed to provide accurate and efficient simulations for large language model (LLM) training. SimAI integrates key processes from leading training frameworks and collective communication libraries, achieving an average of 98.1% alignment with real-world results across various test scenarios. To enhance scalability and performance, SimAI incorporates a lock-free, multi-threaded optimization. It is widely utilized in Alibaba Cloud production environments, where it has contributed to the development of valuable operational guidelines.

Acknowledgments

We thank our shepherd Dr. Fang Zhou and the anonymous NSDI reviewers for their constructive comments. This work was supported by the Beijing Outstanding Young Scientist Program (No. JWZQ20240101008).

References

[1] Nvidia a100 tensor core gpu. https://www.nvidia.com/en-us/data-center/a100/. [Online; accessed 4-May-2024].

[2] Bakhoda Ali, George L. Yuan, Wilson W. L. Fung, Wong Henry, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 04 2009.

[3] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. Unison: A parallel-efficient and user-transparent network simulation kernel. 2024.

[4] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L Abellán, Yash Ukidave, Ajay Joshi, John Kim, et al. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–23. IEEE, 2021.

[5] Fedor Chernogorov, Ilmari Repo, Vilho Räisänen, Timo Nihtilä, and Janne Kurjenniemi. Cognitive self-healing system for future mobile networks. In *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 628–633. IEEE, 2015.

[6] Nvidia mellanox connectx-6 smartnic adapter. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/. [Online; accessed 4-May-2024].

[7] Deepbench: benchmark operations that are important to deep learning on different hardware platforms. https://github.com/baidu-research/DeepBench. [Online; accessed 4-May-2024].

[8] Emmanuel Effah and Ousmane Thiare. Survey: Faults, fault detection and fault tolerance techniques in wireless sensor networks. *Int. J. Comput. Sci. Inf. Secur.(IJCSIS)*, 16(10):1–14, 2018.

[9] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and flexibility in distribution of hot content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, Carlsbad, CA, July 2022. USENIX Association.

[10] R.M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Series on Parallel and Distributed Computing. Wiley, 2000.

[11] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 167–181, New York, NY, USA, 2023. Association for Computing Machinery.

[12] Gtc 2023). https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s52226/. [Online; accessed 4-May-2024].

[13] Nvidia h100 tensor core gpu. https://www.nvidia.com/en-us/data-center/h100/. [Online; accessed 4-May-2024].

[14] Nvidia h20 tensor core gpu. https://developer.nvidia.com/blog/accelerating-inference-on-end-to-end-workflows-with-h2o-ai-and-nvidia/. [Online; accessed 4-May-2024].

[15] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. Masq: Rdma for virtual private cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 1–14, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Peng Hu and Jinhuan Zhang. 5g-enabled fault detection and diagnostics: How do we achieve efficiency? *IEEE Internet of Things Journal*, 7(4):3267–3281, 2020.

[17] InfiniBand Trade Association. RoCE v2 Specification. https://cw.infinibandta.org/document/dl/7781, 2014.

[18] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, 30:54–73, 01 2013.

[19] Peng Jiang, Christian Sonne, Wangliang Li, Fengqi You, and Siming You. Preventing the immense increase in the life-cycle energy and carbon footprints of llm-powered intelligent chatbots. *Engineering*, 2024.

[20] Charles W. Kazer, João Sedoc, Kelvin K.W. Ng, Vincent Liu, and Lyle H. Ungar. Fast network simulation through approximation or: How blind men can describe elephants. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 141–147, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486. IEEE, 2020.

[22] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[23] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, Renton, WA, April 2022. USENIX Association.

[24] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050. 06 2004.

[25] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. Simnet: Accurate and high-performance computer architecture simulation using deep learning. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), jun 2022.

[26] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Liang Chen, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark Gerstein. Ml-bench: Evaluating large language models for code generation in repository-level machine learning tasks, 2024.

[28] Z. Lu and H. Yang. *Unlocking the Power of OPNET Modeler*. Unlocking the Power of OPNET Modeler. Cambridge University Press, 2012.

[29] Marko Luksa. *Kubernetes in action*. Simon and Schuster, 2017.

[30] Marco Marsan, Michele Garetto, Paolo Giaccone, Emilio Leonardi, Enrico Schiattarella, and Alessandro Tarello. Using partial differential equations to model tcp mice and elephants in large ip networks. *Networking, IEEE/ACM Transactions on*, 13:1289 – 1301, 01 2006.

[31] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349, 2020.

[32] Introducing our next generation infrastructure for ai. https://about.fb.com/news/2024/04/introducing-our-next-generation-infrastructure-for-ai/. [Online; accessed 4-May-2024].

[33] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 753–766, New York, NY, USA, 2022. Association for Computing Machinery.

[34] Arslan Munir, Joseph Antoon, and Ann Gordon-Ross. Modeling and analysis of fault detection and fault tolerance in wireless sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(1):1–43, 2015.

[35] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl. [Online; accessed 4-May-2024].

[37] Environment variables of nvidia collective communications library (nccl). https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html. [Online; accessed 4-May-2024].

[38] Nvidia quantum switch. https://nvdam.widen.net/s/k8sqcr6gzb/infiniband-quantum-2-qm9700-series-datasheet-us-nvidia-1751454-r8-web. [Online; accessed 4-May-2024].

[39] Nvidia spectrum switch. https://nvdam.widen.net/s/mmvbnpk8qk/networking-ethernet-switches-sn5000-datasheet-us. [Online; accessed 4-May-2024].

[40] Nvlink and nvlink switch. https://www.nvidia.com/en-us/data-center/nvlink/. [Online; accessed 4-May-2024].

[41] OpenAI. Gpt-4 technical report, 2024.

[42] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H. Low. Multipath tcp: Analysis, design, and implementation. *IEEE/ACM Transactions on Networking*, 24(1):596–609, 2016.

[43] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.

[44] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim: Enabling sw/hw co-design exploration for distributed dl training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 81–92, 2020.

[46] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[47] George F. Riley and Thomas R. Henderson. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[48] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Unveiling the potential of graph neural networks for network modeling and optimization in sdn. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 140–151, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet: Leveraging graph neural networks for network modeling and optimization in sdn. *IEEE Journal on Selected Areas in Communications*, 38(10):2260–2270, 2020.

[50] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883*, 2018.

[51] Aakash Sharma, Vivek M. Bhasi, Sonali Singh, George Kesidis, Mahmut T. Kandemir, and Chita R. Das. Gpu cluster scheduling for network-sensitive deep learning, 2024.

[52] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[53] Srinivas Sridharan, Taekyung Heo, Louis Feng, Zhaodong Wang, Matt Bergeron, Wenyin Fu, Shengbao Zheng, Brian Coutinho, Saeed Rashidi, Changhai Man, and Tushar Krishna. Chakra: Advancing performance benchmarking and co-design using standardized execution traces, 2023.

[54] Fei Tang, Wanling Gao, Jianfeng Zhan, Chuanxin Lan, Xu Wen, Lei Wang, Chunjie Luo, Zheng Cao, Xingwang Xiong, Zihan Jiang, Tianshu Hao, Fanda Fan, Fan Zhang, Yunyou Huang, Jianan Chen, Mengjia Du, Rui Ren, Chen Zheng, Daoyi Zheng, Haoning Tang, Kunlin Zhan, Biao Wang, Defei Kong, Minghe Yu, Chongkang Tan, Huan Li, Xinhui Tian, Yatao Li, Junchao Shao, Zhenyu Wang, Xiaoyu Wang, Jiahui Dai, and Hainan Ye. Aibench training: Balanced industry-standard ai training benchmarking. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 24–35, 2021.

[55] Andras Varga. *A Practical Introduction to the OMNeT++ Simulation Framework*, pages 3–51. 05 2019.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[57] Yu Wang, Gu-Yeon Wei, and David Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. *Proceedings of Machine Learning and Systems*, 2:30–43, 2020.

[58] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[59] Noah Wolfe, Misbah Mubarak, Nikhil Jain, Jens Domke, Abhinav Bhatele, Christopher D Carothers, and Robert B Ross. Preliminary performance analysis of multi-rail fat-tree networks. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 258–261. IEEE, 2017.

[60] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 283–294, 2023.

[61] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. Deepqueuenet: towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 441–457, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. Deepqueuenet: towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 441–457, New York, NY, USA, 2022. Association for Computing Machinery.

[63] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. Mimicnet: fast performance estimates for data center networks with machine learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 287–304, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

9 Appendix

9.1 NCCL environment variables

Table 6 lists all the NCCL environment variables that documented in the official website [37] and whether they are supported in the SimAI simulator. We can see besides those we think are unnecessary for simulation, only four environment variables are not yet supported, which are adaptive routing and Infiniband SHArP related features. We leave them as the future work of the communication simulator of SimAI.

Table 6: NCCL environment variables and how they affects the communication patterns

NCCL environment variable	Description	Support?
NCCL_IB_HCA	Specifies which RDMA interfaces to use for communication	Yes
NCCL_IB_TC	Specifies the InfiniBand Traffic Class to use	Yes
NCCL_IB_QPS_PER_CONNECTION	Number of IB queue pairs to use for each connection between two ranks	Yes
NCCL_NVLS_ENABLE	Enable the use of NVLink SHARP (NVLS)	Yes
NCCL_MAX_ NCHANNELS	Sets the maximum number of NCCL channels per GPU	Yes
NCCL_MIN_ NCHANNELS	Sets the minimum number of NCCL channels per GPU	Yes
NCCL_MIN_NRINGS	Sets the minimum number of NCCL rings per GPU	Yes
NCCL_MAX_NRINGS	Sets the maximum number of NCCL rings per GPU	Yes
NCCL_CROSS_NIC	Controls whether NCCL should allow rings/trees to use different NICs, causing inter-node communication to use different NICs on different nodes	Yes
NCCL_NET_GDR_LEVEL	Allows the user to finely control when to use GPU Direct RDMA between a NIC and a GPU	Yes
NCCL_ALGO	Defines which algorithms NCCL will use	Yes
NCCL_PXN_DISABLE	Disable inter-node communication using a non-local NIC, using NVLink and an intermediate GPU	Yes
NCCL_TOPO_FILE	Path to an XML file to load before detecting the topology	Yes
NCCL_P2P_DISABLE	The NCCL_P2P_DISABLE variable disables the peer to peer (P2P) transport, which uses CUDA direct access between GPUs, using NVLink or PCI.	Yes

Continued on next page

Table 6: NCCL environment variables and how they affects the communication patterns (Continued)

NCCL environment variable	Description	Support?
NCCL_P2P_LEVEL	Define the levelwhen to use the peer to peer (P2P) transport between GPUs.	Yes
NCCL_P2P_DIRECT_ DISABLE	Define and set to 1 to disable direct user buffer access across GPUs.	Yes
NCCL_SHM_DISABLE	NCCL will use the network (i.e. InfiniBand or IP sockets) to communicate between the CPU sockets when SHM is disabled.	Unnecessary
NCCL_SOCKET_IFNAME	Specifies which IP interfaces to use for communication.	Unnecessary
NCCL_SOCKET_FAMILY	The NCCL_SOCKET_FAMILY variable allows users to force NCCL to use only IPv4 or IPv6 interface.	Unnecessary
NCCL_SOCKET_ NTHREADS	Specifies the number of CPU helper threads used per network connection for socket transport	Unnecessary
NCCL_NSOCKS_ PERTHREAD	Specifies the number of sockets opened by each helper thread of the socket transport	Unnecessary
NCCL_DEBUG	Controls the debug information that is displayed from NCCL.	Unnecessary
NCCL_BUFFSIZE	Controls the size of the buffer used by NCCL when communicating data between pairs of GPUs.	Yes
NCCL_NTHREADS	Sets the number of CUDA threads per CUDA block	Unnecessary
NCCL_CHECKS_ DISABLE	Disable argument checks on each collective call.	Unnecessary
NCCL_CHECK_ POINTERS	Enables checking of the CUDA memory pointers on each collective call.	Unnecessary
NCCL_LAUNCH_MODE	Controls how NCCL launches CUDA kernels.	Unnecessary
NCCL_IB_DISABLE	Prevents the IB/RoCE transport from being used by NCCL	Yes
NCCL_IB_TIMEOUT	Controls the InfiniBand Verbs Timeout.	Yes
NCCL_IB_RETRY_CNT	Controls the InfiniBand retry count.	Yes
NCCL_IB_GID_INDEX	Defines the Global ID index used in RoCE mode	Yes
NCCL_IB_ADDR_ FAMILY	Defines the IP address family associated to the infiniband GID dynamically selected by NCCL when NCCL_IB_GID_INDEX is left unset.	Unnecessary

Continued on next page

Table 6: NCCL environment variables and how they affects the communication patterns (Continued)

NCCL environment variable	Description	Support?
NCCL_IB_ADDR_RANGE	The NCCL_IB_ADDR_RANGE variable defines the range of valid GIDs dynamically selected by NCCL when NCCL_IB_GID_INDEX is left unset.	Unnecessary
NCCL_IB_ROCE_ VERSION_NUM	Defines the RoCE version associated to the infiniband GID dynamically selected by NCCL when NCCL_IB_GID_INDEX is left unset.	Unnecessary
NCCL_IB_SL	Defines the InfiniBand Service Level.	Unnecessary
NCCL_IB_AR_ THRESHOLD	Threshold above which we send InfiniBand data in a separate message which can leverage adaptive routing.	No
NCCL_IB_CUDA_ SUPPORT	The NCCL_IB_CUDA_SUPPORT variable is used to force or disable the usage of GPU Direct RDMA	Unnecessary
NCCL_IB_SPLIT_ DATA_ON_QPS	This parameter controls how we use the queue pairs when we create more than one,	Yes
NCCL_IB_PCI_ RELAXED_ORDERING	Enable the use of Relaxed Ordering for the IB Verbs transport.	Unnecessary
NCCL_IB_ ADAPTIVE_ROUTING	Enable the use of Adaptive Routing capable data transfers for the IB Verbs transport.	No
NCCL_CUMEM_ENABLE	Use CUDA cuMem* functions to allocate memory in NCCL.	Unnecessary
NCCL_NET	Forces NCCL to use a specific network, for example to make sure NCCL uses an external plugin and doesn't automatically fall back on the internal IB or Socket implementation.	Yes
NCCL_NET_PLUGIN	Set it to a suffix string to choose among multiple NCCL net plugins	Unnecessary
NCCL_NET_ SHARED_BUFFERS	Allows the usage of shared buffers for inter-node point-to-point communication	Yes
NCCL_NET_ SHARED_COMMS	Reuse the same connections in the context of PXN.	Unnecessary
NCCL_PROTO	Defines which protocol NCCL will use.	Yes
NCCL_COLLNET_ ENABLE	Enable the use of the CollNet plugin.	No
NCCL_COLLNET_ NODE_THRESHOLD	A threshold for the number of nodes below which CollNet will not be enabled.	No
NCCL_TOPO_ DUMP_FILE	Path to a file to dump the XML topology to after detection.	Unnecessary

Continued on next page

Table 6: NCCL environment variables and how they affects the communication patterns (Continued)

NCCL environment variable	Description	Support?
NCCL_NVB_DISABLE	Disable intra-node communication through NVLink via an intermediate GPU.	Yes
NCCL_GRAPH_ REGISTER	Enable user buffer registration when NCCL calls are captured by CUDA Graphs.	Unnecessary
NCCL_SET_STACK_SIZE	Set CUDA kernel stack size to the maximum stack size amongst all NCCL kernels.	Unnecessary
NCCL_DMABUF_ ENABLE	Enable GPU Direct RDMA buffer registration using the Linux dma-buf subsystem.	Unnecessary
NCCL_P2P_NET_ CHUNKSIZE	The NCCL_P2P_NET_CHUNKSIZE controls the size of messages sent through the network for ncclSend/ncclRecv operations.	Yes

ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development

Borui Wan[1,*] Mingji Han[2,*] Yiyao Sheng[2] Yanghua Peng[2] Haibin Lin[2] Mofan Zhang[2]
Zhichao Lai[2] Menghan Yu[2] Junda Zhang[2] Zuquan Song[2] Xin Liu[2] Chuan Wu[1]

[1]*The University of Hong Kong* [2]*ByteDance*

Abstract

Checkpointing to preserve training states is crucial during the development of Large Foundation Models (LFMs), for training resumption upon various failures or changes in GPU resources and parallelism configurations. In addition, saved checkpoints are dispatched to evaluation tasks or transferred across different training stages (e.g., from pre-training to post-training). All these scenarios require resharding distributed checkpoints from one parallelism to another. In production environments, different LFMs are trained with various frameworks and storage backends, depending on model sizes and training scales. A high-performance checkpointing system is needed to enable efficient checkpoint management at scale throughout the lifecycle of LFM development.

We introduce ByteCheckpoint, an industrial-grade checkpointing system for large-scale LFM training. ByteCheckpoint features: a parallelism-agnostic checkpoint representation that enables efficient load-time checkpoint resharding; a generic checkpoint saving/loading workflow to accommodate multiple training frameworks and support different storage backends; full-stack optimizations to ensure high I/O efficiency and scalability; a suite of monitoring tools to streamline large-scale performance analysis and bottleneck detection. Compared to existing open-source checkpointing systems [52, 58], ByteCheckpoint significantly reduces runtime checkpoint stalls, achieving an average reduction of 54.20×. For saving and loading times, ByteCheckpoint achieves improvements of up to 9.96× and 8.80×, respectively.

1 Introduction

Large Foundation Models (LFMs) in language [5, 13, 33, 43], vision [40, 44], audio [59] and other modalities are revolutionizing today's AI landscape, spawning a variety of downstream applications such as conversational agents [49, 56], coding assistants [51], painting tools [53], and video [55] generators.

Unlike traditional deep learning model training, LFM training is significantly more complex. It has multiple training stages, including pre-training [9, 25] and post-training [38, 46, 67, 70]. Furthermore, evaluation tasks are integrated into these stages to effectively assess model quality. Beyond the complexity, the development of LFMs is notably resource-intensive and time-consuming, largely due to immense model sizes and massive training datasets (e.g., DeepSeek-V3 comprises 671 billion total parameters, and is pre-trained on 14.8 trillion tokens) [13, 33]. The scale of LFM training can even be up to running on 12,288 GPUs [25].

As a fundamental technique for preserving training states, checkpointing captures snapshots of these states and stores them in persistent storage to facilitate training resuming. Additionally, during LFM training, checkpoints are required for concurrent evaluation tasks that continually assess model quality. Another use-case of checkpoints is dispatching those snapshots from pre-training to downstream post-training tasks, such as Supervised Fine-Tuning (SFT) or reinforcement learning. The complex development pipeline, enormous scale, and prolonged duration of LFM training present significant challenges to designing a highly efficient checkpointing system.

First, efficient and unified checkpoint management is necessary throughout the life cycle of real-world LFM development. During training, checkpoint resharding is commonly required. This procedure transforms saved distributed checkpoints so they can be correctly loaded into a new parallelism configuration that differs from the one used for their creation. In pre-training, variations in parallelism occur when problematic machines are removed, the available GPU quota fluctuates [29, 72], or training configurations (e.g., context length) and system optimization techniques [25] (e.g., kernel fusion, computation and communication overlapping) are adjusted. Across different stages and tasks, parallelism varies according to the scales of resources and datasets in use. Highly efficient checkpoint resharding is needed to minimize extra overhead and maximize the end-to-end Effective Training Time Ratio (ETTR, calculated as the ratio between the productive training time and the wallclock time of a job) [27]. Moreover, on

*Equal contribution.

an industrial AI platform, LFM training jobs are initiated by various internal users or cloud customers who may choose different training frameworks (e.g., Megatron-LM [47], PyTorch FSDP [73], DDP [31], and others [63]) and select storage backends such as local disk, Hadoop Distributed File System (HDFS), or Network-Attached Storage (NAS) for storing checkpoints according to job characteristics and personal preferences. Crafting customized checkpointing modules and workflows to accommodate these ad-hoc implementations complicates system development and substantially increases maintenance costs. It is vital to provide generic workflows for different training frameworks and storage backends.

Second, Substantial I/O operations are involved in the checkpointing system for saving distributed checkpoints into persistent storage and loading them back for model training under various scenarios. To minimize I/O blocking time and promptly save training states to persistent storage, it is crucial to expedite the I/O workloads. In addition, ensuring the scalability of the checkpointing system is also essential since modern LFM training typically operates at large scales.

Existing checkpointing systems [14, 19, 35, 66] assume consistent parallelism, and fail to address the demands for checkpoint resharding. Although some efforts from the open-source community are devoted to developing checkpointing systems capable of resharding, they have several limitations. Some perform resharding in an inefficient offline manner [32], while others only support specific training frameworks and storage backends [52, 58]. Moreover, without customized optimizations and deployment experience, they suffer from sub-optimal I/O performance and lack the scalability to support large-scale LFM training in real-world production.

This paper presents the design, implementation, and deployment experience of ByteCheckpoint, a checkpointing system crafted for LFM development. ByteCheckpoint incorporates a unified architecture (Sec. 3.1) featuring an automatic *resharding-on-loading* mechanism for distributed checkpoints (load-time checkpoint resharding) and supports multiple training frameworks and storage backends. It integrates full-stack I/O performance and scalability optimization techniques.

▷ ByteCheckpoint's checkpoint representation is decoupled from the specific parallelism adopted during training (Sec. 3.2), enabling efficient load-time checkpoint resharding. For model and optimizer state representation, we separate each tensor shard's metadata from its numerical values and consolidate all the metadata into one global file. The metadata of a tensor shard includes its basic runtime, position, and storage information. For the representation of dataloader states, we divide them into *replicated* and *sharded* states, storing sharded states in individual files while the replicated ones are only saved by the training worker whose global rank is 0.

▷ ByteCheckpoint offers a generic workflow for LFM training tasks employing different training frameworks and various storage backends (Sec. 3.3). It tailors a planner for each framework to generate unified saving/loading plans. These plans

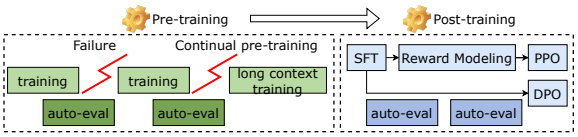


Figure 1: An overview of the training pipeline of LFM.

are then passed to a framework and storage backend agnostic engine to execute the I/O tasks. ByteCheckpoint's workflow utilizes this isolation in architecture, executing the same saving/loading steps for users with different training frameworks and storage backends.

▷ ByteCheckpoint implements multiple optimizations (Sec. 4) to enhance I/O performance, including balanced and zero-redundancy plan generation, fully asynchronous execution pipelines, and efficient irregular tensor processing. We share our engineering experience in optimizing storage systems to support massive I/O requests, optimizing collective communications to guarantee stability, and designing monitoring and visualization tools for performance analysis and bottleneck detection (Sec. 5). This full-stack approach scales ByteCheckpoint to support the training of a 405B LFM on 8,960 GPUs while still maintaining high efficiency.

ByteCheckpoint is deployed on our industrial AI platform with tens of thousands of GPUs for various LFM training tasks, including pre-training and post-training of language models, multi-modal understanding, and generation models (e.g., for video generation). ByteCheckpoint demonstrates significant advantages over existing open-source checkpointing systems, including PyTorch DCP [58] and Megatron Distributed Checkpoint [52] (MCP). Compared to the baselines, ByteCheckpoint achieves improvements ranging from 12.13× to 161.50× in terms of checkpoint stalls reduction, making the end-to-end checkpoint saving and loading procedures 6.05× and 3.88× faster on average, respectively.

2 Background and Motivation

2.1 LFM development

Production pipeline. As depicted in Fig. 1, the development of LFM comprises pre-training and post-training stages. During the initial pre-training phase, The LFM is iteratively trained on extensive data collected from multiple sources to absorb knowledge about the world. Subsequently, continual pre-training is employed to enhance the foundation model's capabilities. For instance, the pre-training of large language models (LLMs) typically involves long-context continual training to gradually increase the supported context length of LLMs. Post-training is employed to align the pre-trained model with human feedback or enhance the model's reasoning capabilities [18, 57]. Various task-specific labeled datasets (e.g., multilingual, code, math, reasoning, etc.) are involved in fine-tuning the LFMs, followed by reinforcement learning,

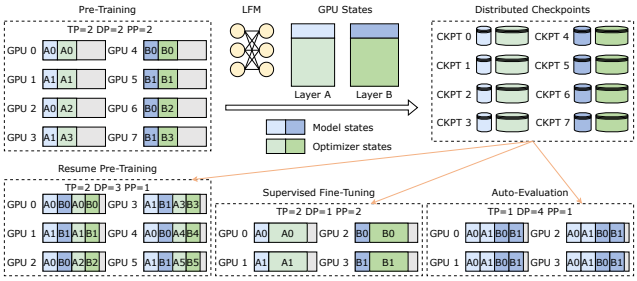


Figure 2: Checkpoint resharding scenarios in LFM training. We only show GPU states for clarity of the figure.

typically including the reward modeling and then performing Proximal Policy Optimization [45] (PPO), or directly conducting Direct Preference Optimization [41] (DPO). Due to the reduced size of these datasets, relatively fewer GPUs are involved in post-training. Automatic evaluation [14, 22] is periodically triggered to get intermediate model checkpoints and assess the quality with diverse criteria.

Checkpointing. Training states of LFM training jobs to be checkpointed include GPU and CPU states. GPU states are learnable parameters in the LFM model and optimizer information (e.g., the float32 precision replica of the model and its momentum and variance in Adam [26]). In state-of-the-art parallel training [28], these states are sharded and placed across multiple GPUs. CPU states include dataloader module, Random Number Generator (RNG) state, global training step number, and learning-rate scheduler, all stored in CPU memory. Our dataloader module incorporates a token buffer to cache input samples of varying lengths read from the data sources; when the number of accumulated tokens reaches the context window size [13, 25, 69, 71], the dataloader assembles all cached samples into a batch (micro-batch). Due to the volatile nature of GPU and CPU memory, these training states should be periodically saved into persistent storage to tolerate any faults and prepare for evaluation tasks.

Storage backends. In production environments, separate distributed file systems (such as Tectonic [39] for Llama 3.1 training [13]) are employed to store checkpoints [14, 20] for formal tasks. Given that various hardware failures and software bugs are inevitable during training [13, 22, 25], storing checkpoints at different global training steps is necessary to safeguard training. Distributed file systems (e.g., HDFS, NAS) provide adequate storage capacity to accommodate multiple checkpoints of large models.

2.2 Checkpoint Resharding Scenarios

During the life cycle of LFM development, checkpoint resharding is consistently required due to changes in parallelism under different scenarios (Fig. 2):

(1) Training Resumption. GPU quota allocated for an LFM training job can vary due to the removal of faulty machines [22, 25], adding new machines released from completed tasks, or running on tidal resources [12, 29, 68, 72]. It is often necessary to adjust training parallelism in response to the resource changes, to maximize resource utilization. Additionally, at the onset of large-scale pre-training, AI engineers often need to experiment with various model configurations and optimization techniques [25], since conclusions from small-scale profiling or simulations do not always translate to optimal performance in large-scale training. In this phase, adjusting parallelism configurations is common, resulting in frequent training resumption. Moreover, long-context training changes the context length, which also requires GPU quota or parallelism adjustment. As shown in the example in Fig. 2, 8 distributed checkpoint files are initially stored, then loaded into 6 training workers upon resuming.

(2) Cross-Stage Transition. When entering post-training , the number of GPUs involved typically decreases due to reduced training data in the latter. Checkpoints saved from pre-training are frequently resharded to align with the specific workloads of each post-training task. As depicted in Fig. 2, only 4 GPUs are involved for a fine-tuning task in the post-training stage, so the checkpoints are resharded accordingly.

(3) Evaluation. Evaluation tasks in both stages require loading model checkpoints and conducting inference on separate resources; their parallelism needs to be adjusted to align with assigned GPUs used for specific datasets. Fig. 2 shows a 4-GPU eval task reshards model checkpoints from pre-training.

We collected the number of times of checkpoint resharding demands on our AI platform (training text, image, and video generation models) over the past six months and identified 1,870 instances of checkpoint resharding during pre-training resumption, 13,080 due to cross-stage reconfiguration, and 19,844 for running evaluation tasks.

2.3 LFM Checkpointing Requirements

Load-time checkpoint resharding. As a critical step in the life cycle of LFM development, the efficiency of checkpoint resharding is vital to minimize extra overheads. In our AI platform, the previous common practice is to develop offline checkpoint resharding scripts and adapt the scripts whenever a new checkpoint resharding scenario arises. This approach is inefficient and labor-intensive (see Appendix A for further details). In addition to the development costs, running resharding scripts leads to a substantial waste of GPU time and resources. Table 1 presents the cost of executing resharding jobs in various scenarios. Before training can resume or new evaluation tasks can begin, independent jobs that execute the resharding scripts must be submitted in advance. These jobs download checkpoints from the storage systems, reshard distributed checkpoints to given parallelism configurations and upload new checkpoints back to the storage systems. The targeted training or evaluation jobs cannot be executed until the completion of resharding jobs, resulting in prolonged pend-

Table 1: Average completion time of executing offline resharding jobs under different scenarios.

Training Resumption	Cross-Stage Transition	Evaluation
1870.38s	650.34s	593.21s

Table 2: Top three training frameworks used on our platform

Framework	Pre-training	Post-training	Average #GPUs Per Job
Megatron-LM	13727	68621	301
FSDP	16842	†	25
DDP	25393	†	6

ing time. Moreover, since checkpoints created by resharding scripts are coupled with specific parallelism, they cannot be reused freely, which in turn increases the storage overhead.

Instead of executing scripts in independent jobs for checkpoint resharding, *load-time checkpoint resharding* [58] (also known as online resharding) is preferred for LFM training. It automatically identifies and retrieves necessary data from existing distributed checkpoints during the loading procedure. To achieve this, the storage representation of checkpoints should be designed to be independent of specific parallelism.

Multiple frameworks and storage backends. On our AI platform, a wide range of training frameworks are used, such as Megatron-LM [47], DDP [31], FSDP [73], veScale [25,63], etc. Table 2 lists the top three preferred training frameworks on our platform, based on six months of trace analysis. Users typically adopt Megatron-LM [47] for training large language foundation models. FSDP [73] is preferred for training tasks involving text-to-video or text-to-speech models, and DDP [31] is commonly used to train image encoder components of multimodal foundation models or for routine algorithm testing. In addition, users can choose from various storage backends for checkpoint persistence depending on the scenario, ranging from local disks for debugging to HDFS or NAS for formal training tasks. Each training framework comes with its own checkpoint module, file format, and ad-hoc implementation of the save/load logic. However, these modules lack critical features for production, such as load-time resharding, asynchronous checkpointing, and support for remote persistent storage. Integrating these features for each framework's checkpoint module and tailoring optimized implementations requires repeated engineering efforts. This leads to inconsistent checkpointing interfaces across training frameworks and storage backends, complicating the codebase. Moreover, the maintenance of diverse checkpoint file formats from different frameworks adds to the complexity of implementing checkpoint transfer logic across training stages and deploying models for evaluation and inference tasks. Therefore, it is vital to provide generic workflows for different training frameworks and storage backends.

Efficient and scalable I/O performance. Mainstream LFMs

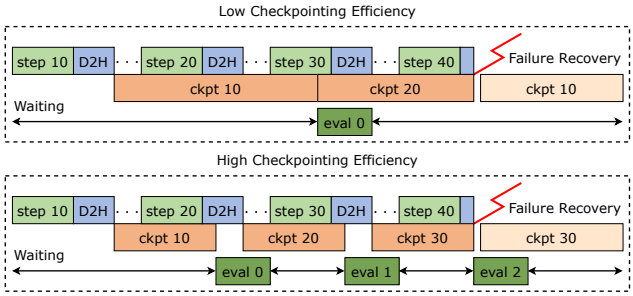


Figure 3: Checkpointing efficiency impacts failure recovery and evaluation tasks. D2H denotes the Device-to-Host copy.

feature extensive model sizes, reaching hundreds of billions of parameters [13]. Consequently, the size of training states also increases significantly, imposing substantial overhead for checkpoint saving and loading. By analyzing our previous LFM training jobs, we observe that the average end-to-end time required to save checkpoints of a GPT 175B model, trained on 4096 GPUs, to HDFS can be 200 seconds. This duration substantially exceeds the time required for a single training iteration. Even though this time-consuming procedure can be partially removed from the critical path of model training by adopting asynchronous checkpointing [25, 34, 35, 65], expediting end-to-end checkpointing remains crucial to minimize training progress loss [66] caused by inevitable frequent failures in large-scale training [25]. As depicted in Fig. 3, although checkpointing overlaps with training, its rapid completion allows more intermediate checkpoints to be stored before a failure occurs, enabling resumption from more recent states and improving ETTR. Besides, evaluation tasks are triggered during training, and intermediate checkpoints are periodically pulled for these tasks. Faster checkpointing ensures their timely execution, reducing blocking time due to preparing these checkpoints in remote persistent storage.

Scaling the checkpointing system while maintaining the high I/O performance is non-trivial. Sub-optimal and risky designs, which are hard to detect in small-scale settings, can lead to severe performance bottlenecks or even catastrophic job failures in large-scale training. For example, massive read/write requests for checkpoint files from the training cluster to the storage systems (e.g., HDFS) can overload the master node, causing delays in file metadata operations. Additionally, naive implementations of collective communications, such as integrity-checking barriers, introduce substantial initialization and synchronization overheads. These overheads can even result in communication timeouts, ultimately causing the entire training job to fail. Furthermore, as training scales up, efficiently analyzing system performance and detecting errors among training and I/O workers across multiple machines becomes increasingly challenging.

Existing checkpointing systems, such as CheckFreq [35], Check-N-Run [14], and Gemini [66], operate under the assumption of consistent parallelism and do not address the

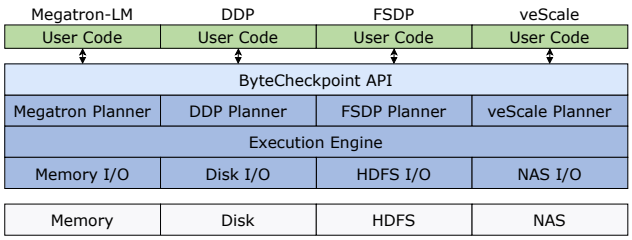


Figure 4: Architecture of ByteCheckpoint.

Loading and saving distributed checkpoints with ByteCheckpoint

```
# Import the ByteCheckpoint library
import bytecheckpoint
# Prepare checkpoint states
ckpt_states = {"model": model, "optimizer": optimizer, "dataloader": dataloader, "extra_states": extra_states}
# Load checkpoints of Megatron-LM from HDFS with overlapping mode
bytecheckpoint.load('hdfs://demo_0/checkpoints', ckpt_states, framework='megatron', overlap_loading=True)
# Train model for several steps.
# Save checkpoints of Megatron-LM to HDFS asynchronously
bytecheckpoint.save('hdfs://demo_0/checkpoints', ckpt_states, framework='megatron', async_checkpoint=True)
```

Figure 5: Examples of using ByteCheckpoint's APIs.

need for checkpoint resharding. DCP [58] and MCP [58] incorporate checkpoint resharding capabilities but are limited in terms of supported parallelism strategies and training frameworks, I/O performance, and scalability. A comprehensive discussion with related works is provided in Appendix F. In ByteCheckpoint, we design decoupled storage representation for efficient load-time checkpoint resharding, propose generic saving/loading workflows to support multiple frameworks and storage backends, integrate full-stack optimizations to enhance I/O performance and share our experience in scaling the checkpointing system to support real-world LFM training.

3 System Design

Based on the above observations, we design ByteCheckpoint according to the following key principles:
(1) Decoupling. The checkpoint representation remains independent of specific runtime parallelism. Interfaces of training frameworks and storage backends are separated from the core execution engine, ensuring robust extensibility.
(2) User-friendliness. The APIs should be concise, making it seamless for AI researchers and engineers to integrate them into their code and runtime environments.

3.1 Unified Architecture

The architecture of ByteCheckpoint is illustrated in Fig. 4. Each component is introduced in detail as follows:
API. ByteCheckpoint's APIs (`bytecheckpoint.save` and `bytecheckpoint.load`) provide a unified entrypoint for user code across various training frameworks. For instance, to save checkpoints, users first prepare the corresponding training states, checkpoint path, framework name, and performance-related options, then call `bytecheckpoint.save`. This high-level entrypoint abstracts underlying system complexities, such as sharding specification, save/reshard plan generation, and I/O operations.
Planner. The Planner serves as the interface for training frameworks. It receives arguments (training states, checkpoint path, etc.) from the API layer, creates ShardMeta (Sec. 3.2) for each tensor shard based on the worker's rank and framework-specific sharding specification such as *Megatron ShardedTensor* or *FSDP DTensor*, and determines the tensors and other states to save/load for each worker. Each worker leverages the Planner to initially create local plans and subsequently collaborates to create global plans. We implement a tailored planner for each training framework to extract information from these specifications and generate plans.
Execution Engine. The Engine, running on each training worker, executes the saving/loading plans generated by the Planner when the respective API is called. It analyzes the given checkpoint path to determine the appropriate storage backend, then interacts with the Storage I/O layer to execute the actual I/O tasks specified in the plans.
Storage I/O. Analogous to the design of the Planner layer, the Storage I/O layer encapsulates different storage backends and manages backend-specific read/write operations and optimizations. The interface between the Engine layer and the Storage I/O layer remains unified across different storage backends, facilitating the seamless integration of new storage backends. ByteCheckpoint supports multiple storage options, including in-memory checkpoint storage [66], local disk storage, and remote storage systems.

Fig. 5 illustrates a typical use case of ByteCheckpoint. Initially, users define a dictionary specifying the states to be saved or loaded, encompassing the model, optimizer, dataloader, and additional states. At the outset of training resumption, stage transition, or evaluation, `bytecheckpoint.load` is invoked to load saved checkpoints. Checkpoint resharding occurs automatically during loading when there are changes in parallelism. During training, `bytecheckpoint.save` is periodically invoked to save checkpoints.

3.2 Decoupled Checkpoint Representation

To achieve parallelism-agnostic checkpointing to enable automatic load-time resharding, we develop a formal specification of model and optimizer tensors in distributed training. Our design conforms to the Distributed Tensor and Checkpoint concepts in the *PyTorch Distributed* library [31].
Model and optimizer states. Each tensor is uniquely identified by a "fully qualified name" (FQN) and has a global shape, representing its original shape before sharding. Tensors can be either sharded or replicated across ranks (aka training workers). For sharded tensors, the specific shard held by a worker is determined by three factors: the parallelism (e.g., tensor parallelism) applied to the tensor, the sharding dimension, and

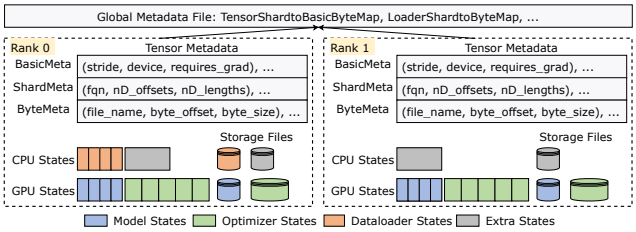


Figure 6: Checkpoint representation in ByteCheckpoint. The pipeline parallelism is applied in this example.

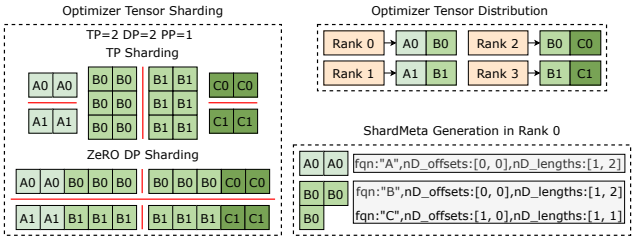


Figure 7: ShardMeta of irregular tensors in optimizer states.

the group rank within the corresponding parallelism group. Based on a tensor's FQN, global shape, and sharding specification, ByteCheckpoint creates the tensor shard metadata (ShardMeta) for each rank, which is utilized in checkpoint storage to represent the tensor shard, independent of parallelism. More precisely, ShardMeta for a tensor shard is an index tuple *(fqn, nD_offsets, nD_lengths)*, where *nD_offsets* and *nD_lengths* indicate the local shard's offsets and lengths along the original multi-dimensional axes in its global shape.

Dataloader and extra states. The dataloader states can be categorized into two types: *replicated* states and *sharded* states. Replicated states include the number of data reading workers, paths to source datasets, and sampling ratios, and are identical across all I/O workers (subprocesses) in different ranks. Sharded states are unique to each I/O worker, including the token buffer and data retrieval offsets for different data sources. In ByteCheckpoint, sharded states are saved in individual files, while replicated states are saved only by the I/O worker in rank 0. This reduces overall size of dataloader states to be saved and facilitates dataloader resharding when parallelism configurations change since the separation of data states into distinct files simplifies states merging and redistributing according to the new parallelism. For other extra states such as the RNG state, we pack and serialize them into one compact byte object before dumping them into storage.

Checkpoint representation. Fig. 6 illustrates the checkpoint representation in ByteCheckpoint. Distributed checkpoints comprise a *global metadata file* and multiple *storage files*. Each rank generates three distinct files: a model state file, an optimizer state file, and an extra state file. The Dataloader state file is generated only by training workers whose ranks for all parallelism degrees, except for DP degrees, are 0. For tensor shards in model and optimizer states, their *Metadata* consists of three parts: *BasicMeta*, which records essential information of individual tensor shards such as stride and device, critical for recovering the runtime state; *ShardMeta*, as previously introduced, recording relative position information of shards in the complete tensor; *ByteMeta*, which specifies the byte start offset and length of each tensor shard within the storage file. All tensor metadata are consolidated into the global metadata file, and a mapping, termed *TensorShardtoBasicByteMap*, is established between saved tensor shards and storage files based on the ShardMeta, BasicMeta, and ByteMeta of each tensor shard, ensuring accurate data retrieval. The global metadata file also includes a *LoaderShardtoByteMap*, which records the file index information of sharded states in each dataloader. All storage files and the global metadata file are stored in the specified storage backend designated by the checkpoint path.

Decomposing irregular tensors. We define irregular tensors as those which, when flattened before sharding, cannot be reshaped back to their original dimensions. These tensors typically arise when applying the Zero Redundancy Optimizer (ZeRO) [42], as implemented in Megatron-LM ZeRO2 and FSDP ZeRO3 [73]. In these implementations, optimizer states within a DP group are flattened, concatenated, and sharded. Consequently, the resulting 1D tensor slices often cannot be directly represented using n-dimensional shapes and offsets. Fig. 7 illustrates an example of irregular tensor sharding: Unlike tensors A and C, tensor B, with an original shape of (3, 2), is evenly split into two shards. Each shard, containing three elements, cannot be directly represented as a two-dimensional shape with corresponding offsets. One intuitive approach to addressing the challenge of irregular tensor shards is to merge all tensor shards into complete tensors before saving checkpoints, thereby simplifying the generation of ShardMeta. For example, to eliminate potential irregular tensors in DCP [58], FSDP performs synchronous all-gather communication operations, interleaved with D2H copy operations for each tensor shard, regardless of whether the shard is irregularly sharded. However, this approach incurs significant communication overhead and requires frequent synchronization between GPU and CPU, substantially impeding efficiency.

To mitigate the overhead associated with merging tensor shards, ByteCheckpoint employs a tensor decomposition strategy for managing irregular tensor shards. Specifically, ByteCheckpoint decomposes an irregular tensor into a series of regular ones, representing each with index tuples. For example, consider the irregular tensor shard of tensor B with three elements on rank 0 in Fig. 7. ByteCheckpoint decomposes this shard into two regular tensors that can be directly represented by $nD_offsets$ and *nD_lengths*. This approach allows ByteCheckpoint to use multiple *ShardMeta* entries to represent a single irregular tensor shard. Although such decomposition slightly increases the metadata size and adds steps to the loading procedure, as reconstructing the target tensor may require looking up multiple smaller segments of

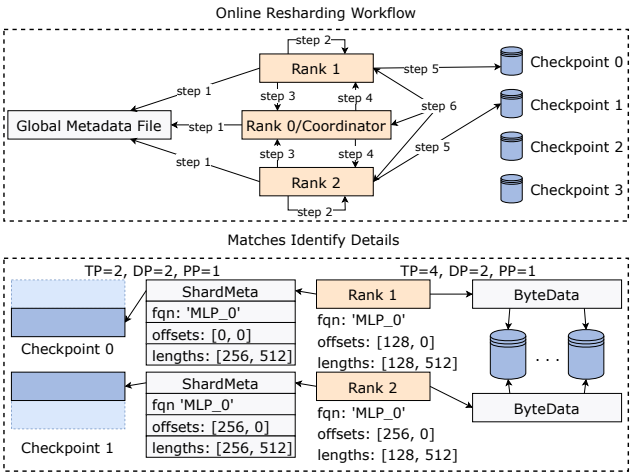


Figure 8: Load-time tensor resharding workflow. Checkpoint k denotes the checkpoint saved by a previous live worker k.

irregular shards, it avoids the costly communication of tensor shards without extra blocking time during saving.

3.3 Checkpoint Resharding Workflow

ByteCheckpoint implements a generic workflow for saving and loading, automatically resharding checkpoints during the loading process. Leveraging the independence between the Engine Layer, Planner Layer, and Storage I/O Layer, we can execute consistent saving and loading steps across different training frameworks and storage backends. We take tensor resharding as an example in Fig. 8, emphasizing how our checkpoint representation enables flexible load-time resharding. The process for checkpoint saving and loading without resharding follows a similar procedure.

Step 1. To initiate load-time checkpoint resharding, each rank invokes the `bytecheckpoint.load()` API, specifying the checkpoint path along with the model/optimizer to be restored. All ranks then load the global metadata file from the path.

Step 2. For each tensor shard in the given model/optimizer, each rank queries the TensorShardToBasicByteMap within the global metadata file, identifying matching segments between the saved tensor shards and the sharding specification of new shards. After identifying these matches, the planner constructs a local load plan, which includes BasicMeta and ByteMeta tailored to the specific shard. This identification mechanism is illustrated at the bottom of Fig. 8.

Step 3. The coordinator planner, typically residing in rank 0, initiates a gather operation to aggregate loading plans from all ranks. It then optimizes each local plan by applying the redundant elimination optimization (Sec. 4.1) to distribute tensor shard loading workloads for reduced completion time.

Step 4. The coordinator initiates a scatter operation to distribute the finalized loading plan. Each rank then receives its final loading plan from the coordinator.

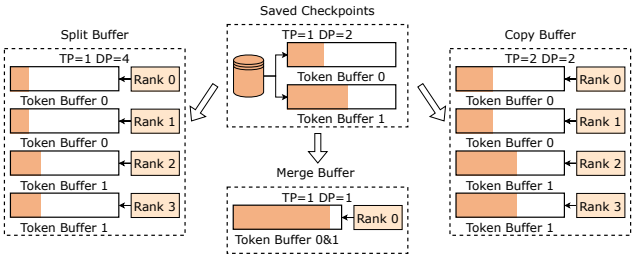


Figure 9: Examples of dataloader resharding. Sharded states such as cumulative token buffers are resharded according to the changes of parallelism configurations. We do not depict data retrieval offsets for clarity of the figure.

Step 5. The execution engine on each rank selects the storage backend wrapper in the Storage I/O layer based on the checkpoint path and then executes the loading pipeline (Sec. 4.2).

Step 6. Upon completion of the loading pipeline, each rank utilizes the optimized asynchronous collective barrier primitive ensuring the atomicity of the distributed loading. Further details can be found in Appendix B.

Dataloader resharding. Similar to tensor resharding in Fig 8, ByteCheckpoint reshards the dataloader by querying the LoaderShardtoByteMap. The illustration of dataloader resharding is depicted in Fig. 9. When parallelism configuration changes, for the dataloader checkpoints, common items in the dataloader checkpoints can be loaded directly, whereas unique items such as accumulative token buffers and data retrieval offsets require resharding. Specifically, when the DP degree size remains constant while other parallel degrees are altered (TP degree changes in the example of Fig. 9), the token buffers should be copied to the destination workers for bitwise-correct resuming. when there is a change in the DP degree size, the token buffers must be either split or merged accordingly to ensure that the resumed dataloaders do not discard cached data and do not retrain data that has already been sampled and fed. Thanks to the split storage strategy designed for the dataloader module, ByteCheckpoint can precisely identify the unique items that require resharding and process them efficiently.

4 Performance Optimization

We now elaborate on ByteCheckpoint's performance optimization techniques, focusing on minimizing the overhead associated with checkpoint saving and loading.

4.1 Optimized Plan Generation

Balancing saving workload. In training scenarios employing data parallelism (DP), model states are replicated across all DP groups, resulting in duplicated model states. Existing checkpointing systems [52, 58] address this issue by designat-

ing the first DP group to save all model states. However, this approach leads to workload imbalance, potentially causing training workers in the first DP group to become stragglers. To address this challenge, we implement a workload-balancing deduplication mechanism during the planning procedure, utilizing a Worst-Fit algorithm. Specifically, the coordinator planner distributes the saving workload based on the size of each tensor shard, assigning the current tensor shard to the rank with the smallest cumulative tensor shard size. This approach ensures a more equitable distribution of the saving workload across ranks. This balances workloads across all workers, improving saving efficiency.

Eliminating redundant loading. When loading checkpoints into parallelism configurations that include data parallelism, ByteCheckpoint optimizes the process by eliminating repetitive tensor reading across DP groups, effectively combining storage file reading with tensor transferring. A crucial observation is that idle inter-GPU bandwidth can be utilized to concurrently transfer loaded tensors to peer GPUs during the file reading process. As illustrated in Fig. 10, tensor reading workloads across DP groups are evenly distributed among training workers during the planning phase, thereby avoiding duplication of reads. Subsequently, I/O threads are launched to read the allocated tensor shards. Concurrently, in the main thread, shards read into CPU memory are first copied into GPU memory and then transferred to other workers that require them, utilizing all-to-all collective communication.

Plan and metadata cache. In large-scale training, the execution of the planning procedure can introduce significant communication overhead, particularly when checkpoint saving occurs frequently. For example, planning the saving procedure for a 405B transformer model distributed across 8960 GPUs requires 62 seconds. However, we have observed that both the save plans and the global metadata file, although coupled with specific parallelism, remain constant throughout a single training session. This allows for caching strategies to reduce planning times and associated overheads in subsequent checkpoint operations. We introduce plan metadata caching, transforming the planning into a one-time cost. Once established for the first time, the save plans and global metadata file are cached for future reuse, eliminating repetitive planning.

4.2 Fully Asynchronous Engine Pipeline

ByteCheckpoint Engine optimizes operation execution during checkpoint saving and loading (resharding) through pipelining. Using loading as an example (Fig. 10), we pipeline file reading, deserialization, Host-to-Device (H2D) copy, and inter-GPU communication for each tensor shard, thereby achieving higher efficiency. The *read* operation downloads files containing desired tensors from the storage system and places them into shared memory (e.g., the `dev/shm` directory). The *deserialize* operation deserializes tensors from shared memory. ByteCheckpoint employs multiple threads

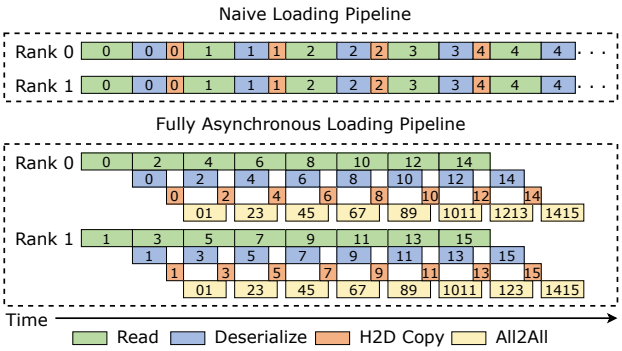


Figure 10: Comparison between ByteCheckpoint's loading pipeline and the naive implementation.

for parallel file downloading and deserialization. The *H2D copy* operation transfers tensors from CPU memory to GPUs, while *All2All* facilitates tensor transferring within the DP group. For saving, we implement a symmetrical, fully asynchronous pipeline comprising D2H copy, serialization, and file uploading operations. To mitigate the performance impact of D2H copy on training, we employ a pinned CPU memory pool combined with a Ping-Pong buffering mechanism to accelerate this operation. We run multiple parallel processes to serialize tensors and dump files into shared memory. Uploading threads actively monitor and initiate file uploading upon the completion of the dumping phase.

4.3 High Performance Read/Write

We optimize the read and write throughput of checkpointing by tailoring our implementation to optimal I/O usage of commonly used storage backends. For example, while HDFS is not primarily designed for random data access, it does offer some random read capabilities via its SDK, allowing applications to access specific file offsets and retrieve data from those positions. We leverage this feature and enable multi-threaded reading of a single file, significantly accelerating the download speed of checkpoint files from HDFS. In our production platform, single-file read speed has improved from 400 MB/s to 2-3 GB/s, all tested on H800 servers equipped with over 100 CPU cores, TB-level memory, and a 200Gbps NIC.

For file writing, HDFS's append-only write makes it impractical to split a single file into multiple parts based on offsets for multi-threaded writing. To overcome this limitation, we split the target file into several sub-files of fixed size and concurrently write them into HDFS using multi-threading. After the upload is complete, we perform metadata-level concatenation to seamlessly merge the sub-files back into a single entity, ensuring the integrity of the stored data blocks. Without network congestion, single-file upload speed can reach 3 GB/s, far exceeding the average read/write throughput of a single HDFS client (under 100 MB/s [48]).

4.4 Prefetching Dataloader States

Each dataloader employs multiple subprocesses, referred to as read workers, to handle data loading and preprocessing. When checkpointing is initiated, the main read worker signals all other read workers to prepare their states. The training process is paused until all states are collected to ensure accuracy, as any update in the main process would result in the state change of workers. The duration of this blocking period depends on both the number of workers and the size of the accumulated token buffers. For instance, when using a dataloader configured with 4 workers and a total state size of approximately 1GB, the state collection process typically takes around 8 seconds.

To alleviate this overhead, we adopt prefetching. Based on the pre-set checkpoint frequency, each read worker prepares its state during the training step just before checkpointing and puts the state into its state queue. At the checkpointing step, the main read worker gathers those prepared states immediately through queue polling. This optimization allows ByteCheckpoint to achieve near-zero dataloader state gathering delays.

5 Checkpointing at Scale

This section explores in depth the methods employed to optimize checkpointing for large-scale LFM training.

5.1 Storage Support

High throughput, scalable storage system. HDFS is the primary storage backend for ByteCheckpoint, we have implemented several optimizations to enhance throughput and scalability for large-scale checkpointing. We have rewritten all HDFS components, including NameNode, DataNode, and SDK [15], in C++, effectively doubling performance compared to the original Java-based implementation. Additionally, we have refined its architecture by introducing a new component named NNProxy [1]. NNProxy functions as a stateless RPC proxy for HDFS NameNode, facilitating large-scale federated deployments of NameNodes while maintaining minimal querying latency. The NNProxy design addresses the QPS bottleneck of HDFS NameNode metadata requests, a challenge intensified by the high volume of distributed checkpoints. NNProxy also provides additional features such as authentication, rate limiting, and metadata query caching. These capabilities enable more granular management of checkpoints in production environments. Our optimized HDFS achieves a substantial capacity of hundreds of PBs, 10 TB/s read/write bandwidth, and approximately 100,000 metadata QPS.

Checkpoint cool-down strategy. We have implemented a two-tier hot-cold storage architecture using a combination of SSD and HDD storage servers. Our key observation is that newly stored checkpoint files are typically downloaded by evaluation tasks shortly after their creation. However, where no training anomalies such as loss spikes occur, the access frequency of these files decreases significantly after being downloaded by evaluation tasks. Nonetheless, all checkpoint files must be preserved for traceability. To efficiently manage storage space while ensuring data availability, we have developed a data cool-down mechanism that migrates data from SSD to HDD storage, thus ensuring that high-performance hot storage consistently has sufficient space for operations. Specifically, we cool down all files that exceed the retention threshold based on their last modification time. We then remap the original file paths to the new HDD storage locations through pure metadata operations. This strategy applies to HDFS directories and preserves the original access paths of the cooled-down files, providing a seamless user experience.

5.2 Collective Communication

Collective communications (e.g., scatter, gather, barrier) are crucial in ByteCheckpoint's workflow (Sec.3.3), particularly for planning and checkpoint integrity guarantee. We initially used NCCL [54] as the communication backend to execute gather and scatter operations at the coordinator. However, when scaling a pre-training task to 8960 GPUs, we observed that NCCL requires a long time to lazily build communication channels and allocate GPU memory during the planning stage of ByteCheckpoint's saving workflow. In some cases, it becomes unresponsive or causes CUDA out-of-memory (OOM) errors, as scatter or gather operations necessitate establishing peer-to-peer communication with each GPU. These GPU OOM issues and long initialization times are not apparent in small-scale trials but become significant at larger scales.

To enhance communication stability during planning, we have re-implemented the procedure using the gRPC framework [17], which eliminates GPU memory usage during planning. However, when scaling training to tens of thousands of GPUs, centralized gathering and scattering operations continued to impose a significant burden on the coordinator, leading to communication failures. We further improved its stability by implementing a tree-based hierarchical communication topology. Training workers on a single machine are organized into first-level subtrees, with the worker of local rank 0 designated as the root. For inter-machine communication, we iteratively group multiple machines, designating the worker with the lowest global rank in each group as the root. This procedure continues until all workers are integrated into a hierarchy converging at the global root (i.e., the coordinator). In large-scale 3D parallel training scenarios, a TP-DP-PP communication tree naturally forms, removing extra connections.

5.3 Monitoring and Analysis

During training, ByteCheckpoint continuously collects critical performance measurements and visualize them for real-

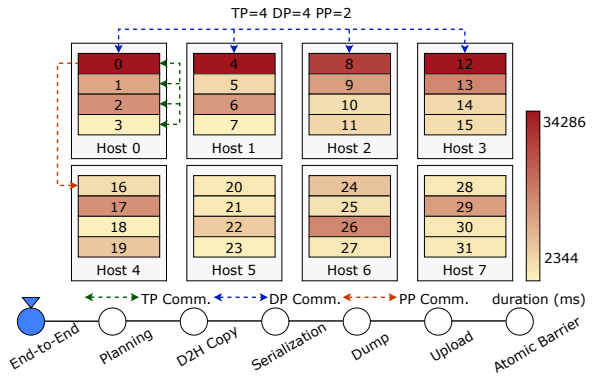


Figure 11: End-to-end checkpoint saving time heat map from a 3D parallel training task with Megatron-LM on 32 GPUs. The color denotes the time of a selected phase on a rank.

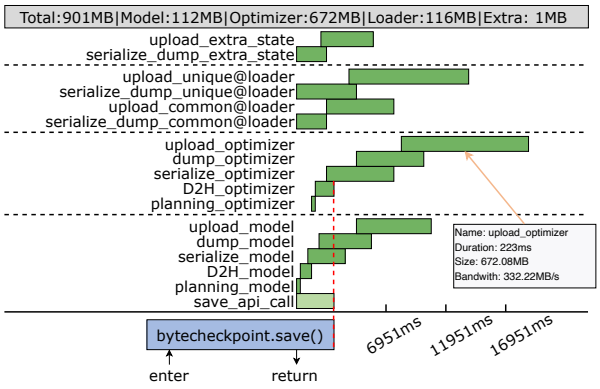


Figure 12: Time breakdown of checkpoint saving on rank 0.

time performance monitoring and analysis. This approach enables quick checkpointing issues detection, such as low read/write throughput, stragglers, and upload failures and retries.

Data collection. We have designed a user-friendly metrics system based on Python's context manager and decorator syntax to monitor critical procedures flexibly. It automatically captures the duration and I/O size of each operation, along with relevant metadata such as each worker's rank, the file path, and the current step. All collected metrics are transferred to a remote database via a background message queue.

Visualization. ByteCheckpoint provides users with a comprehensive topological performance overview of all ranks associated with different checkpoint phases, as well as detailed duration breakdowns for any specific rank. Fig. 11 presents an exemplary heat-map visualization of checkpoint saving times within a 3D parallel training topology. The visualization highlights that ranks 0, 4, 8, and 12 experience the longest saving times as their checkpoints including dataloader states. Additional metrics visualizations are also available for detailed analysis, such as blocking time and fine-grained phases like planning and D2H copy. These visualizations enable users to easily pinpoint straggler nodes or phases for both system development and in production environments. For instance, if certain nodes experience network issues, increased upload or download times during the HDFS transfer phase would be readily apparent. Furthermore, detailed timeline breakdowns of checkpointing procedures at each rank are accessible via the heat map topology overview, enabling users to thoroughly evaluate all optimizations in the system. Fig. 12 illustrates the execution details of each checkpoint-saving phase for a specific rank.

Storage-side monitoring. On the storage client side, we monitor the latency and I/O size of each atomic read/write operation at the I/O chunk level. All collected metrics are periodically transmitted via a message queue to a ClickHouse service, where the data are aggregated and analyzed. Unexpectedly high latency or low bandwidth triggers alerts to engineers for further investigation. On the storage cluster side, our primary focus is on overall performance metrics, including metadata request QPS, cluster-level read/write throughput, and storage capacity utilization. This comprehensive monitoring system plays a crucial role in identifying issues such as slow checkpoint reads and writes, and even complete system unavailability due to storage capacity exhaustion, facilitating the implementation of preventive measures for enhancing system reliability and efficiency.

6 Evaluation

In this section, we detail our deployment and operational experiences with ByteCheckpoint. ByteCheckpoint is built upon DCP [58] (commit hash: 80c07df), a state-of-the-art open-source checkpointing system that supports checkpoint resharding for FSDP [73]. The implementation of ByteCheckpoint consists of about 20,000 lines of Python code.

6.1 Checkpointing Efficiency

We demonstrate checkpointing performance of ByteCheckpoint relative to baselines [52, 58] with different workloads.

Workloads. We adopt two different transformer-based [62] structures, i.e. DiT [40] and GPT-3 [9] to implement two models referred to as *vDiT* and *tGPT*, respectively. The vDiT model is fine-tuned with the FSDP framework for video generation on a cluster of NVIDIA A100 80GB GPUs, while the tGPT model is trained for text generation with Megatron-LM [47] on NVIDIA H800 80GB GPUs. Detailed configurations for model and parallelism are provided in Table 3, where the source number of GPUs and parallelism pertain to the saving and loading evaluations, while the target number of GPUs and parallelism correspond to the configurations used for load-time resharding evaluations. All machines in the training clusters are interconnected via InfiniBand, with HDFS selected as the persistent storage solution for all experiments. We integrate the test models into our LFM training

Table 3: Model and parallelism configurations.

Model	Hidden Size	#Heads	#Layers	#Parameters	Source #GPUs	Soure Parallelism	Target #GPUs	Target Parallelism
vDiT	1664	16	48	4B	32	ZeRO-2	64	ZeRO-2
					128	ZeRO-2	64	ZeRO-2
tGPT	8192	64	80	70B	2400	TP=4, DP=75, PP=8	4800	TP=4, DP=150, PP=8
					4800	TP=4, DP=150, PP=8	2400	TP=4, DP=75, PP=8

Table 4: I/O performance comparison among ByteCheckpoint, DCP and MCP.

Model and Framework	Source #GPUs	Method	T_{Block} (s)	T_{Save} (s)	T_{Load} (s)	$T_{Reshard}$ (s)	ETTR (%)
vDiT 4B	32	DCP	16.25	86.82	50.12	74.89	38.60
		ByteCheckpoint (GPU states)	0.54 (**30.09**×)	27.47 (**3.16**×)	11.69 (**4.29**×)	16.01 (**4.68**×)	46.22 (**1.20**×)
FSDP	128	DCP	61.37	236.34	105.74	91.01	41.62
		ByteCheckpoint (GPU states)	0.38 (**161.50**×)	23.74 (**9.96**×)	12.01 (**8.80**×)	13.64 (**6.67**×)	48.92 (**1.18**×)
tGPT 70B	2400	MCP	4.73	28.97	69.87	126.30	34.61
		ByteCheckpoint (GPU states)	0.39 (**12.13**×)	13.11 (**2.21**×)	49.48 (**1.41**×)	62.10 (**2.03**×)	40.18 (**1.16**×)
		ByteCheckpoint (full states)	0.50	20.55	72.35	401.21	28.80
Megatron-LM	4800	MCP	4.70	76.21	123.80	64.62	31.28
		ByteCheckpoint (GPU states)	0.36 (**13.06**×)	8.59 (**8.87**×)	64.39 (**2.00**×)	54.31 (**1.19**×)	40.29 (**1.29**×)
		ByteCheckpoint (full states)	0.43	25.04	83.56	115.95	34.70

trials, conducting training over 500 steps and saving checkpoints every 100 steps. We then resume training under two scenarios: with and without changes in GPU numbers and parallelism configurations, to assess the performance of load-time resharding and standard checkpoint loading. Saving performance is evaluated based on average additional overheads, specifically checkpoint stalls (training blocking time) and end-to-end checkpoint saving time (from the save API call to the completion of checkpoint integrity checking). Additionally, we measure the end-to-end checkpoint loading time (the blocking time of the load API call) for both load-time resharding and standard loading scenarios. We use the average ETTR as the metric to evaluate the end-to-end system performance. Following GEMINI [66], we assume a consistent occurrence of failures within each checkpointing interval (100 steps), calculating the average wasted time to derive the achieved ETTR. Calculation details can be found in Appendix C.

Baselines. In experiments with FSDP, we use DCP (commit hash: c7338f4) as the baseline, while for Megatron-LM, MCP [52] (commit hash: 3fb5c51) serves as the baseline. We ensure that all hyper-parameters, such as batch size and context length, remain consistent across systems for a fair comparison. Both baselines support asynchronous checkpointing and load-time checkpoint resharding for the model and optimizer, but not for dataloader states. Consequently, we compare I/O performance with these baselines only for GPU states and additionally evaluate the performance of ByteCheckpoint for the full training states. Since neither DCP nor MCP supports HDFS, we implement the logic to enable HDFS access for them, using the same configurations (e.g., thread count, file chunk size, etc.) as those employed in ByteCheckpoint.

Saving performance. As shown in Table 4, ByteCheckpoint significantly reduces checkpoint stalls during runtime, with reductions ranging from 13.06× to 161.50×. This improvement reduces the average checkpoint stall time from minutes or seconds to sub-second durations, outperforming both DCP and MCP. In terms of end-to-end saving time, ByteCheckpoint delivers an average speedup of 6.05×, primarily driven by our fine-grained performance optimizations. Notably, the acceleration provided by ByteCheckpoint scales with the size of the training workload, increasing from 2.21× for 2400 GPUs to 8.87× for 4800 GPUs. This can be attributed to the workload balancing mechanism, which becomes more effective with larger DP sizes. Additionally, we observe a particularly significant reduction in blocking time for FSDP workloads. This results from the high communication and synchronization overheads caused by FSDP's irregular tensor shard processing when using DCP for distributed checkpoint storage. These overheads grow as the training scale increases, making DCP less suitable for large-scale training. In contrast, ByteCheckpoint's decomposition representation strategy incurs zero communication overhead during metadata generation, leading to improved training efficiency. We also provide a detailed overhead breakdown in Appendix D for further investigation. For training with Megatron-LM, we report I/O performance for the handling of full training states.

Loading and resharding performance. In experiments involving the loading of distributed checkpoints into unchanged parallelism configurations, ByteCheckpoint achieves performance improvements ranging from 1.41× to 8.80× compared to baseline methods. Our approach leverages an asynchronous loading pipeline, which overlaps tensor reading and transferring, significantly reducing blocking time before training begins. When it comes to the resharding efficiency of model and optimizer states, ByteCheckpoint delivers an average acceleration of 3.64×. By eliminating the need for running costly

Table 5: Saving optimization microbenchmark.

Workload	Parallel Config.	Optimization	Saving Time (s)
tGPT 13B	TP=2, DP=8, PP=2	No Optim.	50.26
		Async.	34.68 (**1.45**×)
		Async. + WB.	20.28 (**2.48**×)
		Async. + WB. + Cache.	19.97 (**2.52**×)
tGPT 30B	TP=2, DP=8, PP=4	No Optim.	46.34
		Async.	25.56 (**1.81**×)
		Async. + W.B.	18.83 (**2.46**×)
		Async. + W.B. + Cache.	18.56 (**2.50**×)

Table 6: Loading optimization microbenchmark.

Model	Parallel Config.	Optimization	Loading Time (s)
tGPT 13B	TP=2, DP=8, PP=2	No Optim.	63.48
		Async.	48.43 (**1.31**×)
		Async. + Overlap.	41.38 (**1.53**×)
tGPT 30B	TP=2, DP=8, PP=4	No Optim.	77.02
		Async.	74.54 (**1.03**×)
		Async. + Overlap.	48.73 (**1.58**×)

Table 7: Resharding optimization microbenchmark.

Model	Parallel Config.	Optimization	Processing Time (s)
tGPT 13B	ZeRO-2 32 GPUs	All-gather + D2H.	4.12
		Decompose.	0.21 (**19.80**×)
tGPT 30B	ZeRO-2 64 GPUs	All-gather + D2H.	5.84
		Decompose.	0.19 (**30.50**×)

offline resharding jobs, ByteCheckpoint enables flexible and efficient checkpoint transformations across various scenarios. However, we observe that including CPU states, primarily dataloader states, increases the end-to-end load-time resharding time. This is largely due to the time-consuming processing of unique components within the dataloader states, such as the token buffer, which can grow as large as 20GB in text-to-video LFM training. Additionally, since only a subset of workers hold these dataloader states, they often become stragglers during the checkpoint saving and loading (resharding) procedures. We leave the design of a more efficient dataloader to our future works.

End-to-end performance. ByteCheckpoint outperforms both DCP and MCP in terms of the average end-to-end ETTR, achieving improvements ranging from 1.16× to 1.29×. Leveraging full-stack optimizations, ByteCheckpoint improves the execution efficiency of each checkpoint-related operation, thereby effectively minimizing training downtime.

6.2 Microbenchmarks

We conduct several micro-benchmarks (Table 5, Table 6 and Table 7) to highlight the performance improvements. To evaluate different model sizes, We modify the tGPT 70B model to create tGPT 13B and tGPT 30B. For saving and loading microbenchmarks, we used the Megatron-LM training framework, while for resharding, we employed FSDP.

In Table 5, we compare the end-to-end saving times with and without ByteCheckpoint features, as described in Sec. 4. The results demonstrate that asynchronous saving pipeline achieves significant reduction in saving time compared to the unoptimized settings, yielding speedups of 1.45× and 1.81× for tGPT 13B and tGPT 30B, respectively. The workload balancing mechanism and plan cache of ByteCheckpoint can further boost these improvements. When applying all I/O optimization techniques to both model and optimizer states, the average saving time for these models is reduced from 48.3 seconds to 19.27 seconds. In Table 6, we present the performance gains achieved by the asynchronous loading pipeline and read-communication overlap techniques. For tGPT 13B and tGPT 30B, the baseline loading times were 1.53× and 1.58× longer than those achieved with ByteCheckpoint.

For load-time resharding, we compared the irregular tensor processing time between FSDP and ByteCheckpoint's irregular tensor decomposition strategy. As shown in Table 7, ByteCheckpoint incurs an average blocking time of just 0.20s, which is 25.15× shorter than that of FSDP. Moreover, our approach consistently achieves microsecond-level blocking times, regardless of the training scale. (see results in Sec. 6.1 and Sec. 6.4).

6.3 Correctness Verification

In Fig. 13, we depict normalized training loss curves of tGPT 13B before and after resharding with ByteCheckpoint in various situations. For PP and TP resharding, the normalized loss curve after resharding can smoothly match that in the previous phase and continue to display a consistent decline trend. Unit testing confirms that ByteCheckpoint can correctly transform distributed checkpoints across varying parallelism. Results of DP and hybrid resharding are in Appendix E.

We further verify the bit-wise alignment ability of ByteCheckpoint when training resumes without changes in parallelism configurations. We show the normalized loss curve from real production in Fig. 14, where a 175B language foundation model is trained. As highlighted, the normalized loss before and after resuming are exactly the same.

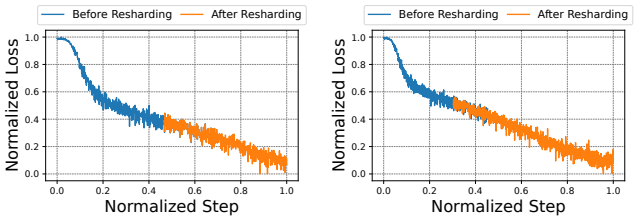
6.4 Checkpointing in Real LFM Production

We share our experience deploying ByteCheckpoint in real-world LFM production environments, highlighting the issues we encountered and resolved.

Scalabilty. As shown in Table 8, ByteCheckpoint was used to train two transformer-based LFMs for image and text generation on H800 GPUs, respectively. ByteCheckpoint consistently maintained average checkpoint stalls under 600ms, even at the largest scale with 8,960 GPUs. Additionally, end-to-end checkpointings were completed efficiently, typically within a few seconds.

Table 8: I/O performance of ByteCheckpoint in large-scale LFM training.

Model and Framework	#GPUs	Parallelism	T_{Block} (s)	T_{Save} (s)	T_{Load} (s)
Vision Transformer 7B FSDP	1488	ZeRO-2	0.34	20.13	265.73
Text Transformer 405B Megatron-LM	8960	TP=8, DP=70, PP=16	0.59	51.06	129.49



(a) PP resharding from TP=1, DP=4, PP=4 to TP=1, DP=4, PP=8.

(b) TP resharding from TP=1, DP=4, PP=4 to TP=2, DP=4, PP=4.

Figure 13: Resharding correctness verification.

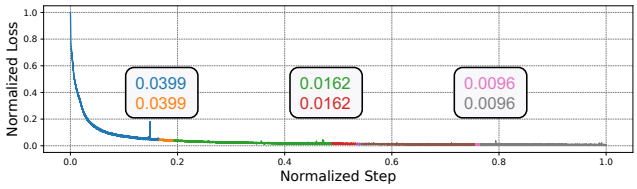


Figure 14: Training resuming with ByteCheckpoint in real production run on 2080 H800 GPUs for several days. Color changes indicate training resuming.

Dataloader stragglers. By analyzing the performance visualization results from our monitoring tools (Sec. 5.3), we identified prolonged uploading time for dataloader checkpoint files. To support dataloader state resharding, ByteCheckpoint divides the states of a dataloader into multiple parts (e.g., 6 parts for the tGPT experiments in Sec. 6.1) and stores them as individual files. These files can be independently loaded during resharding. However, uploading these small files sequentially leads to low bandwidth utilization, causing performance degradation. In cases like the 7B vision transformer, uploading time of dataloader states can account for 73.16% of total saving time. To fix this issue, we implemented a process pool for concurrent uploads, significantly reducing saving time.

HDFS metadata bottleneck. Through monitoring, we identified several bottlenecks related to HDFS. Despite applying the parallel optimization techniques for single-file uploads described in Sec. 4.3, the actual HDFS upload performance fell short of our expectations. A detailed profiling revealed that the metadata concatenation operation in the HDFS NameNode was performed serially, making it a significant performance bottleneck when large volumes of checkpoint files were generated. We mitigated this issue by enabling parallel execution of the concatenation operation. Another issue that emerged was related to our initial use of the HDFS SDK. For the convenience of the users, the SDK encapsulates a lot of safeguard logic, leading to unnecessary overheads introduced by checking, creating parent directories, and verifying target files. To address this, we optimized ByteCheckpoint by ensuring directory existence and file uniqueness prior to invoking HDFS APIs, reducing redundant metadata operations and improving performance significantly. By addressing these issues, we reduced the maximum metadata overhead of HDFS write operations for a single checkpoint file from 3s to 150ms.

7 Conclusion

This paper presents the design, implementation, and deployment of ByteCheckpoint, a production-grade checkpointing system built for LFM development. ByteCheckpoint advocates a unified architecture for checkpoint representation and workflow, enables efficient load-time resharding and supports multiple training frameworks and storage backends. It enables full-stack I/O performance and scalability optimizations and incorporates efficient performance monitoring and analysis tools for LFM development at scale. Compared to state-of-the-art baselines [52, 58], ByteCheckpoint reduces checkpoint stalls by up to 161.50× and shortens end-to-end checkpointing completion time by up to 9.96×. Additionally, the loading (resharding) procedure is optimized to be 3.88× faster on average. We believe our system not only provides valuable practical experience to those building checkpointing systems for real-world LFM development but also offers deep insights for future research in the community.

8 Acknowledgments

We would like to thank our shepherd Ayush Goel and the anonymous reviewers for their constructive feedback. We thank Juncai Liu and Xiang Li for their efforts in facilitating our large-scale experiments. We thank Shibiao Nong, Shuaishuai Cao, Wei Jia, Jingzhe Tang, Jun Wang, Xi Wang, Sa Wang, and other contributors for refining our system and extending its functionalities. We thank Liyang Zhao and the HDFS team of ByteDance for their support. This work was supported in part by a ByteDance Research Collaboration Project, and grants from Hong Kong RGC under the contracts HKU 17204423 and C7004-22G (CRF).

References

[1] HDFS Federation Solution: NameNodeProxy. https://github.com/bytedance/nnproxy, 2016.

[2] Safetensors. https://huggingface.co/docs/safetensors/index, 2024.

[3] TensorStore. https://google.github.io/tensorstore/, 2024.

[4] Zarr. https://zarr.dev/, 2024.

[5] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

[6] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, et al. Fire-Flyer AI-HPC: A cost-effective software-hardware co-design for deep learning. arXiv preprint arXiv:2408.14158, 2024.

[7] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In Proceedings of the Seventeenth European Conference on Computer Systems, pages 472–487, 2022.

[8] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. Advances in Neural Information Processing Systems, 33:1877–1901, 2020.

[10] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 571–582, 2014.

[11] et al. DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

[12] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive,{Liveput-Optimized}{DNN} training on preemptible instances. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 1121–1139, 2024.

[13] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. arXiv preprint arXiv:2407.21783, 2024.

[14] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: A checkpointing system for training deep learning recommendation models. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 929–943, 2022.

[15] Apache Software Foundation. Hadoop Distributed File System. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2024.

[16] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 135–153, 2024.

[17] gRPC Team. grpc: A high performance, open source universal rpc framework get started! https://grpc.io/, 2024.

[18] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948, 2025.

[19] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-In-Time Checkpointing: Low cost error recovery from deep learning training failures. In Proceedings of the Nineteenth European Conference on Computer Systems, pages 1110–1125, 2024.

[20] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. Unicron: Economizing self-healing llm training at scale. arXiv preprint arXiv:2401.00134, 2023.

[21] Frank Herold, Sven Breuner, and Jan Heichler. An introduction to BeeGFS. ThinkParQ, Kaiserslautern, Germany, Tech. Rep, 2014.

[22] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of large language model development in the datacenter.

In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 709–729, 2024.

[23] Drew A Hudson and Christopher D Manning. GQA: A new dataset for real-world visual reasoning and compositional question answering. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pages 6700–6709, 2019.

[24] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 382–395, 2023.

[25] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. MegaScale: Scaling large language model training to more than 10,000 GPUs. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 745–760, 2024.

[26] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[27] Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. Revisiting reliability in large-scale machine learning research clusters. arXiv preprint arXiv:2410.21680, 2024.

[28] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. Proceedings of Machine Learning and Systems, 5:341–353, 2023.

[29] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In Proceedings of the Eighteenth European Conference on Computer Systems, pages 835–850, 2023.

[30] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. Easyscale: Elastic training with consistent accuracy and improved utilization on gpus. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–14, 2023.

[31] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. arXiv preprint arXiv:2006.15704, 2020.

[32] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal checkpointing: Efficient and flexible checkpointing for large scale distributed training. arXiv preprint arXiv:2406.18820, 2024.

[33] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. arXiv preprint arXiv:2412.19437, 2024.

[34] Avinash Maurya, Robert Underwood, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. DataStates-LLM: Lazy asynchronous checkpointing for large language models. arXiv preprint arXiv:2406.10707, 2024.

[35] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, fine-Grained DNN checkpointing. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 203–216, 2021.

[36] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using Megatron-LM. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–15, 2021.

[37] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pages 172–181. IEEE, 2020.

[38] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. Advances in neural information processing systems, 35:27730–27744, 2022.

[39] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 217–231, 2021.

[40] William Peebles and Saining Xie. Scalable diffusion models with transformers. In Proceedings of the IEEE/CVF International Conference on Computer Vision, pages 4195–4205, 2023.

[41] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. Advances in Neural Information Processing Systems, 36, 2024.

[42] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZERO: Memory optimizations toward training trillion parameter models. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16. IEEE, 2020.

[43] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv preprint arXiv:2403.05530, 2024.

[44] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pages 10684–10695, 2022.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[46] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. arXiv preprint arXiv:2409.19256, 2024.

[47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

[48] Konstantin V. Shvachko. HDFS Scalability: The limits to growth. USENIX Magazine, 35, 2010.

[49] Character AI Team. Character.ai. https://character.ai/, 2024.

[50] DeepSeek Team. Fire-flyer file system. https://github.com/deepseek-ai/3FS, 2025.

[51] Github Team. The world's most widely adopted ai developer tool. https://github.com/features/copilot/, 2024.

[52] Megatron Team. Dist checkpointing package. https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/dist_checkpointing.html, 2024.

[53] Midjourney Team. Midjourney. https://www.midjourney.com/home, 2023.

[54] NVIDIA NCCL Team. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2024.

[55] OpenAI Team. Creating video from text. https://openai.com/index/sora/, 2024.

[56] OpenAI Team. Introducing ChatGPT. https://openai.com/index/chatgpt/, 2024.

[57] OpenAI Team. Introducing OpenAI o1. https://openai.com/o1/, 2024.

[58] PyTorch Team. Getting started with Distributed Checkpoint (DCP). https://pytorch.org/tutorials/recipes/distributed_checkpoint_recipe.html, 2023.

[59] Suno Team. Bark is a transformer-based text-to-audio model created by suno. https://github.com/suno-ai/bark, 2023.

[60] Zarr Team. Zarr: chunked, compressed, n-dimensional arrays. https://zarr.dev/, 2024.

[61] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 497–513, Boston, MA, April 2023. USENIX Association.

[62] A Vaswani. Attention is all you need. Advances in Neural Information Processing Systems, 2017.

[63] veScale Team. veScale: A PyTorch Native LLM Training Framework. https://github.com/volcengine/veScale, 2024.

[64] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, pages 195–210, 2024.

[65] Yuxin Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Xinglin Pan, Yang Zheng, Xiaoyu Wu, Amelie Chi Zhou, Bingsheng He, and Xiaowen Chu. Reliable and efficient in-memory fault tolerance of large language model pretraining. arXiv preprint arXiv:2310.12670, 2023.

[66] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 364–381, 2023.

[67] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. arXiv preprint arXiv:2109.01652, 2021.

[68] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 533–548, 2020.

[69] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. GLM-130b: An open bilingual pre-trained model. arXiv preprint arXiv:2210.02414, 2022.

[70] Chi Zhang, Guangming Sheng, Siyao Liu, Jiahao Li, Ziyuan Feng, Zherui Liu, Xin Liu, Xiaoying Jia, Yanghua Peng, Haibin Lin, et al. A framework for training large language models for code generation via proximal policy optimization.

[71] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068, 2022.

[72] Juntao Zhao, Borui Wan, Yanghua Peng, Haibin Lin, Yibo Zhu, and Chuan Wu. QSync: Quantization-minimized synchronous distributed training across hybrid devices. arXiv preprint arXiv:2407.02327, 2024.

[73] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch FSDP: Experiences on scaling fully sharded data parallel. arXiv preprint arXiv:2304.11277, 2023.

[74] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. Swift: Expedited failure recovery for large-scale dnn training. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, pages 447–449, 2023.

[75] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. arXiv preprint arXiv:2406.11931, 2024.

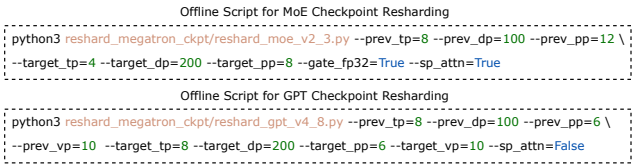


Figure 15: Examples of running offline scripts for checkpoint resharding.

A More Background on Maintaining Resharding Scripts

Examples of offline resharding with customized scripts are given in Fig. 15. Customizing offline scripts is labor-intensive. For GPU state resharding, the offline scripts must cover all different components in the LFM and its optimizer and cater to the diverse behavior of each component under different parallelism strategies. For example, with Tensor Parallelism (TP), GEMM operators in attention and MLP blocks are sharded along different dimensions, while other operators like LayerNorm [8] are replicated across GPUs. When hybrid 3D parallelism [36] is adopted, TP-sharded tensors of one layer (module) in the distributed optimizer are first flattened and then merged before being sharded according to the designated Data Parallelism (DP) degree. Offline scripts must implement resharding logic that is tightly coupled with combinations of model (optimizer) components and parallelism strategies. Additionally, special algorithmic optimization techniques such as GQA [23] and MLA [75] change the tensor layout of certain operators (e.g., the query-key-value projection GEMM operator in the attention block), necessitating corresponding resharding support. To handle various cases in our production environment, our largest script even includes 3193 lines of Python code. This complexity results in significant engineering efforts for both development and maintenance.

B Efficient Integrity Guarantee

A complete checkpoint consists of multiple files stored by different workers. The failure of any single worker can corrupt the entire checkpoint. To prevent such issues, a barrier mechanism is essential to achieve atomic save/load operations among all training workers.

Checkpointing modules in training frameworks like Megatron-LM [47] rely on the `barrier` function in `torch.distributed` to perform integrity checks. This approach synchronizes training workers to ensure that all checkpoint saving/loading operations are finished. We observed that when scaling training to involving approximately 10,000 GPUs, this behavior results in stalls of about 20 seconds each time. To address this inefficiency, we re-implement the `barrier` function using the aforementioned method (gRPC with tree-based communication topology) and conduct the integrity checks asynchronously, effectively eliminating the blocking time. We also incorporate upload/download retry mechanisms in ByteCheckpoint's I/O workers and integrate failure logging, which records the exact stage of failure within the checkpoint saving/loading pipelines in workers who fail to complete checkpointing tasks.

C Average ETTR Calculation

Failures that happened during training introduce progress loss and the last checkpoint loading overhead. Assume failures are evenly distributed within one checkpoint interval [66], the best case is that a failure occurs just after the completion of a checkpoint end-to-end saving procedure whereas the worst case is just before that. Given the per iteration training time T_{iter}, checkpoint interval N, end-to-end checkpoint saving time T_{save} and loading (resharding) time T_{load}, we derive the average wasted time T_{wasted} as:

$$T_{wasted} = T_{save} + T_{load} + \frac{N \cdot T_{iter}}{2} \tag{1}$$

Therefore, the average ETTR is:

$$ETTR = 1 - \frac{T_{wasted}}{T_{save} + T_{load} + N \cdot T_{iter}} \tag{2}$$

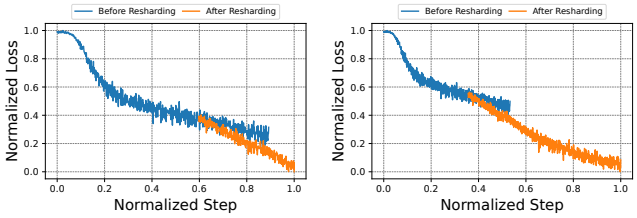
The end-to-end ETTR results presented in Table 4 are averaged across standard loading and resharding settings.

D Checkpointing Overhead Breakdown

We break down the end-to-end checkpoint saving time ($\mathbf{T_{save}}$) for rank 0 by dividing it into several phases and investigating the overhead of each part. The results are depicted in Table 9, where $\mathbf{T}^{\mathbf{First}}_{\mathbf{Plan}}$ denotes the initial planning cost while $\mathbf{T}^{\mathbf{Cache}}_{\mathbf{Plan}}$ refers to the overhead when using the plan and metadata cache for subsequent checkpointing operations. Costs of other phases are averaged over various checkpointing steps. We find that the communication overhead of planning increases as training scales up, leading to significant stalls. Thanks to the caching strategy (Sec. 4.1), it becomes a one-time expense for each (resumed) training session. In addition, the adoption of the pinned memory CPU pool renders the blocking time of D2H almost negligible, while our asynchronous engine pipeline overlaps the execution time of serialization, shared memory dumping, and HDFS uploading, diminishing the end-to-end time. Furthermore, the load-balancing mechanism exploits the capabilities for parallel uploading within each DP group and achieves more performance gains at larger training scales (e.g., the uploading speed of model states with 4800 GPUs is 3.03× faster than that with 2400 GPUs).

Table 9: Detailed overhead breakdown of the checkpoint saving procedure for rank 0.

Model and Framework	Source #GPUs	State	T^{First}_{Plan} (s)	T^{Cache}_{Plan} (s)	T_{D2H} (s)	$T_{Serialize}$ (s)	T_{Dump} (s)	T_{Upload} (s)
vDiT 4B	32	Model	0.05	0.00	0.15	0.52	0.50	11.04
		Optimizer	0.65	0.00	0.37	0.93	1.03	14.60
FSDP	128	Model	0.35	0.00	0.06	0.29	0.12	10.57
		Optimizer	0.89	0.00	0.06	0.29	0.26	9.85
tGPT 70B	2400	Model	8.39	0.00	0.08	1.93	0.48	6.67
		Optimizer	5.02	0.00	0.17	2.00	2.23	2.30
Megatron-LM	4800	Model	17.09	0.00	0.08	1.95	0.47	2.20
		Optimizer	6.46	0.00	0.03	1.69	0.35	1.67



(a) DP resharding from TP=1, DP=4, PP=4 to TP=1, DP=8, PP=4.

(b) Hybrid resharding from TP=1, DP=4, PP=4 to TP=2, DP=8, PP=2.

Figure 16: Resharding correctness verification.

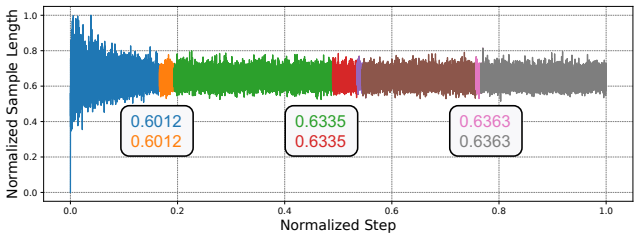


Figure 17: The normalized sample length curves of the dataloader with multiple training restarts.

E More Resharding Correctness Experiments

The normalized loss curves for DP and hybrid resharding correctness are shown in Fig. 16. In the case of DP and hybrid resharding, since we also increase the global batch size, the loss curve after resharding declines more rapidly.

We further show that ByteCheckpoint can achieve bit-wise correct resuming for dataloader states. Since the RNG state is fixed, correct resumptions should yield the same data sampling trajectory. Therefore, we use the normalized data sample length curve (Fig. 17) for evaluation. As highlighted, the normalized data sample length before and after restarts is identical.

F Related Work

Checkpointing frameworks. Some industrial initiatives focus on developing checkpointing systems for deep learning. Prior to DCP [58], PyTorch provided `torch.save` and `torch.load` APIs for local checkpoint management, without native resharding support. DCP [58] introduced resharding capabilities for FSDP but lacks support for parallelism strategies such as TP and PP. DeepSpeed-UCP [32] provides a unified format for DeepSpeed checkpoints and offers resharding capabilities through offline scripts. Megatron MCP [52] builds upon the workflow of DCP [58] and extends storage options to formats like Zarr [60]. All these frameworks provide limited support for various parallelism strategies and training frameworks, and their I/O performance does not scale well in large-scale training.

Checkpointing optimizations. Several works [10, 37] investigated reducing checkpointing costs from different perspectives. Check-N-Run [14], designed for recommendation models, employs differential checkpointing to store only the modified portions of the model, alongside quantization to reduce checkpoint size. CheckFreq [35] pipelines snapshot and save operations with computation to minimize checkpoint stalls and introduces an online algorithm for tuning checkpoint frequency to further lower costs. Gemini [66] advocates for in-memory checkpointing with inter-machine backup for fast recovery, interleaving checkpoint communication with training traffic to enable frequent checkpoints per iteration. JIT-Checkpointing [19] adopts just-in-time checkpointing for low-cost error recovery. In industrial use cases, it is crucial to store checkpoints in separate persistent storage for various tasks such as auto-evaluation [22], hyper-parameter tuning, and model debugging. ServerlessLLM [16] focuses on the demand for loading optimizations which is critical in serverless inference scenarios. It proposes a chunk-based, multi-tier loading pipeline to accelerate checkpoint loading. Unlike existing solutions, ByteCheckpoint champions both optimized I/O performance and flexible load-time checkpoint resharding, supporting general LFM development.

Checkpoint representation. PyTorch's `torch.save/load` functionality relies on `pickle` for serialization and deserial-

ization. This format lacks crucial tensor shard metadata, such as global shape information, precluding automatic resharding. DCP [58] addresses this limitation by introducing a disaggregated format that separates metadata from tensor data. This metadata, which includes global shape and offset details, enables automatic resharding in DCP. Tenplex [64] introduces Parallelizable Tensor Collection (PTC) to flexibly represent and transform tensor states across diverse parallelism configurations. To build a PyTorch-native checkpointing system, ByteCheckpoint adopts the representation of DCP and incorporates necessary adaptations to handle irregular tensor sharding. Array-based storage systems like Zarr [4] and TensorStore [3] allow tensors to be saved as individual arrays, supporting concurrent reads and writes. MCP [52] supports distributed checkpointing using the Zarr format. For secure and efficient tensor storage, Safetensors [2] is a file format for saving tensors safely and loading them efficiently. To improve compatibility with the Hugging Face open-source ecosystem, ByteCheckpoint incorporates functionality to export checkpoints in the Safetensors format.

Storage systems for LFM. Large-scale checkpointing requires robust storage systems. The LLaMA 3.1 report [13] highlights Tectonic [39], Meta's general-purpose file system, as the backbone for storing checkpoints during pre-training. One of the key challenges for this system is managing high-frequency checkpoint writes. Similarly, FireFlyer [6], the AI-HPC system developed by DeepSeek [11, 18, 33] for LFM training, utilizes the custom-built 3FS Distributed File System [50], which is akin to other parallel file systems like BeeGFS [21], to manage checkpoint storage.

Elastic training systems. Some orthogonal efforts are devoted to enhancing the elasticity of DL training jobs [7, 12, 24, 30, 61, 74]. For instance, Varuna introduces *job morphing* to reconfigure training jobs in both pipeline and data parallelism degrees, without changing the hyper-parameters. Bamboo [61] leverages cross-stage redundant computations into the pipeline for elastic training on spot instances. Oobleck [24] implements a pipeline instantiation mechanism via predefined templates to tolerate concurrent failures in different pipelines. These works are limited to specific parallelism strategies (e.g., DP without ZeRO), while ByteCheckpoint does not assume any specific parallelism, supporting real-world production. The flexibility of elastic training systems can be significantly enhanced by adopting our checkpoint representation for training state management.

Mowgli: Passively Learned Rate Control for Real-Time Video

Neil Agarwal[¶] Rui Pan[¶] Francis Y. Yan[†] Ravi Netravali[¶]

[¶]Princeton University [†]University of Illinois Urbana-Champaign

Abstract

Rate control algorithms are at the heart of video conferencing platforms, determining target bitrates that match dynamic network characteristics for high quality. Recent data-driven strategies have shown promise for this challenging task, but the performance degradation they introduce during training has been a nonstarter for many production services, precluding adoption. This paper aims to bolster the *practicality* of data-driven rate control by presenting an alternative avenue for experiential learning: leveraging purely existing telemetry logs produced by the incumbent algorithm in production. We observe that these logs contain effective decisions, although often at the wrong times or in the wrong order. To realize this approach despite the inherent uncertainty that log-based learning brings (i.e., lack of feedback for new decisions), our system, Mowgli, combines a variety of robust learning techniques (i.e., conservatively reasoning about alternate behavior to minimize risk and using a richer model formulation to account for environmental noise). Across diverse networks (emulated and real-world), Mowgli outperforms the widely deployed GCC algorithm, increasing average video bitrates by 15–39% while reducing freeze rates by 60–100%.

1 Introduction

Real-time video conferencing is integral to our daily lives, with widespread use cases across many societal pathways including healthcare, education, gaming, and more. Key to their functionality are the rate control algorithms (e.g., Google Congestion Control or GCC [21]) that conferencing platforms employ. These algorithms are tasked with quickly (e.g., every 50 ms) characterizing network performance based on recent transmissions and selecting a target bitrate for the upcoming frames that maximizes content quality without introducing undue latency or stalls. This value is then shared with the local video codec which performs best-effort compression of the raw frames to match the target prior to transmission.

Recent years have witnessed a flurry of proposals for improving conferencing quality via improved rate control. Most notably, data-driven approaches have shown how reinforcement learning (RL)-based algorithms can substantially outperform GCC's hand-tuned variants [54–56]. As in other networked-system domains [14, 35, 51, 52], the key is in making better use of dense application- and network-layer feedback signals to enable bitrate changes that more closely track rapid network fluctuations in the wild—a key challenge for GCC [21, 54–56].

Yet, despite their promise, data-driven rate control algorithms have seen minimal adoption in practice. Our conversations with operators of large-scale video conferencing platforms reveal that the primary showstopper is the impact that these schemes have on the performance or quality of experience (QoE) observed by real users. Indeed, the experiential learning that these schemes pursue *in real conferencing sessions* fundamentally involves exploring different rate decisions—both good and bad for QoE—and observing their effects in certain scenarios. Our experiments show that this trial-and-error process can increase video freeze rates up to 79% and degrade video bitrates by up to 77%, yielding unacceptable QoE (§2.2). Data-driven strategies could forego such user-facing training by relying on simulators and emulators, but this may jeopardize efficacy in production settings due to the "simulation to reality" gap [12, 20, 27, 51, 55].

This paper aims to build on recent data-driven rate control schemes, not by designing more performant algorithms, but instead by *bolstering their practicality*. Our key insight is that the fine-grained telemetry logs that production conferencing platforms routinely capture for debugging and retrospective optimization [1, 2, 4, 7, 10] already embed sufficient insight to guide data-driven algorithms to outperform their hand-tuned counterparts. The reason is that, while algorithms like GCC struggle to quickly match fluctuating network bandwidths, they often adjust target rates in the appropriate direction with delay (Fig. 1). Consequently, we find that simply reorganizing the same decisions that GCC makes during video sessions (without exploring any alternate rate values) boosts conferencing bitrates by 19% and decreases freezes by 80% (§3.3).

We present **Mowgli**, an end-to-end system that realizes such log-based learning to practically enhance rate control in video conferencing. Mowgli starts by representing production telemetry logs that reflect the behavior and performance of a deployed algorithm (e.g., GCC) as more traditional (state, action, reward) tuples for RL. These logs then guide Mowgli's experiential learning, which follows the soft actor-critic algorithm [30] to develop a lightweight neural network for rate control. Training occurs entirely offline using only logs (i.e., no video playback or simulation), and the resultant model is shipped to clients for deployment.

Though conceptually straightforward, learning in this offline manner faces several challenges centered around *uncertainty* (§3.4). As noted above, the feasible learned improvements are rooted in discovering alternate sequences of logged actions that perform better in certain scenarios. However, assessing such sequences is fundamentally risky as they involve applying previously-seen actions to new scenarios without guarantees that the benefits will port, i.e., unlike traditional RL, we lack direct feedback for these new sequences. This limitation is worsened by the fact that logs in confer-

encing systems (1) typically reflect singular deterministic policies with little variation in action for a given scenario, and (2) are inherently noisy in that observed performance can be influenced not only by rate decisions, but also external phenomena such as codec behavior and stochastic network variations [28, 50, 51, 53, 56].

To manage uncertainty, Mowgli incorporates two key techniques. First, when estimating the consequence of an action in a given scenario, Mowgli takes a *conservative* approach—it lowers the estimated return if no similar state-action pair has been observed before, and proceeds only when the performance improvement outweighs the risk. Second, rather than estimating a single expected outcome, Mowgli explicitly tackles environmental variance by learning a *distribution* over all possible outcomes. This distributional perspective provides Mowgli with richer insights, enabling more informed decision-making. We detail the concrete algorithms in §4.2.

We evaluated Mowgli on a diverse set of emulated and real-world networks spanning 3G–5G cellular and wired broadband links. Overall, we observe that Mowgli consistently outperforms GCC, increasing average video bitrates by 15–39% while reducing freeze rates by 60–100%. Further, Mowgli's wins closely mimic those of recent (impractical) online RL algorithms [27, 54, 55], falling within 0.5–13.1% and 0–19% for average bitrates and freeze rates, respectively.

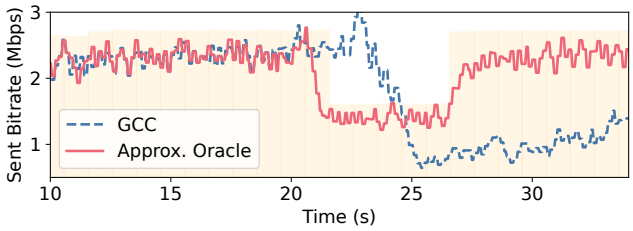
2 Background & Motivation

2.1 Prior Work

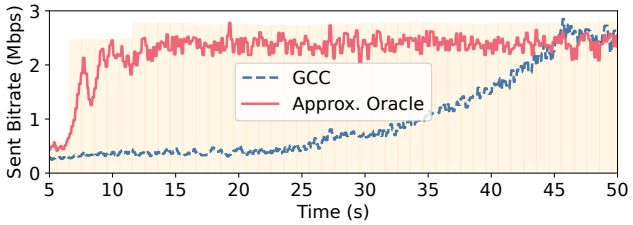
Rule-based heuristics are suboptimal. Today's video conferencing applications (e.g., Microsoft Teams, Google Hangouts, Zoom) rely on rule-based algorithms to provide network bandwidth estimates and guidance as to how to tune the encoding and sending bitrates over time. A widely used, publicly available algorithm is Google Congestion Control (GCC) [21]. GCC first characterizes the current network usage based on recent packet delay and loss measurements, and then updates the target bitrate according to a fixed set of rules (e.g., when the observed packet loss is less than 2%, increase the target bitrate by 5%, or when the system is in a "decrease" state, reduce the target bitrate by 15%).

Google Congestion Control (and other rule-based heuristics) have been widely observed to be suboptimal [27, 28, 54–56]. Recent reports have found that when using state-of-the-art rule-based heuristics, over 20% of over a million video conferencing sessions experience poor performance [56]. Digging deeper, we find that GCC performs particularly poorly in highly dynamic network conditions, where the available bandwidth fluctuates (e.g., in cellular networks). The general hardcoded set of rules fails to fully leverage dense application/network signals that are necessary for predicting the appropriate bitrate adjustments in highly variable regimes.

As a result, for example, when bandwidth drops, GCC can fail to rapidly and appropriately adjust the bandwidth, unnecessarily overshooting the available network capacity and incurring video freezes (Fig. 1a). Further, after an intermittent drop in bandwidth, GCC can be unnecessarily delayed in fully ramping up to the available bandwidth (Fig. 1b). As quality expectations for video conferencing steadily increase with its growing ubiquity, there is a pressing need to find improved rate control algorithms over today's rule-based heuristics.



(a) GCC overshoots network capacity after a bandwidth drop, causing video freezes.



(b) After an intermittent bandwidth drop, GCC ramps up slowly, leading to suboptimal bandwidth utilization.

Figure 1: Examples of GCC's pitfalls, which occur primarily in dynamic networks. To illustrate potential performance improvement opportunities, we plot an oracle algorithm.

Data-driven approaches are promising. In response, the community has explored data-driven approaches to generate alternative rate control algorithms (e.g., R3Net [27], OnRL [55], Loki [54]). Such machine learning-based approaches are particularly skilled at extracting patterns from highly-dense information and have been shown to deliver non-trivial improvements over today's state-of-the-art rule-based heuristics. Given the challenge of explicitly identifying the exact action an algorithm should take in each scenario and using standard supervised learning techniques to learn that mapping, recent proposals have opted to leverage reinforcement learning (RL). An RL agent learns by interacting with an environment and leveraging feedback to iteratively adjust its behavior to maximize cumulative reward over a horizon. In rate control for video conferencing, the RL agent outputs bitrate updates and leverages the transport/application layer feedback to update its decision-making policy.

2.2 Motivations

As confirmed by prior work [27, 54, 55] and our results in §5, data-driven methods can deliver significant wins over rule-based heuristics and are crucial to enabling the effective rate control that today's video conferencing applications demand. Surprisingly, despite their potential benefits, these solutions have yet to gain traction in production deployments. In conversations with the operators of major production video conferencing deployments, these solutions do not meet the

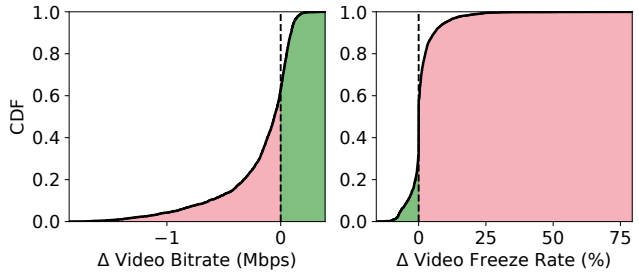


Figure 2: Distribution of changes to video QoE observed (relative to GCC) during online RL model training. Improvements are shaded in green; degradations are shaded in red.

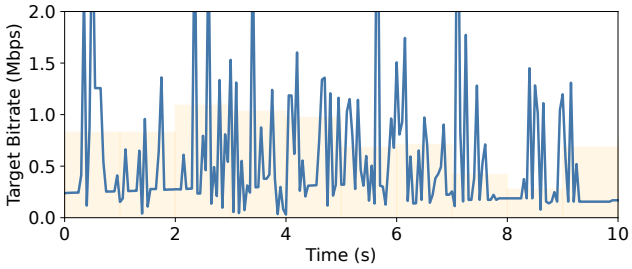


Figure 3: Example disruptive behavior observed during online RL training; network bandwidth is shaded in yellow.

practicality constraints of production environments. In particular, the biggest concern is the disruption of client video conferencing sessions during model training.

Fundamental to existing solutions is their use of reinforcement learning, a trial-and-error learning mechanism that trains a model by iteratively interacting with an environment and updating the model based on the interaction outcomes. For existing solutions, randomly initialized RL agents (models) are placed directly on client devices and are allowed to dictate the bitrate decisions for real user-facing video conferencing sessions. Over time, the models gain experience and converge to a high-performing policy; however, this comes at the cost of disruption to the quality of video conferencing calls during the training process. Prior approaches explicitly encourage exploration of different actions and behavior during training (by adding an entropy bonus to the learning objective); the bitrate decisions taken during exploration may be far from the ideal bitrate decision for that particular scenario and dramatically degrade the quality of a video call.

Quality disruptions of online learning. To observe how the performance of video conferencing sessions is affected during training, we ran a series of experiments. We train an online RL algorithm in our video conferencing testbed and periodically record the QoE metrics of affected sessions. More details about our video conferencing testbed, online RL implementation, and QoE metrics can be found in §5. Note that our implementation includes the temporary fallback mechanism introduced by OnRL [55] which switches back to the rule-based heuristic when overuse is detected in the hope of reducing catastrophic behavior.

Fig. 2 shows the effects on performance (relative to GCC) during model training for two key QoE metrics. We find that 62% of calls experience a worse average video bitrate, with degradations as low as −1.9 Mbps (for context, the average bitrate of a call is 1.03 Mbps). 43% of calls experience higher video freeze rates, with freeze rate increases as much as +79%. Intuitively, these performance degradations are due to the exploratory nature of online RL mechanisms. Fig. 3 highlights some of the disruptive behavior that causes performance degradations: repeated switching between low and high bitrates, bandwidth underutilization, and overaggressive bitrate ramp-ups. Fallback mechanisms fail to resolve these issues because they are only activated once catastrophic behavior is detected. Further, these degradations are exacerbated by the iterative development process of model optimization across different model designs and hyperparameters—the costs of training (e.g., QoE degradations) will be incurred every time.

2.3 Alternative Approaches

Training in simulation. Earlier work [27, 55, 56] has investigated training policies in simulated or emulated environments to avoid the user-facing quality degradations seen in previous approaches. However, they found that these models perform poorly in production due to the "simulation-to-reality" gap [12, 20], where simulated environments fail to fully and accurately capture the complexity of production systems and any slight deviation in system dynamics can lead to cascading performance issues. For more details, we refer readers to prior reports [27, 55]. Improving simulation fidelity in networked systems remains an active area of research [16, 19].

This gap also explains why starting with a "pre-trained model" from simulation and then fine-tuning it online in production is problematic. The disparity between simulated and real environments, combined with the significant experience required for fitting an RL model to a new distribution of environments, means the model typically needs major revisions rather than minor adjustments. This would disrupt production services and negate the benefits of "pre-training". Further, in our review of related work [27, 54, 55], we found that prior approaches always trained models from scratch in the deployment setting, starting with completely untrained models.

Tuning GCC knobs. Prior work [15] has shown the benefits of using data-driven parameter tuning for ABR heuristic algorithms. However, applying this approach to GCC faces three main challenges. First, GCC bakes in handcrafted rules that are inherently heuristic and suboptimal, thereby constraining the potential impact of hyperparameter tuning. Second, GCC does not support extracting insights from a richer set of network signals, depending instead primarily on a single, unreliable network signal (the gradient of queuing delay). Lastly, its codebase includes hundreds of tunable knobs, making it unclear how to prioritize or manage them effectively.

2.4 Summary

Recent data-driven approaches demonstrate considerable improvements over their rule-based counterparts. However, they

face minimal adoption in production due to the disruptive nature of their interactive trial-and-error-based online learning approach. Alternatively, policies trained in simulation perform poorly when deployed to production due to the "simulation to reality" gap. How to capitalize on the performance improvements offered by data-driven approaches while adhering to the constraints of production environments is an open question faced by video conferencing system operators.

3 Vision: Learning By Observation

In this paper, we claim that there is a viable path forward, but it requires rethinking the end-to-end design of data-driven rate control systems, from how we source the data to how we leverage and learn from it.

3.1 The Data of Prior Approaches

Prior data-driven approaches collect training data by iteratively deploying and continually updating DNN-based rate control policies in production video conferencing systems, and logging the resulting behavior and outcomes [27, 54, 55]. These logs are shipped from users to a central server, where they are processed and then fed to a training algorithm that updates the weights underlying the current version of the rate control policy. The updated weights are shipped out to the clients and the next round of training proceeds.

At the central server, data processing of the logs involves extracting structured sequential data, i.e., series of (state, action, reward) tuples. A (s_t, a_t, r_t) tuple represents the following: at a given time t, the rate control policy takes as input recently observed information about application and transport layers (captured as vector s_t) and outputs an updated target bitrate a_t. The effect of updating the target bitrate to a_t on application performance (e.g., the change in video throughput, freezes, frame delay, etc.) is quantified as r_t. Such sequences of (state, action, reward) tuples offer a structured way to reflect the experiences and effects of a rate control policy.

This data provides critical feedback to the training process by reinforcing decision-making that leads to good behavior (high reward) and penalizing decision-making that leads to poor behavior (low reward). Recall that the training algorithm initially knows nothing about the environment or the ideal behavior. As a result, the initial rate control policies take random actions. This exploratory behavior helps build a growing set of diverse experiences, each trajectory corresponding to a different sequence of possible decisions. Over time, as the weights of the rate control policies are updated and additional logs are collected, the model identifies high-potential paths and ultimately, converges to a policy that maximizes the cumulative reward. In the process, each step of deployment and data collection provides a critical feedback loop, enabling the training algorithm to (1) test out new behaviors, and (2) correct any misunderstandings in the learned model.

However, as quantified in §2.2, accumulating this rich and diverse dataset requires deploying partially trained policies to production environments and disrupting the quality of video conferencing sessions for users. From the perspective of a production deployment operator, this is not a viable option.

3.2 An Alternative Source of Data

In this paper, we posit that there is a viable alternative source of data: the experiences of the rule-based algorithm *currently deployed* in production settings for rate control (e.g., Google Congestion Control). Unlike prior approaches, this data source can be obtained without deploying partially trained or untested rate control policies, thereby avoiding the disruptions outlined in §2.2. Instead, the data is collected from a rate control heuristic already in production, where it is logged at a fine granularity for purposes such as monitoring, debugging, and improving the heuristic algorithm [2, 6]. Similar to previous methods, these logs can be post-processed to extract sequences of (state, action, reward) tuples, capturing the experiences of the deployed rate control policy.

However, ultimately, these logs reflect the behavior of a single, static rate control policy (production systems typically deploy a single algorithm, e.g., the in-house rule-based heuristic). The logs of prior approaches, on the other hand, reflect the behavior of hundreds of rate control policies, learned and deployed over hundreds of epochs of training. As previously described, the diversity in behaviors enables the algorithm to learn an effective rate control policy. This begs the question: how we can learn a better rate control policy from the experiences of a single rate control policy?

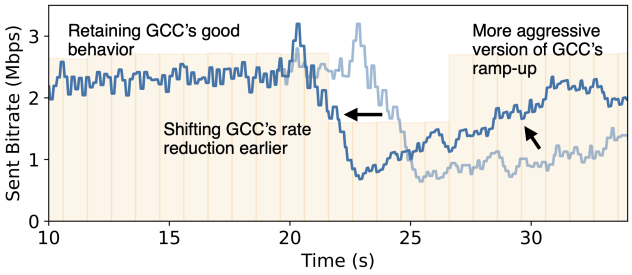
Besides the interactive RL approaches of prior work, there exist alternative data-driven approaches such as imitation learning (e.g., behavior cloning) [31]. However, the goal of these methods is to learn the *same* behavior as reflected in the training data; our goal is to learn a *better* rate control strategy.

3.3 Opportunity

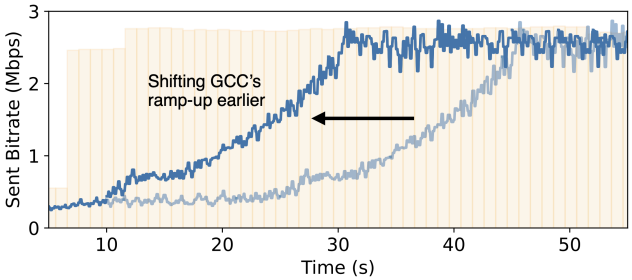
In this section, we introduce a novel approach to learning from the logs of GCC; we start by describing a key observation about the nature of GCC's shortcomings.

As discussed in §2.1, GCC's suboptimal performance stems from delayed or overly cautious responses to bandwidth changes. During sudden bandwidth drops (Fig. 4a), GCC's reliance on hand-tuned thresholds of limited network signals delays congestion detection and subsequent corrective actions. This lag results in network overutilization and video freezes. When network capacity increases, GCC exhibits overly cautious ramp-up behavior, hesitating to increase the bitrate for fear of overshooting the available capacity (Fig. 4b).

Notably, despite these issues, *GCC eventually makes the correct directional adjustments*— ramping down after detecting congestion or gradually increasing its bitrate when the network allows. This observation is crucial: it implies that, within the very logs where GCC performs poorly, there are also *positive* examples that demonstrate how timely or more aggressive actions could have prevented unnecessary freezes or underutilization. These examples present a unique opportunity to develop an improved rate control algorithm directly



(a) After the network bandwidth drops at `t=22`, it takes 3 seconds for GCC's sending bitrate to catch up. By shifting GCC's bitrate reduction earlier, we can reduce the degree of network overutilization.



(b) After the network bandwidth increases at `t=7`, it takes 40 seconds for GCC's sending bitrate to catch up. By shifting GCC's bitrate ramp-up earlier, we can increase network utilization.

Figure 4: Potential improvements by rearranging sequences of actions within GCC's logs.

from GCC's logs. Instead of considering entirely new actions or behaviors, we can improve through careful selection and application of actions already demonstrated in the logs.

For example, in Fig. 4a, as GCC eventually reduces its bitrate following a bandwidth drop, the trace captures a range of actions: initially, actions at the same bitrate and subsequently, actions at lower bitrates. These logs also capture detailed network telemetry data reflecting the state before and after each action. Together, this data provides valuable examples that can be analyzed and optimized. Specifically, these examples can be used to (1) identify congestion signals more effectively and (2) select lower target bitrates more promptly. Similarly, as illustrated in Fig. 4b, when GCC (conservatively) increases its bitrate in response to improved bandwidth, the telemetry traces capture a sequence of steadily increasing bitrate adjustments. These actions, paired with their corresponding telemetry signals, provide examples of different responses and ultimately, a path to achieving improved network utilization.

Both scenarios illustrate that, despite capturing suboptimal decisions, GCC's logs contain valuable data for achieving critical improvements. By systematically analyzing GCC's behavior alongside dense telemetry signals, it becomes possible to reorder its actions into a more optimal sequence without the need to devise entirely new strategies. The central insight lies in leveraging GCC's imperfect but fundamentally directionally correct decision sequences as foundational building blocks for constructing a more effective rate control policy.

To quantify the potential gains from this log-based learning approach, we implemented an approximate oracle algorithm that has access to ground-truth network dynamics but is restricted to the set of actions that appear in a given GCC log. On the network trace depicted in Fig. 4a, the oracle achieves a 52% increase in video bitrate and a 98% reduction in freeze rates. For the scenario in Fig. 4b, it achieves an 80% increase in video bitrate and a 79% reduction in freeze rates. Across our entire corpus of network traces (§5), this oracle-based approach yields a 19% improvement in video bitrate and an 80% decrease in freeze rates compared to GCC.

3.4 Challenges

However, realizing this approach in a way that is both practical and effective while adhering to the constraints of a production deployment setting requires addressing two key challenges.

Challenge #1: Lack of feedback. Ultimately, the goal is to learn an improved rate control policy without disrupting users (the main pitfall of prior approaches). However, producing an improved rate control policy requires learning a strategy that deviates from the behavior seen in the GCC logs (e.g., an alternative sequence of actions). Learning this alternative (and better) strategy requires reasoning about (and extrapolating) the expected outcomes of alternative behaviors. This is a risky proposition without access to feedback (i.e., testing the new strategy and validating the prediction). The greater the deviation, the greater the potential risk of extrapolation error; any errors in extrapolating will compound, as the resulting deviation will lead to more deviation. Ultimately, this comes at the detriment of the performance of the learned policy. In the field of learning sequential decision-making without feedback, this phenomenon is referred to as "distribution shift" [33]. In our evaluation ablation studies (§5.5), we find that failing to address distribution shift can dramatically cause performance degradations, increasing P90 video freeze rates over 12×. To address this issue in the context of learning improved rate control policies, we need a way to effectively balance decision-making deviations with risk mitigation.

Challenge #2: Environmental variance. Further complicating the ability to develop an understanding of how any given target bitrate update affects observed outcomes (rewards) is the presence of external phenomena outside the control of the bitrate decision-making policy that affect application behavior and ultimately, the outcomes of a given bitrate update. Unlike prior work [27, 56] that leverages emulation or simulation-based systems and can control for (or eliminate) the amount of noise introduced, production-based deployments do not have that luxury. Concretely, we find that this noise manifests in two ways. First, video conferencing applications apply additional downstream application logic after consuming a target bitrate update from the bitrate controller; this logic affects the *achieved* encoding and sending bitrate [28, 56]. Second, network conditions change rapidly (often in unpredictable ways [50, 51, 53])—taking the same action at the same state in two different instances could result in different outcomes because of rapid changes in network conditions (i.e., the band-

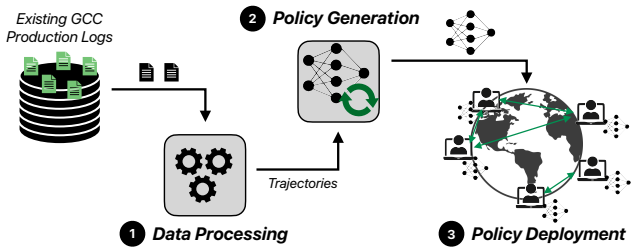


Figure 5: Overview of Mowgli's 3 phases: (1) data processing for extracting trajectories from GCC telemetry logs, (2) policy generation for model training, and (3) policy deployment.

width dropped). Ultimately, this makes it difficult to determine whether differences in the observed outcomes of two bitrate decisions arise from external factors or the effectiveness of the target bitrate decision itself. In our evaluation studies (§5.5), we find that failing to address the effects of environmental variance can increase P90 video freeze rates over 10×. To have an effective solution, we need a way to explicitly express and account for environmental noise.

4 Design

Fig. 5 describes Mowgli's workflow. First, Mowgli consumes existing production logs or builds on existing logging instrumentation to extract telemetry signals and generate trajectories of (state, action, reward) tuples that reflect the experiences of the underlying rate control algorithm (§4.1). Then, Mowgli uses these trajectories to train a lightweight neural network for rate control (§4.2). Training occurs entirely offline using only telemetry logs, i.e., no video playback or simulation is required; the resulting policy is then deployed on client devices (§4.3). We describe implementation details in §4.4.

4.1 Data Collection & Processing

Mowgli operates on production telemetry data that reflects the experiences of in-house rate control algorithms. These logs are typically already captured for other purposes such as debugging, observability, and quality assurance [2, 6]. For example, recently released logs from Microsoft Teams capture periodic (every 60 ms) application and transport layer metrics (e.g., packet loss, packet delay, received bitrate, and target bitrate) along with session-level QoE metrics (e.g., duration of video freezes, average media bitrate received) [2]. If the existing production logs do not provide the necessary data, we can build on existing instrumentation platforms to extract the additional data (e.g., Zoom, Microsoft Teams, WebRTC have built-in logging capabilities [1, 2, 4, 7, 9, 10]). Once logs are aggregated across clients to a central server, we apply Mowgli's processing logic to extract trajectories of (state, action, reward) tuples.

We define the state vector and action based on prior work [27, 54, 55]. The state vector is a window of periodically captured transport and application-level statistics (we use a window of 1 second). We describe the state vector in Table 1. We find that augmenting the state vector inputs with four additional features further improves performance: the previous bitrate action, the minimum RTT observed so far, the number of timesteps since the last transport feedback report was received, and the number of timesteps since the last packet loss report was received. We measure the impact of these additional features in §5.5. The action is the updated target bitrate that is consumed by the application. We define the reward as a function of the achieved bitrate, the average RTT, and the average packet loss. Inspired by prior work, we leverage the following formulation:

State Vector Inputs
Sent Bitrate
Acknowledged Bitrate
Previous Action
One-Way Packet Delay
One-Way Packet Delay Jitter
Inter-packet Arrival Delay Variation
Round Trip Time
Minimum Round Trip Time Observed So Far
Timesteps since Last Transport Feedback Report
Packet Loss
Timesteps since Last Packet Loss Report

Table 1: State vector of transport and application-layer statistics logged every ~ 50 ms.

$$R = \alpha \cdot throughput - \beta \cdot delay - \gamma \cdot loss \quad (1)$$

We normalize throughput to the range (0, 6 Mbps) and the delay to (0, 1000 ms); we set α to 2, β to 1, and γ to 1.

4.2 Policy Generation

Next, Mowgli leverages the aggregated and preprocessed logs to generate improved rate control policies. In §3, we described an approach to improve upon GCC by rearranging the actions within a GCC log. In this section, we explain how to realize this approach in practice. First, we describe Mowgli's use of a lightweight neural network to represent a rate control policy on client devices; we then detail how we overcome the challenges of realizing this approach presented in §3.4.

Leveraging neural networks. To reason about potential actions for a scenario, we need a way to estimate the expected outcomes for (state, action) pairs. We turn to neural networks that we can train offline (using GCC logs) and subsequently deploy to client devices. In particular, we leverage a state-of-the-art learning algorithm, Soft Actor-Critic (SAC) [30].

SAC consumes the trajectories of (state, action, reward) tuples previously experienced by GCC and trains two complementary functions (both represented by parameterizable neural networks): the actor and the critic. The actor network (π_θ) learns a deterministic policy that maps states to actions; the critic network (Q_ϕ) evaluates the expected long-term reward (return) for a given (state, action) pair. SAC leverages the critic to provide a learning signal (via the Q-value) for

the actor, enabling it to improve the policy by following the gradient of expected returns.

Following standard Q-Learning [48] techniques, we train the critic by minimizing the Mean Squared Bellman Error:

$$\mathbb{E}_{(s_t,a_t,r_t)\sim D}\Big[(Q_\phi(s_t,a_t)-(r_t+\gamma\cdot\max_{a'}Q_\phi(s_{t+1},a')))^2\Big] \quad (2)$$

The actor network is updated by maximizing the Q-value predicted by the critic:

$$\max_\theta \mathbb{E}_{s\sim D}[Q_\phi(s,\pi_\theta(s))] \quad (3)$$

We train Q_ϕ and π_θ in tandem, repeatedly iterating through the dataset of (state, action, reward) tuples. The pseudocode in Algorithm 1 describes the key components of the dual training process: updating the critic, and then updating the actor. Upon convergence (and during inference), we only need to retain π_θ: given state s, return $\pi_\theta(s)$.

Algorithm 1: Soft Actor Critic Algorithm

Input: Corpus of GCC Observations: $D \in \langle S,A,R\rangle$

repeat until convergence:
- Randomly sample a batch of transitions, $B=\{(s,a,r,s')\}$ from D
- Compute target:
  - $y \leftarrow r+\gamma\cdot Q_\phi(s',\pi_\theta(s'))$
- Update critic function:
  - $\nabla_\phi \frac{1}{|B|}\sum_{(s,a,r,s')\in B}(Q_\phi(s,a)-y)^2$
- Update actor network:
  - $\nabla_\theta \frac{1}{|B|}\sum_{s\in B} Q_\phi(s,\pi_\theta(s))$

def `update_bitrate(`*state s*`):`
- $a \leftarrow \pi_\theta(s)$
- **return** a;

Further, we prepend the actor and critic networks with a learned embedding over the raw state vector to take advantage of the temporal aspect of the state. Following prior work, the state is a windowed series of metrics—we incorporate a Gated Recurrent Unit (GRU) to extract trends over the window and reduce the raw state to a more condensed vector [23]. Implementation details can be found in §4.4.

Conservative learning. While the rate control policies are now trained offline in the cloud, the challenge of a lack of feedback persists **(Challenge #1)**. Learning an improved rate control policy requires learning a strategy that deviates from the behavior seen in GCC. The greater the deviation, the greater the potential risk of extrapolation error; any errors in extrapolating will compound and drastically harm performance. In this section, we describe how the errors come to be in the context of the actor and critic networks, and then we go on to describe a risk mitigation strategy.

Recall that the learned critic function ultimately guides the decision-making policy (actor) to select actions that optimize for the best behavior. The critic Q_ϕ is responsible for learning a regression to estimate the value of (state, action) pairs observed in the dataset. It is also responsible for learning a regression to estimate the value of (state, action) pairs *not seen* in the dataset. The latter is much harder because it requires extrapolating—that is, using values of observed (state, action) pairs to estimate values for unseen (state, action) pairs. As a result, the regressor is likely to be more error-prone for those unseen regions of the state-action space. This is particularly problematic because the actor is trained to learn actions that maximize the critic function (Equation 3). Any mistakes or erroneous value assignments can lead the actor astray; in particular, the actor becomes biased toward selecting actions that have been erroneously assigned high values. The result is a poor-performing policy, one that can be potentially worse than the one featured in the data used to train it.

To address this, we opt for the following approach: when leveraging the output of the learned critic to teach the actor, trust the estimates more when the regressor is confident (i.e., the estimate is based on (state, action) pairs seen or are close to those observed in the dataset) and trust the estimates less when the regressor is less confident (i.e., the estimate is heavily extrapolated). Recall that in cases where GCC performs poorly, it eventually makes adjustments in the correct direction (§3.3)—these sequences provide a sufficient number of examples to confidently extrapolate about alternate trajectories in areas that GCC needs improvement.

In practice, we can achieve this by penalizing the estimated values for low-confidence regions and elevating those for high-confidence regions. Consequently, when the actor is leveraging the output of the critic, it learns to select actions that maximize the modified estimated value instead (i.e., one that takes into account the accuracy of the estimate) and ultimately, avoid falling into a trap of taking actions with erroneously high estimates due to errors in the learned regressor.

In particular, we leverage a state-of-the-art technique known as Conservative Q-Learning (CQL) [32]. CQL adds a regularizer to the critic's loss function (Equation 2):

$$\alpha\cdot\mathbb{E}_{s\sim D}\Big[\mathbb{E}_{a\sim\pi(a|s)}Q(s,a)-\mathbb{E}_{a\sim D}Q(s,a)\Big] \quad (4)$$

The regularizer first guides the critic to learn a lower bound on estimated values for all (state, action) pairs (i.e., a "conservative" estimate); simultaneously, it "pushes up" the values for (state, action) pairs observed in the training dataset. CQL provides an adjustable parameter α to identify the appropriate magnitude of the conservative penalty: too high of a penalty creates a conservative, low-risk policy that performs similarly to the behavior featured in the logs; too low of a penalty negates the benefits of this approach altogether, resulting in a risky (and potentially, low-performing) policy. We empirically find that setting $\alpha = 0.01$ achieves the best tradeoff; in our ablation studies in §5.5, we compare different settings of α.

Distributional representation. The second main challenge of learning from heuristic data collected in production deploy-

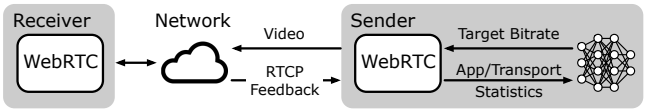


Figure 6: Mowgli deploys its model in the sender application's rate control logic.

ment settings is the presence of external phenomena outside the control of the rate control policy that ultimately complicates reasoning about the observed outcomes for a given (state, action) pair **(Challenge #2)**. To address this, we opt to explicitly account for the variance in expected return for a given (state, action) pair by modifying the way the learned value is represented. Traditionally, the critic learns a scalar value—i.e., the expected value of starting at state s and taking action a. We modify the critic function to learn a *distribution* over expected outcomes instead [18]. A probability distribution explicitly accounts for the possibility of multiple different futures, despite the bitrate agent taking the same action. Directly encoding a probability distribution into the model results in a more detailed representation of the environment's stochasticity and a more comprehensive way of capturing the effects of any given decision.

To implement this, we modify the output of the critic function to be a vector (representing a distribution) and update the loss function to support a vector representation. In line with prior work, we incorporate the Quantile Huber loss function, which compares distributions and penalizes estimates differently depending on which quantile they belong [24].

4.3 Policy Deployment

Based on insights from our generalization study (§5.3), Mowgli achieves performance wins across diverse network scenarios, provided that the corresponding state/action distributions are represented in the consumed telemetry logs. To adapt to new network environments, Mowgli continuously monitors these logs, and if a shift in the underlying state/action distribution is detected, the system triggers model retraining. Note that, unlike models trained in simulation (§2.3), Mowgli's model is amenable to further fine-tuning, as it is initially learned on the same distribution of environments and dynamics as the deployment setting. As a result, only minimal corrections are needed during online deployment (e.g., to account for environmental drift).

Deploying Mowgli's rate control algorithm requires modifying the application's rate control logic and sending over the weights parameterizing the learned model (Fig. 6). Inside the application, we spawn an additional Python process responsible for serving the model. The application code and Python process communicate via an interprocess pipe; the Python process consumes live telemetry data logged by the application instrumentation code and outputs an updated bitrate.

4.4 Implementation

Inspired by OpenNetLab [26], we implement Mowgli on top of WebRTC [8], an open-source framework for real-time video conferencing. Our current implementation, in line with prior work [28], targets unidirectional video without audio. Further, to isolate the effects of rate control in WebRTC, we set `DegradationPreference=DISABLED` [5]. We leverage the PyTorch [39] and d3rlpy [44] libraries to train and deploy Mowgli's learned rate control algorithm. We set the conservative loss penalty hyperparameter (α) to 0.01 and the number of quantiles (N) in our distributional value representation to 128. The actor and critic neural networks have 2 hidden layers of size 256; the GRU has a hidden unit size of 32.

5 Evaluation

We evaluated Mowgli on a diverse set of networks, both in emulation and in the wild. Our key findings are:

- Mowgli delivers substantial QoE improvements over GCC in emulated networks, increasing average video bitrates by 15–39% while reducing video freeze rates by 60–100%.
- Mowgli achieves similar performance to existing (impractical) data-driven online RL approaches, with average bitrates within 0.5–13.1% and freeze rates within 0–19%.
- Mowgli can achieve wins across diverse network scenarios if sufficiently represented in consumed telemetry logs.
- We deploy and evaluate Mowgli on real cellular networks across four U.S. cities. On target networks with high dynamism, Mowgli increases average video bitrates by 17.7% while maintaining similar levels of video freezes.

5.1 Methodology

Experiment setup and testbed. For lack of access to a production video conferencing deployment, we set up our own testbed to collect GCC logs and evaluate Mowgli. We extended the AlphaRTC [26] fork of WebRTC to run an end-to-end client-to-client video conferencing workflow. We run both clients on a single machine and use a network emulation tool (Mahimahi [37]) to emulate the network between clients.

To evaluate Mowgli on a diverse and challenging set of networks, we created a corpus of 87 hours of network bandwidth traces from two real-world datasets: FCC [3] broadband traces and Norway [42] cellular traces. We split each trace into 1-minute chunks. Following prior work [35], we filtered out traces with an average bandwidth of $<$0.2 Mbps or $>$6 Mbps. We used 60% of the traces for training, 20% for validation, and 20% for testing. Each network trace is randomly assigned to one of the following RTTs: 40, 100, and 160 ms. We use a queue length of 50 packets.

Similar to prior work [43], instead of recording and sending live video, we modify the WebRTC codebase to read from a prerecorded video. We use 9 different one-minute videos from a video conferencing dataset [38]; we randomly assign a video to each bandwidth trace. To create a corpus of "production logs," we collect logs from running GCC on the network traces in the training dataset. We note that although production logs from existing video conferencing platforms [2] exist, we

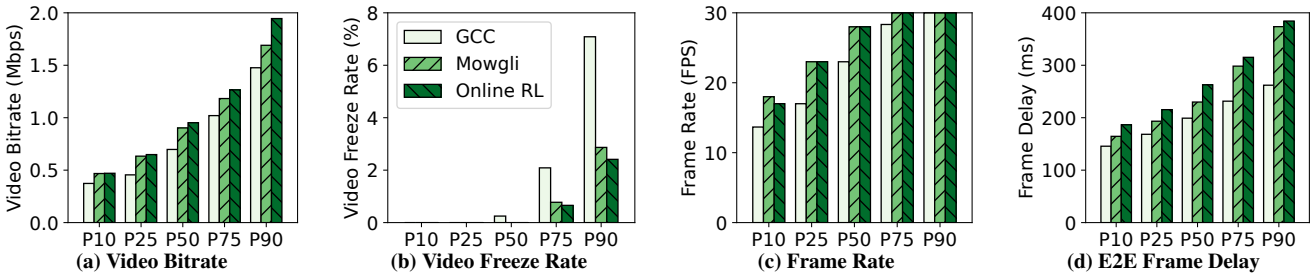


Figure 7: On emulated network traces, Mowgli consistently outperforms GCC and nearly matches the online RL baseline while avoiding QoE degradations during training.

do not have access to their systems to perform an evaluation; therefore, we opt to use data from our own setup.

QoE metrics. We evaluate QoE across the following metrics: (1) average received *video bitrate* (Mbps), (2) *video freeze rate*—fraction of session experiencing freezes (as defined by WebRTC [13]), (3) *frame rate* (FPS), and (4) average end-to-end *frame delay*. The first three metrics are already available in the WebRTC application logs. To calculate the end-to-end frame delay, we embed a QR code into each video frame to indicate the frame ID. We then calculate the timestamp difference between when the frame is read and when the corresponding frame is displayed. Note that we only measure the end-to-end frame delay for experiments done on emulated networks because on real networks, the clients are located on different devices and require nontrivial time synchronization.

Baseline algorithms. We compare Mowgli against the following baselines:

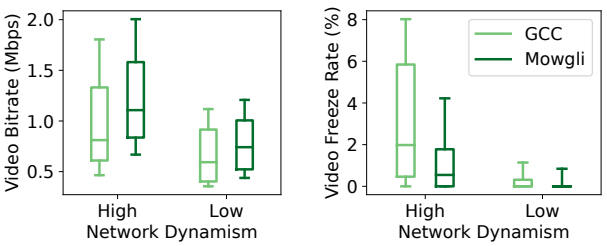
- **Google Congestion Control (GCC)** [21]: the de facto rate control algorithm for video conferencing. It employs hand-tuned rule-based heuristics to characterize network behavior based on recent packet delay and loss measurements. We use the built-in implementation of GCC in WebRTC.
- **Online RL**: an online reinforcement learning-based approach. We implement an in-house solution following the design and methodology of prior work [27, 54, 55]. The details of our implementation are provided in §A.1. Note that the reported results exclude quality degradations incurred during training (see §2.2); instead, we only present results from the model that performs the best on the test dataset.
- **Behavior Cloning (BC)** [40, 49]: an offline learning strategy that trains a rate control policy by imitating the behavior featured in existing logs through supervised learning.
- **Critic Regularized Regression (CRR)** [47]: another offline learning strategy that relies exclusively on existing logs for training. It is the underlying mechanism for Sage [52], a related work aimed at learning improved TCP congestion control algorithms from logs generated by dozens of different existing CC algorithms. Whereas CQL focuses on conservatively adjusting the critic function to avoid overestimating out-of-distribution actions, CRR regularizes the policy by using the critic's estimated values to guide the actor toward high-value actions of the dataset.

5.2 Overall Performance

Main results. Fig. 7 compares Mowgli with GCC and Online RL. There are two main takeaways. First, Mowgli consistently improves upon GCC; across reported percentiles, Mowgli increases the average bitrate by 14.5–39.2%, decreases the freeze rate by 59.5–100%, and increases the frame rate by 0–35.3%. End-to-end frame delays are within the 400 ms interactivity threshold [11], even with RTTs up to 160 ms. Second, despite learning completely offline, Mowgli's performance nearly matches that of the online RL baseline. Across percentiles, Mowgli achieves an average bitrate within 0.5–13.1% of Online RL's average. Mowgli's P75 and P90 video freeze rates are 0.77% and 2.87%, respectively, only slightly higher than Online RL's rates of 0.66% and 2.41%. For comparison, GCC's P75 and P90 freeze rates are 2.09% and 7.09%.

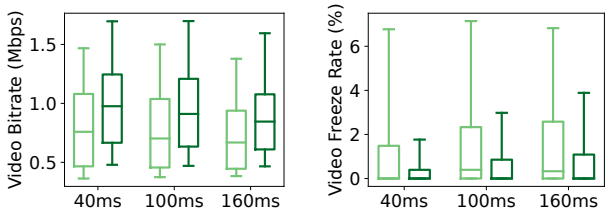
Breakdown by dynamism. To better understand how Mowgli performs in the network conditions where GCC performs poorly, we split the dataset of network traces based on the degree of network dynamism. Specifically, we calculate the standard deviation of 1-second network bandwidth chunks within each trace and split the dataset along the mean standard deviation across traces. We observe that Mowgli achieves its largest wins over GCC in traces with high bandwidth dynamism – across reported percentiles, Mowgli increases the average video bitrate by 10.8–43.8% and decreases the video freeze rate by 47.4–100% (Fig. 8). In contrast, for traces with less dynamism, Mowgli increases the average video bitrate by 8.0–29.6% and decreases the video freeze rate by 26.2–100%.

Breakdown by network dataset & delay. In Fig. 9, we break down the results from Fig. 7 based on different characterizations of the underlying network traces. We find that as network delay increases, Mowgli's P50 video bitrate decreases (976 kbps → 911 kbps → 845 kbps) and its P75 video freeze rates increase (0.39% → 0.86% → 1.09%). We report P75 video freeze rates because P50 video freeze rates are 0%. This is expected as higher network delays increase the time it takes for a model to receive feedback about potential network congestion and react appropriately; as a result, the model is less aggressive in its bandwidth ramp-ups. When split by

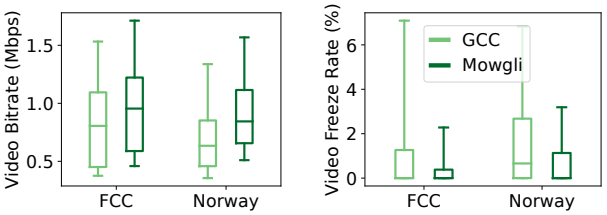


(a) Video Bitrate (b) Freeze Rate

Figure 8: Evaluating Mowgli's performance in network traces of varying network dynamism, measured by the degree of bandwidth variation. Whiskers show P10–P90. Mowgli's win relative to GCC is higher under high network dynamism.



(a) Video Bitrate (Diff. RTT) (b) Freeze Rate (Diff. RTT)



(c) Video Bitrate (Diff. Dataset) (d) Freeze Rate (Diff. Dataset)

Figure 9: Evaluating Mowgli's performance in network traces of varying delay (RTT) and dataset. Whiskers show P10–P90.

dataset, Mowgli performs better in the FCC dataset than in the Norway dataset (954 kbps vs. 844 kbps P50 video bitrates, 0.39% vs. 1.13% P75 video freeze rates). The Norway dataset was collected over 3G cellular networks and therefore, features more network dynamism than the traces in the FCC dataset (which were captured on wired broadband networks).

Additional baselines. In Fig. 10, we compare Mowgli with two alternative learning strategies: Behavior Cloning (BC) and Critic Regularized Regression (CRR). We find that BC behaves less aggressively than Mowgli, achieving a P90 video bitrate that is 14.4% lower than GCC, whereas Mowgli increases the bitrate by 14.5% compared with GCC. This difference arises because BC only aims to imitate the behavior observed in the training logs and fails to effectively extrapolate to unseen scenarios. CRR, the underlying learning algorithm in Sage [52], performs worse than GCC on both metrics, with a 4.4% increase in the P90 video freeze rate and an 8.8% decrease in the P90 video bitrate. We hypothesize that this is due to the lack of state-action coverage featured in the logs of Google Congestion Control; the logs of Sage, on the other hand, contain the experience of dozens of different CC algorithms reducing the likelihood of erroneous estimates.

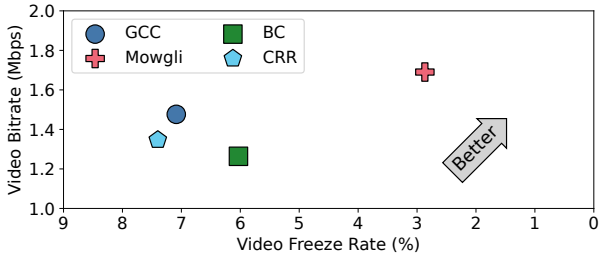
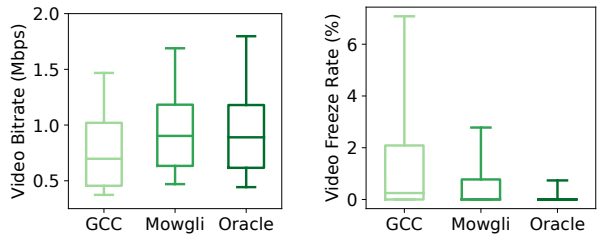


Figure 10: Mowgli outperforms additional baselines in bitrate and freezes (P90 performance is shown).



(a) Video Bitrate (b) Freeze Rate

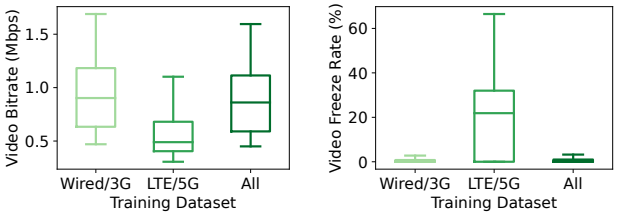
Figure 11: Comparing Mowgli against an approximate oracle algorithm that represents an upper bound on performance improvements over GCC; whiskers correspond to P10 and P90.

Comparison to approximate oracle. In §3.3, we implemented an approximate oracle algorithm to estimate the maximum possible improvement over GCC. This algorithm serves as an estimate of the (unattainable) upper bound on performance improvements. Fig. 11 compares this oracle with Mowgli. Across reported percentiles, Mowgli comes within 6% of the oracle's achieved video bitrate. Compared with GCC's video freeze rates of 2.1% at P75 and 7.1% at P90, Mowgli significantly reduces these rates to 0.8% and 2.9%, while the oracle further lowers them to 0% and 0.7%. These additional reductions are expected since the oracle algorithm has access to ground-truth network bandwidths in advance.

5.3 Generalization & Deployment Considerations

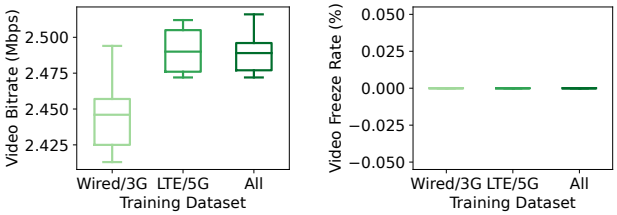
In this section, we examine the limits of Mowgli's ability to generalize to network conditions and types not represented in the production telemetry dataset. Additionally, we seek to quantify the benefits of model specialization for specific network conditions. To this end, we incorporate an additional set of network traces captured on LTE and 5G networks [29]. Following our established methodology, we collected GCC logs on these LTE/5G traces and trained a corresponding policy. We then evaluated this policy on network traces from the primary dataset, which consists of wired and 3G traces. The results are summarized in Fig. 12, with two key observations.

First, we find that Mowgli's generated policies perform poorly when evaluated on network conditions that significantly differ from those featured in the training telemetry logs. For instance, the LTE/5G-trained policy performs notably worse than the Wired/3G-trained policy when evaluated on Wired/3G network traces. This results in a 45.8% reduction in the P50 video bitrate and a 40.3× increase in the P75



(a) Video Bitrate (b) Freeze Rate

Figure 12: Evaluating Mowgli's performance on the Wired/3G network dataset when varying the network telemetry dataset consumed; whiskers show P10–P90.



(a) Video Bitrate (b) Freeze Rate

Figure 13: Evaluating Mowgli's performance on the LTE/5G network dataset when varying the content of the network telemetry dataset consumed; whiskers show P10–P90.

video freeze rate. Importantly, this is expected due to underlying state/action distribution differences in the corresponding GCC logs (e.g., GCC's average video bitrate is 1.6 Mbps higher in the LTE/5G dataset). We further discuss this in §7.

Second, we observe that specializing a model for a specific set of network traces is not strictly necessary. Instead, general models can achieve strong performance across network conditions, provided they are well-represented in the training data. For example, a model trained on both network datasets performs comparably to one trained solely on the Wired/3G dataset. While the Wired/3G-specific model achieves slightly better results (a 4.6% higher P50 video bitrate and a reduction in the P75 video freeze rate from 1.00% to 0.77%), the general model still performs effectively across conditions.

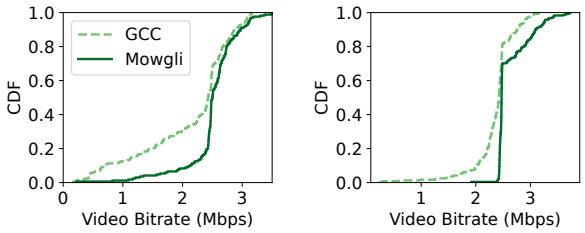
We conducted a similar analysis using the LTE/5G dataset, evaluating the same generalized model on these network traces (Fig. 13). The observed trends persist, with the Wired/3G-trained policy, for example, reducing the median video bitrate by 1.8%.

5.4 Real World Experiments

We evaluated Mowgli and GCC on real cellular networks in 4 different cities (Table 2). In these experiments, we set up conferencing sessions with a client running on a Macbook Pro (tethered to a Google Pixel) and a server in the cloud. Experiments were performed in several mobility scenarios (e.g., train, bus, car, walking, and stationary). To generate a dataset of GCC logs, we collected over 8 hours of video calls using GCC in Princeton, NJ and San Jose, CA on a 4G/LTE network. Once Mowgli generated a policy, we evaluated the resulting model in 2 different scenarios: (a) the same networks and cities and (b) different cities. During our evaluation, we

Scenario	Network	Cities
A	4G/LTE	Princeton, NJ; San Jose, CA
B (new cities)	4G/LTE	New York City, NY; Nashville, TN

Table 2: Cities and network types for in-the-wild evaluation.



(a) Scenario A (b) Scenario B

Figure 14: Evaluating Mowgli on real-world cellular networks; scenarios are described in Table 2. Mowgli outperforms GCC across bitrate percentiles, while freezes remain statistically indistinguishable between the two (not shown).

alternated running GCC and the generated policy, collecting over 4 hours of data for each scenario.

We observe that Mowgli's wins over GCC extend to cellular networks in both scenarios, increasing video bitrates across reported percentiles by 3.0%–2.1× (Fig. 14a) and 2.0%–20.8% Fig. 14b). Video freezes—as a rare event—are inherently challenging to measure reliably [51]. While we currently lack sufficient evidence to draw definitive conclusions, the observed rates across 120 runs per policy/scenario appear statistically indistinguishable (results not shown).

5.5 Ablation Studies and Microbenchmarks

Algorithm design ablation. Fig. 15a compares Mowgli with two variants: (1) Mowgli without the conservative learning regularizer, and (2) Mowgli without the distributional representation. Removing the regularizer, which adjusts the learned estimates of the critic function based on sample confidence, makes the model prone to actions with erroneously high estimated values and results in 11.3× higher P90 video freeze rate. Without the distributional representation, which accounts for external phenomena outside the control of the bitrate decision-making policy, P90 video bitrates drop by 5.6% and P90 video freeze rates increase by 9.9×.

Varying state design. Fig. 15b shows the benefits of the additional state features. Removing "Report Intervals", which indicates the staleness of the sender-side transport feedback report, leads to an 8.7% lower video bitrate. Removing "Min RTT", which indicates how fast the client can receive (and react to) feedback and helps control the model's aggression, leads to a 1.2× higher freeze rate. Removing "Prev Action", which enables smooth rate control, results in a 3.1× increase in the video freeze rate.

CQL α parameter sensitivity. α dictates the relative weight of the conservative penalty in the learning algorithm. In Fig. 15c, a larger α creates a conservative, low-risk policy (∼57% lower freeze rates), but lowers video bitrates by 34.9%

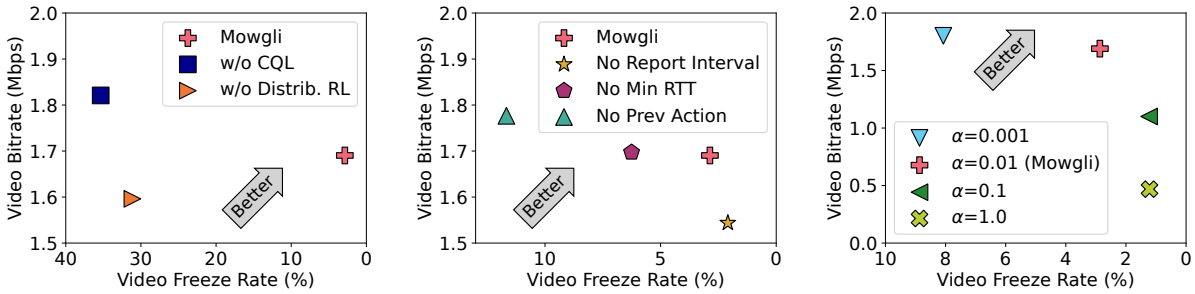


Figure 15: Ablation results varying the algorithm design, state design, and alpha. Markers correspond to P90. Mowgli's design achieves the best bitrate-freeze tradeoff.

and 72.3%, respectively. In contrast, $\alpha < 0.01$ increases the amount of deviation (and therefore risks: 1.8× higher video freeze rate) albeit increasing bitrate by 6.6%.

System overheads. We study the compute and storage overhead of deploying Mowgli. The compressed (state, action, reward) logs take ∼117 kB for a 1-minute call. Mowgli's generated policy (weights) is 316 kB (corresponding to 79k parameters) and takes ∼6 ms to run on the CPU.

6 Related Work

Alternative designs and knobs for video conferencing. Recent work has explored different optimization dimensions in videoconferencing. Salsify [28] co-designs the video codec and transport protocol to quickly respond to changing network conditions and achieve low latency. Gemino [45] designs a neural codec for high perceptual quality in low bandwidth environments. Grace [22] designs a loss-resilient neural codec. AFR [36] adapts the frame rate for video conferencing applications with ultra-high-definition demand. Mowgli is complementary to these works and can be extended to incorporate these alternate dimensions, which we leave for future work.

ML for networked systems. Machine learning has been applied to various other challenges in networked systems. For example, to improve adaptive bitrate (ABR) selection in video streaming, Pensieve [35] trains a reinforcement learning agent in simulation, whereas Puffer [51] deploys a neural network-based model predictive control scheme under real-world conditions. Unlike video streaming, which deals with discrete bitrate adaptation, video conferencing requires continuous rate control, tighter latency constraints, and frequent (every 50 ms) decision-making for on-the-fly encoding and compression. Beyond ABR, ML has been explored for TCP congestion control; Orca [14] employs an online RL agent, while Sage [52] trains offline but requires additional data collection with multiple expert policies. In contrast, Mowgli leverages existing logs of a single policy (e.g., GCC).

7 Discussion

Pitfalls of leveraging production traces. Conceptually, the challenges described in §3.4 align with those in recent research on counterfactual reasoning in trace-driven simulations for adaptive bitrate selection [16,17,19,46], e.g., the impact of latent confounders (factors that impact system dynamics but are not explicitly featured in logs). Veritas [19] addresses this by employing advanced statistical methods to explicitly infer these variables. While these efforts aim to refine simulations for more reliable counterfactual reasoning, Mowgli focuses on generating an enhanced rate control policy derived from the behavior of an existing heuristic algorithm. Rather than depending on accurate counterfactual reasoning universally, Mowgli limits it to scenarios where sufficient data is available for reliable analysis (i.e., a conservative approach—§4.2). In cases lacking enough log data, Mowgli learns to mimic the baseline GCC behavior. Incorporating insights from these works into Mowgli and other video conferencing systems is a promising avenue for future exploration, such as improving the critic by explicitly inferring latent confounders.

Limitations of a data-driven approach. A limitation of any data-driven approach is that if the underlying data distribution shifts substantially from the one captured in training, the model must be updated [34]. In the case of Mowgli, encountering previously unseen network conditions (e.g., new bandwidth ranges) would necessitate a model update (§5.3). This limitation reflects the inherent trade-off of data-driven methods: Mowgli is constrained to perform well in environments it has been trained on, unlike GCC, which operates under a broader set of pre-designed assumptions. However, unlike models trained in simulation or from scratch, Mowgli's model—trained on real user interactions and dynamics—starts on a much closer distribution to its deployment setting. This makes it more amenable to fine-tuning, requiring only minimal adjustments to account for environmental drift.

8 Conclusion

In this paper, we presented Mowgli, a system for *practically* learning improved bitrate control algorithms for video conferencing. Whereas existing data-driven reinforcement learning-based approaches fall short of meeting the practicality constraints of production settings, Mowgli demonstrates the possibility of learning from data already collected in existing system telemetry logs (and avoiding the QoE disruptions of prior work) to generate improved bitrate control algorithms.

Acknowledgements. We thank our shepherd, Zahaib Akhtar, and the anonymous NSDI reviewers for their valuable feedback and comments. This work was supported by NSF CNS grants 2147909, 2151630, 2140552, 2153449, and 2152313.

REFERENCES

[1] Accessing Zoom Meeting and Phone Statistics. https://support.zoom.com/hc/en/article?id=zm_kb&sysparm_article=KB0070504.

[2] Bandwidth Estimation Challenge Dataset. https://www.microsoft.com/en-us/research/academic-program/bandwidth-estimation-challenge/data.

[3] Measuring Broadband America. https://www.fcc.gov/general/measuring-broadband-america.

[4] Microsoft Teams Client Diagnostic Logs. https://learn.microsoft.com/en-us/microsoftteams/log-files.

[5] RTP Parameters. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/api/rtp_parameters.h.

[6] Understanding the Consent Settings in Data & Privacy Center. https://support.zoom.com/hc/en/article?id=zm_kb&sysparm_article=KB0057779.

[7] Use real-time telemetry to troubleshoot poor meeting quality. https://learn.microsoft.com/en-us/microsoftteams/use-real-time-telemetry-to-troubleshoot-poor-meeting-quality.

[8] WebRTC. https://webrtc.org/.

[9] WebRTC Logging. https://webrtc.github.io/webrtc-org/native-code/logging/.

[10] Zoom Service Quality Logging. https://developers.zoom.us/docs/video-sdk/web/quality/.

[11] G.114 One-way transmission time. Technical report, International Telecommunication Union, May 2003.

[12] Closing the Simulation-to-Reality Gap for Deep Robotic Learning, 2017. https://research.google/blog/closing-the-simulation-to-reality-gap-for-deep-robotic-learning/.

[13] Identifiers for WebRTC's Statistics API, July 2018. https://www.w3.org/TR/webrtc-stats/.

[14] S. Abbasloo, C.-Y. Yen, and H. J. Chao. Classic Meets Modern: a Pragmatic Learning-Based Congestion Control for the Internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 632–647, 2020.

[15] Z. Akhtar, Y. S. Nam, R. Govindan, S. Rao, J. Chen, E. Katz-Bassett, B. Ribeiro, J. Zhan, and H. Zhang. Oboe: Auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58, 2018.

[16] A. Alomar, P. Hamadanian, A. Nasr-Esfahany, A. Agarwal, M. Alizadeh, and D. Shah. CausalSim: A Causal Framework for Unbiased Trace-Driven Simulation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1115–1147, 2023.

[17] M. Bartulovic, J. Jiang, S. Balakrishnan, V. Sekar, and B. Sinopoli. Biases in Data-Driven Networking, and What to Do About Them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 192–198, 2017.

[18] M. G. Bellemare, W. Dabney, and M. Rowland. *Distributional Reinforcement Learning*. MIT Press, 2023.

[19] C. Bothra, J. Gao, S. Rao, and B. Ribeiro. Veritas: Answering causal queries from video streaming traces. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 738–753, 2023.

[20] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, and V. Vanhoucke. Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, page 4243–4250. IEEE Press, 2018.

[21] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo. Analysis and Design of the Google Congestion Control for Web Real-time Communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, 2016.

[22] Y. Cheng, Z. Zhang, H. Li, A. Arapin, Y. Zhang, Q. Zhang, Y. Liu, K. Du, X. Zhang, F. Y. Yan, et al. Grace: Loss-resilient real-time video through neural codecs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*, pages 509–531, 2024.

[23] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*, abs/1406.1078, 2014.

[24] W. Dabney, M. Rowland, M. Bellemare, and R. Munos. Distributional Reinforcement Learning with Quantile Regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[25] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, 2019.

[26] J. Eo, Z. Niu, W. Cheng, F. Y. Yan, R. Gao, J. Kardhashi, S. Inglis, M. Revow, B.-G. Chun, P. Cheng, et al. OpenNetLab: Open platform for RL-based Congestion Control for Real-Time Communications. *Proc. of APNet*, 2022.

[27] J. Fang, M. Ellis, B. Li, S. Liu, Y. Hosseinkashi, M. Revow, A. Sadovnikov, Z. Liu, P. Cheng, S. Ashok, et al. Reinforcement Learning for Bandwidth Estimation and Congestion Control in Real-Time Communications. *arXiv preprint arXiv:1912.02222*, 2019.

[28] S. Fouladi, J. Emmons, E. Orbay, C. Wu, R. S. Wahby,

and K. Winstein. Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, pages 267–282, 2018.

[29] M. Ghoshal, Z. J. Kong, Q. Xu, Z. Lu, S. Aggarwal, I. Khan, Y. Li, Y. C. Hu, and D. Koutsonikolas. An In-Depth Study of Uplink Performance of 5G mmWave Networks. In *Proceedings of the ACM SIGCOMM Workshop on 5G and Beyond Network Measurements, Modeling, and Use Cases*, pages 29–35, 2022.

[30] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[31] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne. Imitation Learning: A Survey of Learning Methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[32] A. Kumar, A. Zhou, G. Tucker, and S. Levine. Conservative Q-Learning for Offline Reinforcement Learning. *Advances in Neural Information Processing Systems*, 33:1179–1191, 2020.

[33] S. Levine, A. Kumar, G. Tucker, and J. Fu. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv preprint arXiv:2005.01643*, 2020.

[34] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang. Learning under Concept Drift: A Review. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[35] H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[36] Z. Meng, T. Wang, Y. Shen, B. Wang, M. Xu, R. Han, H. Liu, V. Arun, H. Hu, and X. Wei. Enabling High Quality Real-Time Communications with Adaptive Frame-Rate. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, pages 1429–1450, 2023.

[37] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pages 417–429, 2015.

[38] E. Nowroozi, A. Dehghantanha, R. M. Parizi, and K.-K. R. Choo. A Survey of Machine Learning Techniques in Adversarial Image Forensics. *Computers & Security*, page 102092, 2020.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[40] D. A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

[41] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *The Journal of Machine Learning Research*, 22(1):12348–12355, 2021.

[42] H. Riiser, P. Vigmostad, C. Griwodz, and P. Halvorsen. Commute path bandwidth traces from 3G networks: analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 114–118, 2013.

[43] M. Rudow, F. Y. Yan, A. Kumar, G. Ananthanarayanan, M. Ellis, and K. Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, pages 953–971, 2023.

[44] T. Seno and M. Imai. d3rlpy: An offline deep reinforcement learning library. *The Journal of Machine Learning Research*, 23(1):14205–14224, 2022.

[45] V. Sivaraman, P. Karimi, V. Venkatapathy, M. Khani, S. Fouladi, M. Alizadeh, F. Durand, and V. Sze. Gemino: Practical and Robust Neural Compression for Video Conferencing. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*, pages 569–590, 2024.

[46] P. C. Sruthi, S. Rao, and B. Ribeiro. Pitfalls of data-driven networking: A case study of latent causal confounders in video streaming. In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI '20, page 42–47, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Z. Wang, A. Novikov, K. Zolna, J. S. Merel, J. T. Springenberg, S. E. Reed, B. Shahriari, N. Siegel, C. Gulcehre, N. Heess, and N. de Freitas. Critic Regularized Regression. In *Advances in Neural Information Processing Systems*, volume 33, pages 7768–7778, 2020.

[48] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[49] B. Widrow. Pattern recognition and adaptive control. *IEEE Transactions on Applications and Industry*, 83(74):269–277, 1964.

[50] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 459–471, 2013.

[51] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein. Learning in Situ: a Randomized Experiment in Video Streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 495–511, 2020.

[52] C.-Y. Yen, S. Abbasloo, and H. J. Chao. Computers Can Learn from the Heuristic Designs and Master In-

ternet Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 255–274, 2023.

[53] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 509–522, 2015.

[54] H. Zhang, A. Zhou, Y. Hu, C. Li, G. Wang, X. Zhang, H. Ma, L. Wu, A. Chen, and C. Wu. Loki: Improving Long Tail Performance of Learning-Based Real-Time Video Adaptation by Fusing Rule-Based Models. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 775–788, 2021.

[55] H. Zhang, A. Zhou, J. Lu, R. Ma, Y. Hu, C. Li, X. Zhang, H. Ma, and X. Chen. OnRL: Improving Mobile Video Telephony Via Online Reinforcement Learning. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[56] A. Zhou, H. Zhang, G. Su, L. Wu, R. Ma, Z. Meng, X. Zhang, X. Xie, H. Ma, and X. Chen. Learning to Coordinate Video Codec with Transport Protocol for Mobile Video Telephony. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

Hyperparameter	Value
Learning Rate	5e-5
Batch Size	512
Gradient Steps	500
Replay Buffer Size	1e6
Init. Entropy Coefficient	0.5
GRU Hidden Size	32
Num Parallel Workers	30
Optimizer	Adam

Table 3: Online RL Hyperparameter Values. For all other unspecified hyperparameters, we use the default values of the implementation in Stable Baselines3 [41]. The same learning rate is used for all networks (Q-Values, Actor, and Value function).

A Appendix

A.1 Online RL Setup

Experiment Setup and Testbed. Similar to prior work [14, 54], our online RL baseline relies on a cloud server to support the training. The centralized server has an AMD EPYC 7543P 32-core CPU, 256 GB of RAM, and an NVIDIA RTX A6000-48G GPU. We use 30 CloudLab [25] nodes ("workers") to emulate end users in our experiments, each with an Intel Xeon D-1548 8-core CPU and 64 GB of RAM. We modify Stable Baselines3 v2.1.0 [41] to support ingesting state-action-reward tuples from parsed traces and use PyTorch v2.0.1 to train and serve the neural network. Similar to prior work, we use a state-of-the-art off-policy algorithm [30]. Following OnRL [55], we implement a fallback mechanism that allows the sender to temporarily downgrade to the default heuristic (GCC) if catastrophic QoE degradations are detected during training.

In every round of online RL training, the RL server dispatches the latest model to all 30 worker nodes. Similar to how we collect GCC logs in our main experiments, we run both the sender and receiver client on the same node. Each worker uses Mahimahi [37] to replay a random one-minute network bandwidth trace from the same diverse dataset Mowgli uses. At the end of every video conferencing session, each node parses the sender/receiver logs to construct a state-action-reward trace and sends it back to the RL server. The RL server performs one training epoch based on the aggregated state-action-reward tuples across all workers and repeats this process. We enumerate the hyperparameters used by our online RL baseline in Table 3.

We retain the state and action formulation as defined in the main text. However, we observe that using the following reward definition for training the online RL policy further improves performance:

$$\begin{aligned} R = {} & throughput \cdot delay \cdot (1 - \gamma \cdot loss) \\ & - \zeta \cdot \max(prev_action - sending_bitrate, 0) \\ & - use_gcc \cdot gcc_penalty \end{aligned} \tag{5}$$

where $\gamma = 2$, and $\zeta = 3$. use_gcc acts as an indicator variable for whether the fallback mechanism was invoked; we set $gcc_penalty = 0.05$. We normalize $throughput$, $delay$, $loss$, $prev_action$, and $sending_bitrate$ to (0, 4.5 Mbps), (0, 1000 ms), (0, 1), (0, 4.5 Mbps), and (0, 4.5 Mbps).

Dissecting and Streamlining the Interactive Loop of Mobile Cloud Gaming

Yang Li[1*], Jiaxing Qiu[1*], Hongyi Wang[1], Zhenhua Li[1], Feng Qian[2], Jing Yang[1]
Hao Lin[1,4], Yunhao Liu[1], Bo Xiao[3], Xiaokang Qin[3], Tianyin Xu[4]
[1]*Tsinghua University* [2]*University of Southern California* [3]*Ant Group* [4]*UIUC*

Abstract

With cloud-side computing and rendering, mobile cloud gaming (MCG) is expected to deliver high-quality gaming experiences to budget mobile devices. However, our measurement on representative MCG platforms reveals that even under good network conditions, all platforms exhibit high interactive latency of 112–403 *ms*, from a user-input action to its display response, that critically affects users' quality of experience. Moreover, jitters in network latency often lead to significant fluctuations in interactive latency.

In this work, we collaborate with a commercial MCG platform to conduct the first in-depth analysis on the interactive latency of cloud gaming. We identify VSync, the synchronization primitive of Android graphics pipeline, to be a key contributor to the excessive interactive latency; as many as five VSync events are intricately invoked, which serialize the complex graphics processing logic on both the client and cloud sides. To address this, we design an end-to-end VSync regulator, dubbed LoopTailor, which minimizes VSync events by decoupling game rendering from the lengthy cloud-side graphics pipeline and coordinating cloud game rendering directly with the client. We implement LoopTailor on the collaborated platform and commodity Android devices, reducing the interactive latency (by ∼34%) to stably below 100 *ms*.

1 Introduction

Recent years have witnessed the emergence and prosperity of cloud gaming for mobile games (termed *mobile cloud gaming*, or MCG), which enables high-quality gaming experiences on mid- to low-end devices. Typically, mobile games run in virtualized environments (e.g., an emulator/a container) hosted on cloud servers, and rendered frames are live-streamed over the network to a thin client app on the user device.

Motivation. To achieve smooth gaming experiences, MCG platforms are expected to provide users with a high media streaming bandwidth (e.g., 10+ Mbps) [23] and a low *interactive latency* (i.e., the delay between a user-input action and when the game's response to that action manifests at the client) within ∼100 *ms* [54]. In practice, the former has been largely fulfilled by the deployment of geo-distributed cloud/edge servers [24]. Nevertheless, the latter, as a key metric that directly determines the QoE of MCG, seems far from satisfactory [3, 54]. MCG's interactive latency has not been well understood—it is more than network latency due to the complexity of cloud-side processing and virtualization [72].

We conduct the first in-depth comparative measurement study of MCG and CCG (cloud gaming for console games, or *console cloud gaming*) on interactive latency characteristics in practice. Our measurement is based on eight representative cloud gaming platforms: GeForce Now [61], Boosteroid [20], JoyArk [41], CloudMoon [26], NetEase Cloud Gaming [58], Tencent Pioneer [78], Tencent Start [79], and X-MCG (anonymized due to their commercial concerns), regarding both mobile and console games under diverse scenarios. The study reveals important and surprising results:

- Our measured minimum interactive latency of MCG is as high as 112 *ms*, 42% larger than that of CCG. This implies that the processing logic of MCG should be more complex than that of CCG and the interactive latency of today's MCG cannot meet the users' requirements (<100 *ms*).
- The interactive latency has little to do with the user request workload but much to do with the game rendering workload. Specifically, the interactive latency looks similar during peak and non-peak hours (indicating sufficient scalability), but surprisingly (up to 48%) higher for simple 2D games than for complex 3D games (mostly relating to the unbalanced resource allocation strategy of a platform).
- On average, network latency only accounts for 17% of the interactive latency, indicating that network is not the key bottleneck. In fact, a jitter (≤2 *ms*) in network latency often has a "butterfly effect" on the interactive latency (≥10 *ms*).

The above drive us to go beyond RTT and understand the *interactive loop* of MCG, including every step in the back-and-forth MCG processing pipeline. However, the interactive loop of commercial MCG platforms is rarely studied in depth as it involves not only network links but also complex backend processings in the cloud, which are typically invisible.

Analysis. To deeply understand the interactive loop of MCG, we contacted developers of each platform. One platform X-MCG (anonymized due to their commercial concerns) shared their design and agreed to collaborate with us on improving interactive latency. Its cloud side relies on a state-of-the-art Android emulator *Trinity* [33] (optimized for high-performance graphics rendering) and a widely-used streaming tool *Sunshine* [51]. Its client side is extended from *Moonlight* (the client of Sunshine), supporting hardware-accelerated video decoding and High Dynamic Range (HDR) streaming. We

[*] Co-primary authors. Zhenhua Li is the corresponding author.

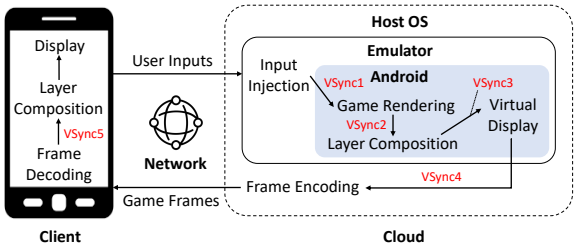


Figure 1: The interactive loop of X-MCG.

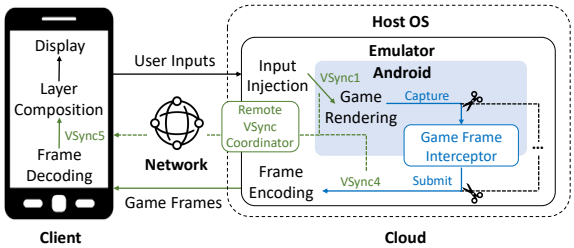


Figure 2: The interactive loop of X-MCG using LoopTailor.

take this unique opportunity to first expose to the public a state-of-the-art MCG system's interactive loop, and uncover the root cause of the unsatisfactory interactive latency.

To precisely trace the data flow of a game frame in X-MCG, we instrument relevant system services of Android, the GPU module of *Trinity*, and core modules of *Sunshine/Moonlight*. We find that the key bottleneck of the interactive latency is the cloud-side graphics pipeline. More in depth, the major latency of the cloud-side graphics pipeline does not lie in game rendering (23%), but the Vertical Synchronization (VSync) operations [2, 18, 53] (36%) in mobile OSes. As a synchronization primitive, VSync synchronizes frame production and consumption rates (e.g., the output frame rate of the GPU and the refresh rate of the display) among different stages in a graphics pipeline to avoid *screen tearing* [53] (i.e., a visual artefact that occurs when a display device shows pixels from multiple frames in a single screen draw). Given that graphics pipelines of mobile systems are designed to handle diverse workloads with power-constrained hardware and fixed-refresh-rate display [18], VSync is built into the Android graphics pipeline as a system-level always-on mechanism.

Given the multi-component and multi-stage nature of the MCG graphics pipeline, severe performance penalty is incurred as each VSync event may cause an uncertain latency. As shown in Figure 1, a game frame encounters as many as five VSync events, creating up to 83 *ms* extra latency. Given the periodicity of VSync events, even a slight network jitter (e.g., 1 *ms*) can lead to a frame deadline miss, resulting in a significant fluctuation (e.g., 13 *ms*) in interactive latency. Note that the tight coupling of VSync with Android makes it not limited to specific emulators or streaming tools, but affects other representative MCG solutions (e.g., DroidCloud [47]) as well, incurring even higher (20%+) latencies (§6).

We argue that the VSync overhead in the MCG graphics pipeline can be effectively optimized away with specialized system design. Basically, current uses of VSync are built for general-purpose mobile usage, where multiple apps and system services may render simultaneously and thus require sophisticated synchronizations to achieve smooth and correct rendering. However, in MCG, the game app process monopolizes the rendering pipeline, and thus game-irrelevant synchronizations within the cloud (i.e., VSync2 and VSync3) could be avoidable, given that layer composition and virtual display only append system UIs (e.g., a status bar) to a game frame. Essentially, only coordination with the client is a necessity, leaving considerable space for optimization inside the cloud.

Optimization. We present LoopTailor, an end-to-end VSync regulator to decouple game rendering from the lengthy cloud pipeline (thus reducing four cloud VSync events to two), and coordinate its pace directly with the client (VSync5). To do so, LoopTailor carefully streamlines MCG's interactive loop. As shown in Figure 2, LoopTailor consists of two modules: *Game Frame Interceptor* which extracts and reroutes game frames to bypass unnecessary VSyncs (VSync2 and VSync3), and *Remote VSync Coordinator* which coordinates ahead of time the remaining VSyncs (VSync1, VSync4, and VSync5) on both sides. LoopTailor's approach is transparent to specific VSync implementations, making it a generic approach requiring minimum modifications to the host/guest OS.

Nevertheless, realizing LoopTailor is challenging. First, extracting game frames halfway from the pipeline can introduce non-trivial guest-host frame copy overheads. Second, VSync coordination between the cloud and the client requires accurately predicting latencies of intervening stages, as a prediction error may cause a frame deadline miss at the client.

For intercepting game frames, conventional approaches like app-level render redirection [15, 87] and framework-level buffer hooking [83] induce substantial overheads of frame copies in the virtual mobile graphics stack. Our solution instead breaks the boundary of the guest system and the host virtual GPU to efficiently identify render instructions issued by the game and capture the game frames *in place*, thus minimizing frame copies between the CPU and the GPU. The Interceptor then immediately forwards the game frames to the encoder, so that VSync2 and VSync3 are bypassed.

For multi-stage latency prediction, we discover that the latencies (incurred by game rendering, frame encoding, network transmission, and frame decoding), despite heterogeneous and fluctuating, can be forecast hierarchically with a low standard deviation ($<$1.3 *ms*) based on their latent correlations. Thus, we devise a Synergetic VSync Alignment approach which proactively adjusts the execution time of rendering/encoding operations across the cloud-side pipeline, so as to minimize the client-side frame deadline miss rate (0.3%).

We implement LoopTailor atop X-MCG with 15K lines of C/C++/Python code and evaluate its real-world performance with 100 mobile devices. Our prototype is applicable to both single- and multi-player mobile games. The results

demonstrate that LoopTailor can achieve desirable interactive latencies of 82-96 *ms* (with X-MCG's geo-distributed edge servers), 34% lower than that of X-MCG's original solution. The average ratio of interactive latency fluctuation to network jitter is greatly reduced from 11.52 to merely 1.23.

Contribution. We make the following contributions:

- We make a month-long measurement study on 8 representative cloud gaming platforms, revealing important results on interactive latency and optimization opportunities.
- We conduct the first in-depth study of MCG's interactive loop. Our analysis identifies the VSync mechanism in the cloud-side graphics pipeline of MCG as a key contributor to the excessive interactive latency, which can be effectively optimized with specialized design for MCG.
- We design and implement LoopTailor to streamline X-MCG's interactive loop, achieving sub-100 *ms* interactive latency in real-world deployments. The code and data are released at https://MCGlatency.github.io.

2 Measurement Study in the Wild

In this section, we introduce our methodology of measuring the interactive latency of commercial cloud gaming platforms (§2.1), and present multifold data analysis results (§2.2).

2.1 Measurement Methodology

Due to commercial interests and security concerns, almost all the commercial cloud gaming platforms are closed-source systems and provide very limited technical documentation to the public. Therefore, we measure the interactive latency of a platform from the outside in a record-and-recognize approach.

First, we use high-speed cameras to record videos of user interactions with games with an ultra-high frame rate of 2000 FPS (Frame Per Second) and Full-HD resolution (1920×1080). We position 2 cameras in front of and at the side of the device screen respectively, so that both user actions and visual game contents can be clearly observed. For recorded videos, we examine each frame in a hybrid (manual and algorithmic) approach to recognize the timestamp when the user input action happens, as well as the timestamp when the corresponding response of the action manifests.

In detail, we record specific interaction patterns for each game (e.g., invoking game menus, changing views, and touching in-game buttons), which are performed with styluses instead of fingers to ease detection. After recording, we leverage conventional computer vision algorithms [43, 90, 92] to detect and track objects in the frames. We first detect the stylus and the screen from the side view of the device screen, and then recognize the very frame (when the distance between the stylus and the screen reaches the minimum) as the start of an interaction. After that, we track the pattern-specific response objects (e.g., button highlight) case-by-case in the front view of the device screen to identify the end of an interaction. Such an approach is common practice in existing work for collecting visual-related delays [31], and the accuracy of the algorithms can reach 99+% according to our random sampling-based manual checking. In this way, the interactive latency can be deemed as the time interval between the two timestamps, with a negligible error of up to two frames (∼1.0 *ms*) introduced by the camera.

Table 1: Eight representative cloud gaming platforms studied in this work, in alphabetical order by their names.

ID	Platform	Country	Type
1	Boosteroid	USA	CCG
2	CloudMoon	USA	MCG
3	GeForce Now	USA	CCG
4	JoyArk	USA	MCG+CCG
5	NetEase Cloud Gaming	China	MCG+CCG
6	Tencent Pioneer	China	MCG
7	Tencent Start	China	CCG
8	X-MCG	China	MCG

Further, we are curious about the network impact on the interactive latency since this is what common users are the most concerned with. Once a user suffers from high interactive latency, s/he is most likely to complain about the network latency. To address the concern, almost all the commercial cloud gaming platforms have on-screen widgets presenting the real-time network latency (via their client apps), which greatly facilitates our recording the network latency.[1] Since a low network bandwidth can also lead to high interactive latency, we periodically record the downlink bandwidth with the lightweight Android API *NetworkStatsManager.querySummaryForDevice()* in the meantime.

As listed in Table 1, we study 8 representative cloud gaming platforms that are highly popular in either the USA or China, using 100 mobile devices of 43 device models (1.8-3.3 GHz CPU frequency, 4-16 GB memory, and Android 9-13) geo-distributed in the corresponding countries (50 in 3 cities in the USA and 50 in 3 cities in China). Among them, three are CCG platforms, three are MCG platforms, and two support both CCG and MCG. We include CCG in our measurements to understand latency differences between MCG and CCG.

For each platform, we select its top-10 popular games to measure the interactive latencies with different radio access technologies (RATs): 5G (37%), 4G (29%), and WiFi (34%). We set the streaming resolution and frame rate of each platform to 1080P and 60 FPS respectively, which are the default settings of most platforms. Also, we use the 100 mobile devices to play games at different times (0:00, 3:00, ..., 21:00, UTC+8) every day during a whole month (May. 2023), so as to figure out the influence of user request workloads. The duration of game playing each time is five minutes.

Ethical Claim. All the measurements and analyses throughout this work are conducted with explicit permission of users and no ethical concern is raised in this work.

[1] Some platforms may not be reporting accurate latency values, and different platforms may have different methodologies to calculate the latency.

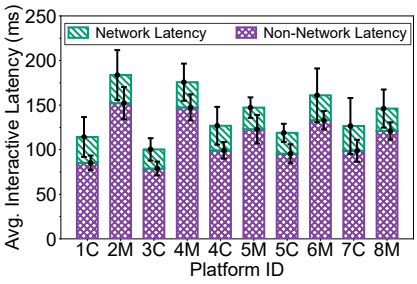


Figure 3: Average interactive latency of each platform. "M" and "C" denote MCG and CCG respectively.

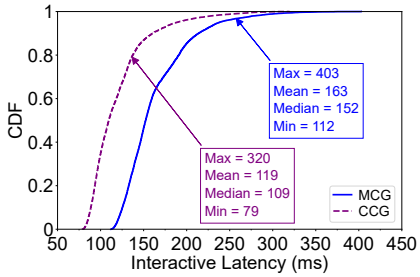


Figure 4: Interactive latencies of our studied MCG and CCG platforms.

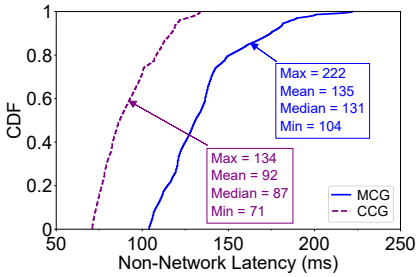


Figure 5: Non-network latencies (i.e., interactive latency excluding network latency) of MCG and CCG platforms.

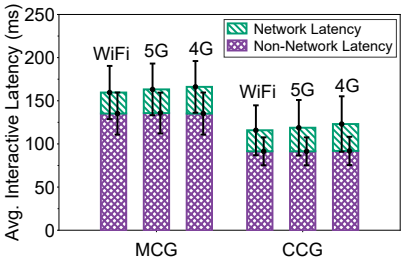


Figure 6: Average interactive latencies of our studied MCG and CCG platforms with different RATs.

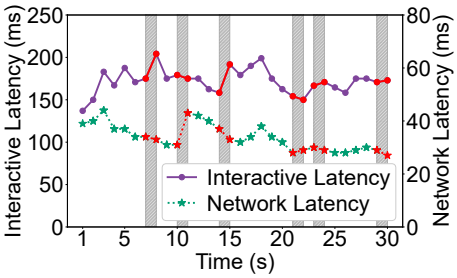


Figure 7: Interactive and network latencies during a typical measurement on top of Tecent Pioneer.

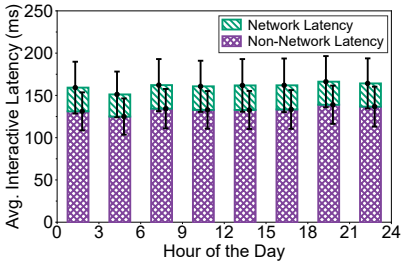


Figure 8: Average interactive latency of Tencent Pioneer at different times of a typical day (time zone: UTC+8).

2.2 Measurement Results

Among our records of individual measurements, a small portion (6%) are undesirable because it is difficult for us to analyze their relevant interactive latencies. For example, when the downlink bandwidth is insufficient, we can hardly quantify its remarkable and highly fluctuating influence on the interactive latency, thus making our analyses obscure. After ruling out such cases, we obtain a total of 20,096 valid records, which reveal quite a few surprising results with regard to interactive latencies of today's commercial MCG platforms.

General Statistics. All the studied MCG platforms suffer from unsatisfactory interactive latencies ranging from 112 *ms* to 403 *ms*—recall in §1 that smooth gaming experiences require a low interactive latency within 100 *ms*. In contrast, in non-negligible (35%) cases interactive latencies of CCG stay below 100 *ms*. Obviously shown in Figure 3 (in all the bar charts throughout this paper, each error bar denotes the standard deviation of the corresponding sub-bar), interactive latencies of MCG are generally higher than those of CCG.

As shown in Figure 4, our measured minimum interactive latency of MCG (112 *ms*) is 42% larger than that of CCG (79 *ms*), and the average interactive latency of MCG (163 *ms*) is 37% larger than that of CCG (119 *ms*). Regarding average interactive latency, even the fastest MCG platform (8M: X-MCG) is slower than the slowest CCG platform (4C: JoyArk). We attribute the above to the more complex processing logic of MCG compared with CCG, i.e., the mobile OS virtualization specially required by MCG. In fact, CCG platforms typically use hardware-assisted virtualization techniques (e.g., SR-IOV [62]) to share resources to VM instances, whose performance is close to direct hardware access. But for MCG, given the poor driver support of desktop GPUs in mobile OSes, para-virtualized GPUs with additional virtualization logic are usually required. Hence, there exists a gap in rendering performance between the CCG and MCG backends.

Network Impact. Contrary to most users' common beliefs, most of the excessive MCG latency is not attributed to the network. With the network latency excluded (then the remaining latency is referred to as the *non-network latency*), the minimum non-network latency of MCG is still as high as 104 *ms* (>100 *ms*), 46% larger than that of CCG (71 *ms*), as indicated in Figure 5. This is because all our studied platforms have deployed geo-distributed cloud and/or edge servers across either the USA or China, making the network latency pretty low in most cases. Concretely, in 81% of the valid records, the user device lies in the same city as the cloud/edge server. More in detail, no matter which RAT is used, the network latency (averaging between 24 *ms* and 31 *ms*) only takes up a small portion (averaging between 15% and 25%) of the interactive latency, as indicated in Figure 6.

Common wisdom suggests a positive correlation between interactive latency and network latency. Surprisingly, in a non-trivial portion (13%) of our measurements the interactive latency is negatively correlated with the network latency. To visualize this, Figure 7 depicts results during a typical measurement that includes such unexpected cases. Dots under

shadows denote cases of a negative correlation between interactive latency and network latency. The reason behind the above counter-intuitive phenomenon will be detailed in §3.3.

User Request Workload. Figure 8 presents average interactive latencies at different times of a typical day on Tencent Pioneer, the most popular MCG platform in our measurement. There is no significant difference between the interactive latencies during peak hours (12:00 – 21:00) and non-peak hours (00:00 – 12:00, 21:00 – 24:00). Further, we do not see notable variation among the interactive latencies in different days of a week, either. The above hold true for other MCG platforms as well, revealing that today's MCG platforms possess sufficient scalability to serve a fluctuating user request workload.

Game Rendering Workload. Recall that for each platform, we select its top-10 popular games to play and measure interactive latencies, including simple 2D games, complex 3D games, and so forth. Surprisingly, we note that on all MCG platforms, simple 2D games with light rendering workloads suffer from higher interactive latencies (10% on average) compared with graphics-intensive 3D games. Given that official documents of some MCG platforms mention that they selectively schedule GPU resources for different game types [80], we suspect that the surprising observation might be owing to unbalanced resource allocation strategies, i.e., overly prioritizing the QoE of complex 3D games while ignoring requirements of lightweight 2D games. To validate the conjecture, we consulted the customer service of each MCG platform, and two of them acknowledge that simple 2D games are oftentimes subject to insufficient GPU resource allocations.

3 Diagnosing the Undesirable Interactive Latency of MCG

To further discover the root causes behind the above observations, we contacted the developers of the five commercial MCG platforms. Eventually, the developers of X-MCG agreed to share their design with us and collaborate with us to diagnose the undesirable interactive latency atop their platform.

In this section, we first introduce important background knowledge regarding mobile graphics pipelines and VSync (§3.1). Then, we dissect the end-to-end interactive loop of X-MCG into 16 pipeline stages (§3.2). Afterwards, we measure the latencies of key stages in the X-MCG pipeline to quantitatively explain why the interactive latency is unsatisfactory (§3.3). We will discuss system architectures of other MCG solutions and the generalizability of our findings in §6.

3.1 Mobile Graphics Pipeline and VSync

The graphics pipelines of modern mobile OSes consist of multiple *stages* due to the users' diverse needs for graphics; each stage performs its unique functionality. When a game is played on an Android device, for instance, the game is typically the beginning of the graphics pipeline. The game performs graphics rendering by writing its contents into a frame buffer (or *frame* for short) from a buffer queue termed `Surface` [9]. At the same time, other system apps (e.g., system UI) may render contents into their own `Surfaces`.

Various `Surfaces` are connected to a compositor called `SurfaceFlinger` [10], whose job is to merge the contents of individual `Surface` layers into one or more graphic frames for display. In other words, after game frames are rendered, `SurfaceFlinger` conducts *Layer Composition* [14] on the frames. The composited result is then sent to a Hardware Composer (`HWcomposer`) module [13], which submits the contents to the actual display hardware.

It is worth noting that the stages usually work in their own paces, which can cause frame access issues. For example, while `SurfaceFlinger` regularly conducts Layer Composition at the refresh rate of the display, the pace of game rendering can be rather uncertain. When the game delivers a new frame to `SurfaceFlinger` while it is compositing, the composited result may contain portions of multiple game frames, leading to the *screen tearing* [53] problem.

To address screen tearing, modern mobile OSes typically adopt the VSync mechanism; other mechanisms (e.g., Triple Buffering [34]) have also been proposed, but each of them alone is generally unsuited to power-restrained mobile systems without the help of VSync [2]. As mentioned before, VSync is a classic access control mechanism (or synchronization primitive) for the graphics pipeline, which controls the frame intervals of two adjacent stages to make the frame access exclusive. It works in an event-driven manner: frame consumption is triggered periodically by VSync events at a pace specified by the consumer (e.g., when the display with a refresh rate of 60 FPS is the consumer, the period is 1/60 s). Note that the frequency and period of VSync events are determined by the frame rate of the consumer in a graphics pipeline, and thus cannot be adjusted freely.

In each period, when the producer finishes frame production, it waits for the *vertical blanking interval* [44] (i.e., the time interval between the end of a production and the next VSync event of the consumer). Then, it delivers the frame to the consumer only when no consumption (e.g., frame drawing) is in progress, so as to avoid screen tearing while saving its power (i.e., preventing the producer from generating too many frames that cannot be processed by the consumer). In other words, when the processing of one frame is finished, the graphics pipeline will not process a new frame immediately; instead, the processing of a new frame will be delayed till the signaling of the next VSync event, thus creating an uncertain latency. Worse still, the latencies can accumulate in a lengthy graphics pipeline, leading to unsatisfactory user experiences.

3.2 The Interactive Loop of X-MCG

X-MCG is developed and commercialized through the combinatory use of Trinity and Sunshine on the cloud side, as well as extending Moonlight (the client of Sunshine) as the client app. The developers of X-MCG carefully make the choice by conducting benchmarks on various existing MCG solu-

tions, which will be elaborated in §6. Further, they customize the AOSP (Android Open Source Project) [11] system that runs within the cloud servers by pruning unnecessary system services like `BatteryService` [76].

The high-level interactive loop of X-MCG is presented in Figure 1. It comprises as many as 16 stages as shown in Figure 9, which will be detailed below. The pipeline starts from the user interaction with the client display hardware, and the client app captures user inputs **(Stage 1: User Input Capture)**. Inputs are then sent to the cloud-side streaming component Sunshine **(Stage 2: User Input Transmission)**. Upon receiving inputs, Sunshine injects them into the mobile game application that runs on the guest OS **(Stage 3: Input Injection)**. As mentioned before, the guest OS is a customized AOSP system that runs within the Trinity emulator.

Afterwards, the game prepares to process the inputs and render the game graphics. To avoid screen tearing and conserve power, mobile games synchronize the game's frame rate to a portion of the display refresh rate of the device [28, 81]. As explained in §3.1, they use VSync events to control how quickly game frames should be produced, which is usually specified by the developer/user via an in-game option. Consequently, user inputs need to wait for a VSync event before it gets processed and the game rendering starts **(Stage 4: VSync1)**, and VSync1 cannot be disabled as it is built deep into the game rendering logic for supporting fixed-refresh-rate mobile display.

With regard to game rendering, modern mobile games usually use rendering APIs like OpenGL ES [16] to render into their `Surfaces` **(Stage 5: Game Rendering)**. In order to achieve GPU-accelerated rendering in AOSP, typical solutions (e.g., Google Android Emulator [8] and DroidCloud [47]) forward render instructions from the GPU driver in AOSP to the host-side virtual GPU device using remote procedure calls (RPCs), which turns out to be slow and expensive. In contrast, through the novel *graphics projection* mechanism [33], the Trinity emulator directly resolves the vast majority (>99%) of rendering API calls within the guest GPU driver, significantly reducing RPC calls and improving the rendering performance.

When game rendering finishes, the results are submitted to `SurfaceFlinger`. `SurfaceFlinger` has to work in the same pace with the display hardware, and therefore needs to wait for VSync **(Stage 6: VSync2)** before compositing the output of the **game app** including system UIs **(Stage 7: Layer Composition I)**. Subsequently, the composited frame is sent to `HWcomposer`, which waits until the display hardware has finished the scan-out [32] of the previous frame **(Stage 8: VSync3)**. Then, `HWcomposer` presents the frame to the virtual offscreen display device provided by the Trinity emulator **(Stage 9: Virtual Display)**. VSync2 and VSync3 can introduce fluctuating delays due to highly variable game rendering workloads as well as possible resource contention in commercial MCG platforms, given that hardware resources are typically shared to a large number of virtualized instances

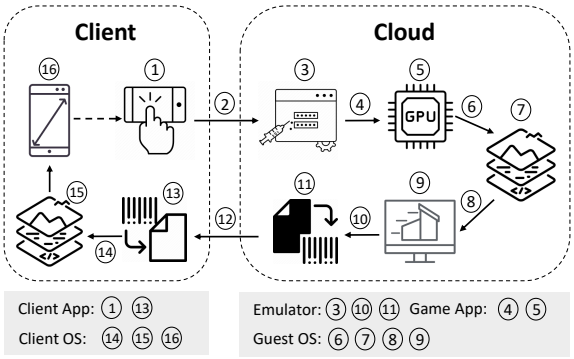


Figure 9: The 16 stages in the interactive loop of X-MCG.

with diverse game workloads.

Afterwards, Virtual Display forwards the frame to the Sunshine encoder. The encoder performs frame encoding according to the video frame rate set by the user, so a separate VSync is used **(Stage 10: VSync4)**. VSync4 exists because the client-side video frame rate may be different from the cloud-side game frame rate, and simple strategies like delivering frames immediately cannot satisfy the frame rate requirement of the client. The encoder and the Virtual Display are coordinated in the host virtual GPU, so that frame forwarding and encoding take place in the GPU with minimal frame copies between GPU and CPU **(Stage 11: Frame Encoding)**.

Over the network, the encoded game frame is transmitted to the client **(Stage 12: Game Frame Transmission)**, which collects and decodes the frame **(Stage 13: Frame Decoding)**. Then, similar to VSync2, the decoded frame waits for another VSync event triggered by the client `SurfaceFlinger` **(Stage 14: VSync5)** before conducting Layer Composition **(Stage 15: Layer Composition II)**. Note that VSync5, as a client-side VSync, can be disabled, but only with developer/root privileges. Furthermore, once it is disabled, the effect is global and other apps will also suffer from side-effects like screen tearing and increased power consumption, which is thus unacceptable to normal users. Finally, the composited frame is scanned out by the display hardware **(Stage 16: Display)**[2].

3.3 Root Cause Analysis

Among the 16 stages, we want to uncover which ones contribute the most to the interactive latency. To this end, we conduct an in-depth measurement on the end-to-end loop of X-MCG using 10 phones (with different device models) whose OSes are tailored for detailed latency recording. Given that delays of Stage 1 and Stage 16 are mostly induced by the display hardware and thus can hardly be measured by software means, we focus on measuring delays of Stages 2-15 (so that we can infer the sum delay of Stage 1 and Stage 16).

To precisely trace the data flow of a specific game frame in a lightweight manner, we measure passively by instru-

[2]Similar to VSync3, the composited frame needs to wait for another VSync event before the display scan-out, but as this VSync lies deep inside the client system and is invisible to user apps, we merge it to **Stage 16**.

Table 2: Latency breakdown of X-MCG (in unit of *ms*).

Category	Mean	Median	Min	Max
Network	15.7	16.4	7.9	24.1
VSync	43.9	44.0	4.5	82.9
Game Rendering	27.9	28.2	25.7	33.2
Video Processing	27.2	28.9	18.1	46.7
Layer Composition	5.4	5.1	3.0	6.6
Others	2.6	2.3	1.6	3.9

menting relevant system (Android, AOSP, Trinity and Sunshine/Moonlight) modules and recording the timestamps of relevant network packets. Specifically, we measure the network latency (Stage 2 and Stage 12) by piggybacking timestamps with network packets between Sunshine and Moonlight. In the cloud-side AOSP system, we instrument system services including `InputDispatcher`, `SurfaceFlinger`, and `HWComposer` in the framework layer, as well as the OpenGL ES library in the Hardware Abstraction Layer (HAL). In this way, delays of Stages 3-8 can be accurately recorded. In Trinity, we instrument the virtual display module to obtain the delay of Stage 9. Further, through tracking the encoding process of Sunshine, delays of Stage 10 and Stage 11 can be clearly observed. Finally, we instrument Sunshine and the client Android system to measure the delays of Stages 13-15.

Based on the above fine-grained instrumentation throughout the pipeline, we collected data at various times of each day during a week, and obtained the latency traces for a total of 35 hours of game playing. The measurement overhead is confined to a negligible level: the average CPU utilization is 1%, and the average memory consumption is 703 KB. We classify the delays of the measurable stages (Stages 2-15) in the loop into six categories, including Network (Stage 2 and 12), VSync (Stages 4, 6, 8, 10, and 14), Game Rendering (Stage 5), Layer Composition (Stages 7 and 15), Video Processing (Stages 11 and 13), and Others (Stages 3 and 9).

As listed in Table 2, in terms of mean latency, the VSync mechanism (5 VSync events) contributes the most (35.7%) to the non-network latency, even more than Game Rendering (22.7%) and Video Processing (22.1%). Moreover, delays incurred by VSync events are highly fluctuating between 4.5 *ms* and 82.9 *ms* (even more fluctuating than the network latency in our measurements), causing the interactive latency to vary significantly even when the network latency is stable. The above findings explain the counter-intuitive phenomenon we observe in §2.2—a lower network latency does not necessarily translate to a better interactive latency, since network is usually not the deciding factor. As for the "butterfly effect" mentioned in §1, it happens mostly when network jitters cause subsequent pipeline stages to miss VSync events according to our observation.

As shown in Table 3, each VSync event is required by an individual stage. Delving deeper into the loop, we discover that 5 VSync events are tightly coupled with other stages. VSync1 has been built into modern game engines like Unity [82] and Unreal [29] (Stage 5). VSync2, VSync3, and VSync5 are required by core graphics modules of both cloud-side and client-side Android systems, (namely `SurfaceFlinger` and `HWComposer`) (Stages 7, 9, 15). VSync4 is necessitated by the video encoder to encode videos at user-defined frame rates (Stage 11). Unfortunately, a lack of coordination between the VSync events causes an uncertain delay for each game frame with respect to each event. According to our measurements, the latencies of 5 VSync events all exhibit a nearly uniform distribution between 0 and 16.7 *ms*.

Table 3: Specific roles of the five VSync events and the average latency incurred by each VSync event.

Stage	Caused By	Location	Latency
4	Game Rendering	Game	8.5 *ms*
6	Layer Composition I	Guest Android	9.0 *ms*
8	Virtual Display	Guest Android	9.5 *ms*
10	Frame Encoding	Encoder	9.1 *ms*
14	Layer Composition II	Client Android	7.9 *ms*

Fortunately, while all the VSync events are *structurally* required and are hard to disable from the perspective of an MCG platform, not every VSync event is *functionally* useful. In the context of MCG, games are usually the only foreground apps besides system UIs (e.g., Android task bar), so Layer Composition and Virtual Display (Stage 7 and Stage 9) in the guest Android system (AOSP) only append game-irrelevant system UIs to a frame in almost all cases. Thus, these stages (along with their VSync events) are *functionally* unnecessary for MCG, as the game frames are ready to be displayed on the client after Game Rendering (Stage 5). This motivates our attempt to bypass some VSync events and decouple game rendering from the lengthy graphics pipeline, hoping to avoid or reduce their impact on the interactive latency.

4 System Design

Since the VSync mechanism is the biggest contributor to the interactive latency of X-MCG, we wish to minimize its negative impact. Guided by our diagnosis insights in §3.3, we design LoopTailor, an adaptive whole-loop regulator of VSync events for lowering the interactive latency of X-MCG to a desirable level (<100 *ms*). In this section, we first present a brief overview of LoopTailor along with the design goals (§4.1). Then, we detail its two key components: Game Frame Interceptor (§4.2) and Remote VSync Coordinator (§4.3).

4.1 LoopTailor Overview

LoopTailor sets out to meet the following design goals:

- Bypassing VSync events that are functionally unnecessary without disrupting the frame access control of the graphics pipeline (i.e., without harming the game's visual display);
- Accurately predicting and aligning the remaining VSync events to minimize their incurred latency;
- Introducing very little computational overhead and negligible impact on the smoothness of game graphics.

Figure 2 shows the architectural overview of LoopTailor, which streamlines the interactive loop of X-MCG by cropping the cloud-side graphics pipeline and stitching it to the client-side pipeline in a fine-grained manner. At the heart of LoopTailor are two key components: 1) Game Frame Interceptor (GFI, §4.2) that efficiently captures the game frames ahead of schedule to bypass 2 (out of 5) VSync events; and 2) Remote VSync Coordinator (RVC, §4.3) that forecasts and synchronizes the remaining 3 VSync events by carefully coordinating the cloud and the client. Note that the five VSync events are necessitated by the Android system design and the nature of constant-frame-rate video codec; apart from our emulator-based solution, the high-level design of LoopTailor is applicable to other (potential) solutions like container-based or function-based serverless gaming as well.

4.2 Game Frame Interceptor (GFI)

GFI aims to capture raw frames from game rendering ahead of time, before any form of layer composition. In Android, the desired interception should take place between the game app and `SurfaceFlinger` (before Stage 6 in §3.2), and there exist multiple intuitive methods. App-level render redirection [17], for instance, redirects the app rendering `Surface` to a virtual display with hooking libraries such as XPosed [87], and uses the `MediaProjection` API [15] to directly record rendered contents. Framework-level buffer hooking [83] adds a hook inside `Surface`, and performs interceptions whenever a game-owned `Surface` delivers a frame to `SurfaceFlinger`.

Nevertheless, for MCG where captured frames need to be subsequently encoded and sent to the client device, such intuitive approaches incur considerable frame copy overheads (note that frame encoding in the original X-MCG pipeline is in-place, c.f., Stage 11 in §3.2). In essence, both approaches capture frames solely from *inside* the guest, but ultimately send the frames using the network provided by the host. Therefore, data copies between the guest and host are unavoidable.

To realize low-overhead frame interception, our solution breaks the virtualization boundary and captures game frames in place. Figure 10 plots the workflow of GFI. The high-level idea is to leverage cross-layer information from both the guest system and the host-side virtual GPU to intercept and forward *handles* (as opposed to the actual frame content), so as to minimize data copies. In detail, we customize the `gralloc` graphics allocator [12], the guest GPU driver, along with the virtual GPU device in Trinity. `Gralloc` provides crucial information regarding frame correlations across the virtualization boundary, so that we can identify the actual host-side GPU resources using the guest-side frame identifier. Meanwhile, in order to separate render instructions of the target game from other apps like Android system UI, we modify the guest GPU driver, adding Android framework-level `Surface` information into GPU render contexts.

Based on the above cross-layer information, we monitor

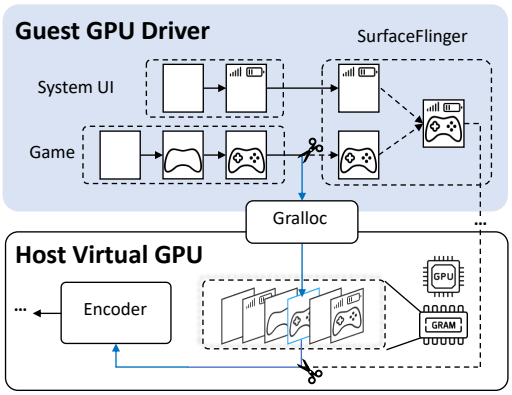


Figure 10: The workflow of GFI.

frame swap events (`eglSwapbuffers`) in the GPU driver, so that when a game-owned `Surface` delivers a frame to `SurfaceFlinger`, we can extract the actual resource handle(s) referring to the game frame in the host-side virtual GPU device. Afterwards, the intercepted resource handle(s) are immediately forwarded to the video encoder module, which utilizes the *interop* libraries provided by the GPU vendor (e.g., `nvenc` [60] for NVIDIA GPUs) to achieve in-place colorspace conversion and video encoding. In this way, data exchange between the CPU and the GPU, as well as between the guest and the host, is kept minimal.

With the game frames intercepted, Stages 6-9 (c.f. §3.2) in the original graphics pipeline can be bypassed, including two VSync events, Layer Composition I, and Virtual Display. Additionally, for `SurfaceFlinger` and the virtual display device in Trinity, we disable their rendering logic to reduce the GPU load, since the composited frames are no longer needed (frame swapping is preserved to not disrupt the data flow). Moreover, to support a minority of games that use multiple `Surfaces` for video playback or UI overlay, we add a trimmed-down compositor in the encoder to replace the original Android compositor in the VM, since the original Android compositor introduces considerable overheads (e.g., computing visible regions, matching display devices, and drawing layers). Specifically, we customize the Android compositor by 1) discarding game-irrelevant layers, 2) removing unnecessary post-processing (e.g., submitting frames to display), and 3) synchronizing the composition pace with the encoder to minimize latency overheads. Instead of relying on a standalone VSync (which defeats our original purpose of bypassing VSync events), frames are cached and composited right before encoding, so the delay caused by frame composition in these corner cases can be essentially reduced.

4.3 Remote VSync Coordinator (RVC)

Design Overview. With the informative help of GFI (Game Frame Interceptor), RVC aims to align the remaining three VSync events (VSync1, VSync4, and VSync5). As VSync5 is controlled by the client OS and cannot be changed by a user

app, RVC proactively synchronizes the timing of the cloud-side VSync1 and VSync4 with the client-side VSync5. To this end, RVC first obtains the timing information of VSync5 from the Android frame pacing library [6] in the client[3], and then predicts the delay of each intervening stage between VSync1 and VSync5 based on hierarchical time series forecasting [85] **(Hierarchical Latency Prediction)**.

Afterwards, RVC performs Synergetic VSync Alignment which exploits the latency forecasts to adaptively postpone VSync1 **(Synergetic VSync Alignment for VSync1)** and strategically perform reactive frame encoding by decoupling VSync4 **(Synergetic VSync Alignment for VSync4)**. Hence, a game frame (whose rendering and encoding are driven by VSync1 and VSync4 respectively) can arrive at the client (more specifically, arrive at VSync5) just in time, without waiting for the signaling of VSync4 and VSync5.

Hierarchical Latency Prediction. There are four stages happening between VSync1 and VSync5: game rendering, frame encoding, network transmission, and frame decoding. Although the latency of each stage can be independently forecast as a time series and summed directly to get the overall latency, the prediction result is oftentimes highly biased due to error propagation [39].

To minimize the prediction error, our key insight is that the stages have latent correlations within the end-to-end graphics pipeline. For example, if the game player suddenly enters a complex scene, game rendering will take longer time, the encoder will need more time to encode the frame as a standalone key frame (i.e., I frame) [74], and the encoded frame will be larger in size, leading to longer network transmission time and longer client-side frame decoding time. Also, we note that the involved latencies can be hierarchically organized as shown in Figure 11. Therefore, we adopt hierarchical time series forecasting [85] based on the MinT approach [85].

First, we adopt regression trees [52] to independently forecast the time series at all levels of the hierarchy (termed *base forecasts*). We resort to the regression tree algorithm as opposed to other algorithms because it is effective for both series with seasonal features [67] and fluctuating series like network latencies [89]. Moreover, it incurs low computation (<0.5 *ms*) and memory (<1 *MB*) overhead. Efficiency is of particular significance in our case, since the predictions need to be done at intervals of a few seconds.

Regarding the inputs of the regression trees, we follow common practice [67, 89] and feed them with the most recent latency data, as well as relevant data including the size of the encoded frame, network loss rate, and network bandwidth passively estimated from the media stream [89]. Since we need to make predictions at the frame-level granularity, the time window (i.e., the time unit in the forecast) is set as the smallest VSync interval among VSync1, VSync4 and VSync5 (typically 1/60 *s*). Accordingly, the information window size (i.e., the number of history time windows) and the forecast horizon (i.e., the number of future time windows) are typically set as 240 (=60×4) and 60, respectively. More details about the setting of hyperparameters will be provided later in §5.4.

[3]The timings of client-side stages (e.g., packet receiving, decoding, and VSync5) are synchronized with the cloud server's clock, leveraging a widely-used clock offset estimation approach [22] and a classic skew compensation algorithm [56]. The historical latency series (including the one-way network latency) are measured according to the calibrated timestamps.

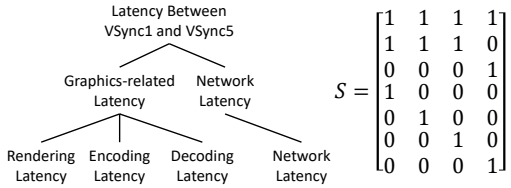


Figure 11: The tree and the matrix representation of the hierarchical latency series.

Given the base forecasts, we aim at minimizing the forecast errors by strategically reconciling the forecasts across the levels of the latency hierarchy. Let $\mathcal{D}_t = \{y_1, y_2, ..., y_t\}$ denote the history data observed up to time t, where y_i $(i = 1, 2, ..., t)$ is a vector containing all latencies in all levels of the hierarchy at time i. Let $\hat{y}_t(k)$ $(k = 1, 2, ..., K)$ denote forecasts of k future time windows based on $\mathcal{D}_t$, where k is the forecast horizon. Then, the optimal forecasts of k future time windows minimizing the conditional expectation

$$E[\| y_{t+k} - \hat{y}_t(k) \|_2^2 | \mathcal{D}_t], \tag{1}$$

where $\| \cdot \|_p$ is an L_p norm [30], is derived from

$$\mu_t(k) = E[y_{t+k} | \mathcal{D}_t]. \tag{2}$$

More details on optimal hierarchical forecasts are in [39].

If the conditional expectation (1) has errors, we wish to provide the best estimate of $\mu_t(k)$. Specifically, from the base forecasts $\hat{y}_t(k)$, we compute the revised forecasts as

$$\tilde{y}_t(k) = SP\hat{y}_t(k) \tag{3}$$

with a weight matrix $P \in \mathcal{P} \subseteq \mathbb{R}^{4\times 7}$, where $\mathcal{P}$ is the domain of P and S is a "summing" matrix of order 7×4 used to aggregate the lowest-level latency series, as shown in Figure 11. In detail, we compute the weight matrix P by solving the following optimization problem:

$$\min_{P \in \mathcal{P}} Tr(Var[y_{t+k} - \tilde{y}_t(k) | \mathcal{D}_t]) \text{ subject to } SPS = S; \tag{4}$$

$Tr(\cdot)$ is the trace of a matrix, based on MinT estimator [85].

To conclude, we realize combination forecasts for all levels of the latency series hierarchy through an optimal weight matrix P. In this way, we minimize the standard error of the coherent forecasts across the entire hierarchy.

Synergetic VSync Alignment for VSync1. With the predicted latencies in hand, the paces of VSync events can be adjusted in a synergetic fashion. In practice, we adjust the timing of VSync1 by blocking game rendering in the virtual GPU. As demonstrated in Figure 12, properly adjusting VSync1 can

reduce the interactive latency, by making the rendered frames reflect more recent user inputs (e.g., the first three arrows in Input Series 2). Specifically, once alignment occurs, user inputs that happen in the interval highlighted by the red box (i.e., the first 3 inputs in Input Series 2) will be processed by "Rendering1" instead of by the original "Rendering2". Consequently, the response of these user inputs can be displayed earlier, leading to a lower interactive latency.

We use Monte Carlo sampling [71] to minimize the delay expectation of n future frames in the pipeline, where n equals the forecast horizon. Specifically, we let V_c denote the timestamp of the most recent VSync5 event in the client, and let $x = \{x_1, x_2, ..., x_n\}$ denote predicted timestamps when the client finishes frame decoding, where x_i is the timestamp of the i-th future frame from now on. Then, we denote the timestamps of subsequent relevant VSync5 events as $V = \{V_1, V_2, ..., V_n\}$, where V_i is computed as

$$V_i = V_c + \left(1 + \left\lfloor \frac{x_i - V_c}{I_c} \right\rfloor\right) \cdot I_c, i = 1, 2, ..., n. \quad (5)$$

Here I_c is the time interval of VSync5. Given x and V, the problem is to obtain the minimum positive value of o (denoting the offset of the current blocking adjustment) that minimizes $E(x, V, o)$, which is the latency expectation of the subsequent n frame processings. It is computed as

$$E(x, V, o) = \sum_{i=1}^{n} f(x_i, V_i, o). \quad (6)$$

Here $f(x_i, V_i, o)$ is the expected delay induced by the VSync5 event on V_i, which is computed as

$$f(x_i, V_i, o) = (I_c - (V_i - x_i - o)) \mod I_c. \quad (7)$$

Given that the latency expectation (6) is a discontinuous function, we approximate the optimal value of o based on Monte Carlo sampling. Finally, the VSync1 events are postponed with an offset of o, completing the alignment. In reality, we perform the alignment for VSync1 periodically every n frames, where n is the forecast horizon.

Synergetic VSync Alignment for VSync4. Given that the cloud-side game frame rate and the client-side video frame rate might be different, frame encoding is normally driven by VSync4, whose interval matches the video frame rate. However, with the help of GFI, game frames can be encoded in a reactive manner without waiting for VSync4, while maintaining client-side frame rate stability based on the forecasts.

RVC decouples frame encoding from VSync4 events to reactively encode ready-to-consume frames in time. When the cloud-side game frame rate is lower than or equal to the client-side video frame rate, all produced game frames need to be encoded for streaming to satisfy the client-side requirement as much as possible. Otherwise, the forecasts are used to determine whether the encoder should encode or drop a game frame, to match the video frame rate requested by the user.

The encoder memorizes the timeline of the original VSync4 events, as well as the timestamp of the last encoded frame.

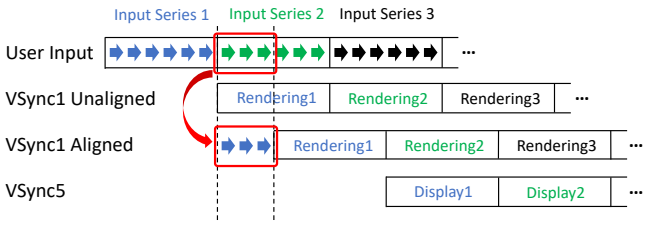


Figure 12: An example of Synergetic VSync Alignment. After the alignment, the first three inputs in Input Series 2 are processed ahead of schedule.

When a new game frame F_i arrives at T_i, it is encoded if the age of the last encoded frame exceeds one VSync interval, or if the predicted arrival time T_{i+1} of the next frame F_{i+1} satisfies $|T_{i+1} - T_v| \geq |T_i - T_v|$ (T_v is when the next VSync4 event occurs). If neither condition holds, F_i will be dropped since F_{i+1} better fits the client-side video pace. In this way, we can avoid delays incurred by VSync4 while adaptively maintaining the frame rate stability of the video.

5 Evaluation

We first describe our evaluation setup in §5.1, followed by evaluating the overall performance of LoopTailor in §5.2 and assessing the contributions of individual modules in §5.3. In §5.4, we present several micro-benchmarks to quantify the impact of key parameters and network dynamics.

5.1 Experimental Setup

Collaborating with X-MCG, we deploy LoopTailor on their cloud infrastructure. This facilitates fair comparisons between the original X-MCG and X-MCG enhanced with LoopTailor, which we refer to as LoopTailor for brevity. We also compare LoopTailor with other representative MCG platforms studied in §2.1: Tencent Pioneer, NetEase Cloud Gaming, JoyArk, and CloudMoon. Regarding client devices, we leverage the 100 mobile devices used in our measurement study (§2.1). Other settings (including camera settings, game types, game play duration, RAT, streaming resolution / frame rate, etc.) are the same as those in §2.1. We use the 100 mobile devices to collect evaluation data during a whole month (Nov. 2023), and collected a total of 21,743 valid records.

We also implement two intuitive baseline solutions based on the X-MCG architecture to evaluate the performance of LoopTailor in more depth. (1) Disable VSync: Since the client-side VSync5 and the in-game VSync1 cannot be controlled by an MCG platform, we disable VSync2/3/4 in the cloud. (2) In-VM Streaming: With the help of virtualized codec and NIC devices, videos are streamed in the guest OS (though the host NIC should still be used).

5.2 Overall Performance

Figure 13 plots the average interactive latency for X-MCG, LoopTailor, and four other MCG platforms (refer to Table 1 for their name-ID mappings), with a breakdown between

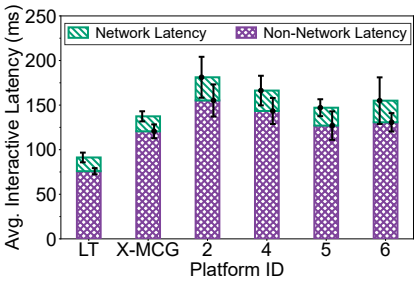


Figure 13: Average interactive latency. LT denotes LoopTailor.

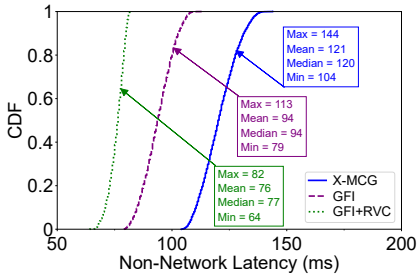


Figure 14: Performance breakdown with regard to non-network latency.

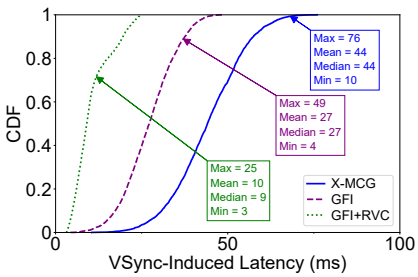


Figure 15: Performance breakdown regarding VSync-induced latency.

Table 4: Latency comparison (in milliseconds) between LoopTailor and baseline solutions. "Tail" refers to 99% percentile.

Solutions	Interactive Latency			Non-Network Latency		
	Avg.	Med.	Tail	Avg.	Med.	Tail
LoopTailor	91	91	95	76	77	82
Disable VSync	106	105	121	90	90	104
In-VM Streaming	141	139	162	125	124	146

the network vs. non-network latency. As shown, LoopTailor yields the lowest average interactive latency of 82-96 *ms* (91 *ms* on average, 95 *ms* in 99% percentile), 35% lower than the original X-MCG (139 *ms* on average, 158 *ms* in 99% percentile). Recall that a smooth gaming experience requires a sub-100 *ms* interactive latency, which is achieved by LoopTailor. Note that the desirable interactive latency of LoopTailor is also ascribed to X-MCG's wide deployment of edge servers in the latter half of 2023. The non-network latency of LoopTailor is only 64-82 *ms* (76 *ms* on average, 82 *ms* in 99% percentile), 37% lower than the original X-MCG (121 *ms* on average, 141 *ms* in 99% percentile) and 40% lower than the best of the other four MCG platforms (127 *ms* on average, 150 *ms* in 99% percentile).

As shown in Table 4, LoopTailor outperforms the baseline solutions in terms of interactive latency and non-network latency. Although it appears that LoopTailor has no significant latency advantage over Disable VSync, the latter introduces severe side effects including unstable frame rate and screen tearing [42]. As for In-VM Streaming, it performs the worst due to the additional overheads of virtualized codec and NIC devices for multiplexing hardware resources.

Regarding other QoE metrics for cloud gaming, we observe that the average client-side video frame rate stays high at an average of 59.8 FPS, with a small standard deviation of 0.6. This indicates that LoopTailor is capable of delivering a smooth game streaming experience. Regarding the image quality of LoopTailor, we do not observe jittery frames or degradation of image quality compared to original X-MCG.

5.3 Contributions of Individual Modules

We evaluate the performance gain brought by each module of LoopTailor: GFI (Game Frame Interceptor) and RVC (Remote VSync Coordinator), by enabling them incrementally. We exclude the network latency when reporting the results.

We begin with only enabling GFI. Recall that GFI eliminates VSync2 and VSync3. As shown in Figure 14, this results in an average non-network latency of 94 *ms*, a 22% reduction compared to X-MCG. The *maximum* non-network latency of GFI-only LoopTailor is even 7% lower than the *average* non-network latency of X-MCG. To delve deeper into benefits of GFI, we measure the latency reduction of different stages. First, we consider the latency incurred by only VSync events (termed VSync-induced latency) and plot its distributions in Figure 15. As shown, GFI reduces the VSync-induced latency by 38% on average. Second, GFI helps cut the layer composition latency by 52% since several unnecessary pipeline stages are bypassed. Third, GFI further helps accelerate game rendering by 6% because with a few pipeline stages bypassed, more resources can be allocated for rendering.

Next, we enable RVC to make a full-fledged LoopTailor system. As in Figure 14, RVC further reduces the average non-network latency by 19%, compared to using GFI alone. This translates to a 37% reduction in the total non-network latency compared to X-MCG. Through Synergetic VSync Alignment, RVC greatly reduces the average VSync-induced latency by 57% (73%) compared to using GFI alone (X-MCG). Unlike GFI which only optimizes the cloud side, RVC tackles both the cloud and the client side. It reduces the average latency incurred by VSync5 on the client side from 8 *ms* to 3 *ms*.

The above results demonstrate both GFI and RVC are essential for LoopTailor. Their synergy allows LoopTailor to reach a consistently low non-network latency of 64-82 *ms*.

5.4 Micro-benchmarks

Impact of the Information Window Size. Recall from §4.3 that the information window size (IWS) determines the number of time windows used in latency prediction. Figure 16 shows the relationship between the IWS and the standard error of latency prediction (the forecast horizon is set to 1 in this experiment). As shown, the standard error first decreases and then increases. The decrease is because larger information windows usually provide more sufficient obervations on history data, whereas the increase is explained by too many outdated history data within the window.

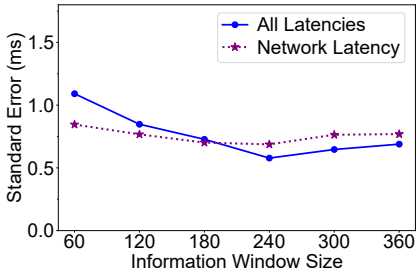


Figure 16: Avg. std error of RVC with different information windows.

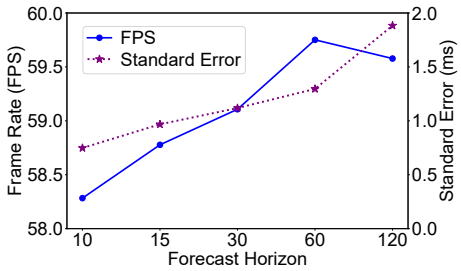


Figure 17: Avg. video frame rate & std error with different forecast horizons.

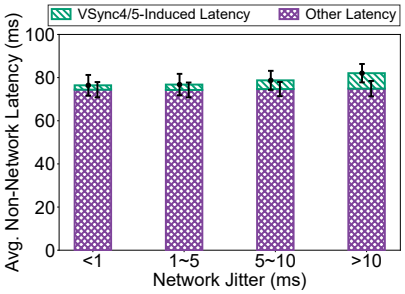


Figure 18: Avg. non-network latency with different network jitters.

Given the existence of key frames in game rendering, it is preferable to set the IWS to be larger than the intervals between each pair of key frames (usually about 10 to 20 VSync intervals based on our measurements). Meanwhile, IWS should not be too large considering both the forecasting errors and computation overheads. Figure 16 shows that RVC achieves the lowest standard error when IWS is around 240, which reaches the minimum standard error (0.6 *ms*) with a low computation overhead (<0.5 *ms*). Regarding the prediction of network latency, RVC achieves an average standard error of <1 *ms* (0.69-0.84 *ms* as shown in Figure 16) under different IWS settings. The standard errors exhibit tiny fluctuations given that the network condition in our evaluation is stable with only ∼2 *ms* jitters. The performance of LoopTailor under diverse network conditions is discussed later in this section.

Impact of the forecast horizon. The forecast horizon determines the number of future time windows produced by a prediction in RVC. On one hand, a smaller forecast horizon leads to a higher frequency of VSync1 adjustments, as illustrated in §4.3; adjusting VSync1 too frequently may result in a slight client-side video frame rate drop, because an incorrect forecast may cause a frame deadline miss at the client. On the other hand, a large forecast horizon may increase the forecasting error. We experimentally quantify this tradeoff in Figure 17, where the IWS is set to 240. Note that the slight drop in the frame rate at a forecast horizon of 120 is due to the sharp increase in the standard error. Based on the results, we select a forecast horizon of 60 to balance the above tradeoff.

Impact of Network Jitters. LoopTailor should ideally maintain a low non-network latency even when the network condition is highly dynamic. Figure 18 shows the average non-network latency under different network jitters (i.e., standard deviations of network latencies). The results demonstrate the robustness of LoopTailor under diverse network conditions: with ≤10 *ms* network jitters, there is only negligible fluctuation in the non-network latency (75-78 *ms*); even when the network jitter exceeds 10 *ms*, the non-network latency only increases marginally to 82 *ms*. The slight increase in the non-network latency is caused by the inflation of VSync4/5-induced latency, which is attributed to the elevated forecasting error under highly fluctuating network conditions.

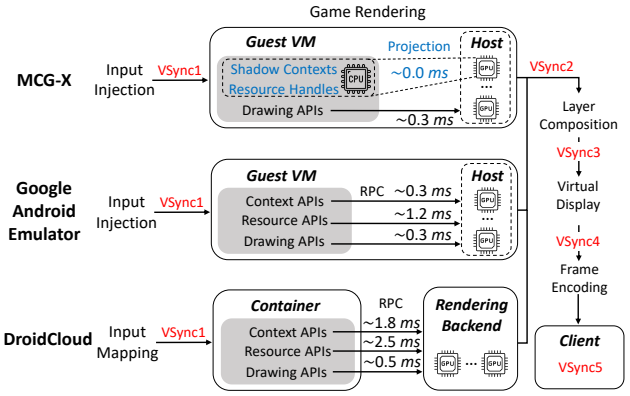


Figure 19: Architectural comparison of X-MCG and other representative MCG solutions. The numbers atop arrows refer to the average latency overhead per game frame.

6 Discussion

Generalizability. VSync1∼5 are not specific to X-MCG; in fact, it is impractical to disable them as they are necessitated by the design of the Android system (on both cloud and client sides) and the nature of constant-frame-rate video encoding (see §3.2). Thus, the defects of X-MCG are common to every Android- and video streaming-based MCG platform. Figure 19 illustrates a detailed comparison of the architectures of X-MCG and other representative MCG solutions. As shown, employing other solutions such as Google Android Emulator-based (abbreviated as GAE-based) and DroidCloud [47] also results in the five VSync events in the end-to-end graphics pipeline. Given the above facts, we believe that our insights and designs are widely applicable to other platforms.

The developers of X-MCG make a careful choice among existing MCG solutions by conducting comprehensive benchmarks on them. The experiment results reveal that the graphics virtualization mechanism of Trinity introduces ∼0.3 *ms* latency overhead per frame, 83%-94% less than other representative solutions [33]. Additionally, Sunshine exhibits 12.6%-26.7% lower end-to-end latency compared to other streaming tools [37, 59, 63] due to its efficient in-place frame processing capabilities. However, these advanced virtualization and frame processing techniques still need the VSync

mechanism for frame access synchronization with other stages in the graphics pipeline. Worse still, the high efficiency of rendering and frame processing of X-MCG may even amplify the impact of VSync events on the interactive latency.

Although we were unable to scrutinize other mobile OSes such as iOS, we believe our insights and design principles still apply, given that VSync is a graphics primitive and that the virtualization stack is widely used for the cloud gaming backend. In fact, the closed-source nature of iOS prompts many MCG applications running on iOS to leverage virtualized Android backends to stream mobile games [41]. Furthermore, iOS also uses layer compositors (e.g., Core Animation), and iOS cloud gaming backends must also use virtualization to achieve resource multiplexing.

Application scope. MCG platforms generally focus on enabling users with middle/low-end devices to smoothly play graphics-demanding mobile games. For budget reasons, the refresh rates of such devices are typically fixed and limited, allowing LoopTailor to fully exploit its capability to optimize the interactive latency for mainstream MCG users. For high-end user devices with adaptive/high refresh rates, things are different for VSync5 as the intervals between adjacent VSync5 events become variable/short.[4] In such cases, only part of RVC (i.e., Synergetic VSync Alignment for VSync1) is less effective. Other LoopTailor components are still effective, as VSync1∼4 are not on client devices.

7 Related Work

Cloud Gaming Systems. Based on where rendering instructions are executed, cloud gaming systems can be classified into two categories: host-rending and client-rendering. Host-rendering systems typically follow the thin-client architecture [19]. It offloads computation-intensive tasks to cloud servers, and livestreams rendered game frames as an encoded video to thin clients for display. Many systems follow this paradigm such as those proposed by the academia [37,40,47,48,64,76,86,88] and the industry [4,21,55,58,61,75,78,79].

Client-rendering systems, such as Games@Large [57] and LiveRender [49], intercept graphics instructions in the cloud and send them (along with graphics data such as vertices and textures) to the client for execution. While this paradigm can deliver high-fidelity graphics and enable secure GPU computing [38], it requires powerful client devices and networks [77], making it less attractive to low-end devices. Our work focuses on optimizing the performance of host-rendering systems given their dominant popularity.

Android in the Cloud. Landing Android onto cloud systems has become an emerging research topic. Existing cloud solutions typically run Android in virtual machines [7,8,91] or containers [21,47,76]. Considering the generality of both the Game Rendering Interceptor and Remote VSync Coordinator, we believe LoopTailor applies to most of the above systems. These systems generally use API remoting [27] or device emulation [65,69] for graphics virtualization, which can be laggy and inefficient due to, for example, excessive guest-host interactions [33]. To overcome such limitations, recent advances such as Trinity [33] (upon which we built LoopTailor) and vSoC [66] employ various techniques to reduce guest-host interactions, leading to higher graphics performance.

Another body of works set out to adapt Android to cloud environments. For instance, DroidCloud [47] enables Android to render on off-system GPU pools; CARE [76] cloudifies Android into a cloud-native system by streamlining Android system services. Our work instead operates on the graphics pipeline, and is orthogonal to the above research.

Latency Mitigation for Real-time Communication. Beyond VSync, there is a plethora of works on reducing the latency (incurred by rendering, network, and video coding) for real-time communication (RTC) systems such as MCG. They investigate a wide range of latency optimization dimensions including rendering acceleration [25,68], network congestion control [5,84], video coding [36,73,88], optimization model/metric [35,93,94], and cloud architecture [46,50,70]. Regarding graphics-related latency, recent studies [45,86] attempt to reduce it by predicting video frames, but they may fall short in real-world conditions due to the high randomness of user actions [45,86]. Contrasting above efforts, our work focuses on shortening the lengthy cloud-side graphics pipeline through a new dimension of regulating VSync events.

8 Conclusion

Mobile cloud gaming, as a highly interactive immersive application, faces new latency challenges due to its sophisticated backend graphics processing. In this paper, we identify VSync as a major contributor to the end-to-end interactive latency overhead and show that a specialized design can effectively minimize the VSync overhead. The result is a stable sub-100 *ms* interactive latency, leading to an average reduction of 35% compared with those provided by state-of-the-art MCG platforms. We believe that the ideas and key building blocks of LoopTailor, such as in-place frame capture and multi-stage latency prediction, have more profound implications to the design and implementation of emerging immersive applications, especially those operating in the edge/cloud-based model.

Acknowledgments

We are grateful to our shepherd, Zili Meng, for his invaluable feedback. We thank the anonymous reviewers for their insightful comments. This work is supported in part by the National Key R&D Program of China under grant 2022YFB4500703, the National Natural Science Foundation of China under grants 62332012 and 62472245, and the Ant Group.

[4] In Android, variable refresh rate is supported by switching between multiple discrete refresh rates rather than continuously adapting refresh rates (as of Sep. 2024) [1], and thus the client-side VSync5 always exists.

References

[1] Ady Abraham. High Refresh Rate Rendering on Android, 2020. https://android-developers.googleblog.com/2020/04/high-refresh-rate-rendering-on-android.html.

[2] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. CRC Press, August 2018.

[3] Ahmad Alhilal, Tristan Braud, Bo Han, and Pan Hui. Nebula: Reliable Low-Latency Video Transmission for Mobile Cloud Gaming. In *Proc. of ACM WWW*, pages 3407–3417, 2022.

[4] Amazon. Luna Cloud Gaming Service, 2024. https://amazon.com/luna/landing-page?ref=ln_social.

[5] Maryam Amiri, Hussein Al Osman, Shervin Shirmohammadi, and Maha Abdallah. An SDN Controller for Delay and Jitter Reduction in Cloud Gaming. In *Proc. of ACM MM*, pages 1043–1046, 2015.

[6] Android Developers. Android Frame Pacing Library, 2023. https://developer.android.com/games/sdk/frame-pacing.

[7] Android Developers. Cuttlefish Virtual Android Devices, 2023. https://source.android.com/docs/setup/create/cuttlefish.

[8] Android Developers. Google Android Emulator, 2023. https://developer.android.com/studio/run/emulator.

[9] Android Developers. Surface API, 2023. https://developer.android.com/reference/android/view/Surface.

[10] Android Developers. SurfaceFlinger and WindowManager, 2023. https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager.

[11] Android Developers. Android Open Source Project, 2024. https://source.android.com/.

[12] Android Developers. BufferQueue and Gralloc, 2024. https://source.android.com/docs/core/graphics/arch-bq-gralloc.

[13] Android Developers. Hardware Composer HAL, 2024. https://source.android.com/docs/core/graphics/implement-hwc.

[14] Android Developers. Layers and Displays, 2024. https://source.android.com/docs/core/graphics/layers-displays.

[15] Android Developers. Media Projection API, 2024. https://developer.android.com/guide/topics/large-screens/media-projection.

[16] Android Developers. OpenGL ES Overview, 2024. https://developer.android.com/develop/ui/views/graphics/opengl/about-opengl.

[17] Android Developers. Screen Capturing and Sharing, 2024. https://developer.android.com/about/versions/lollipop/android-5.0.

[18] Android Developers. The VSync Mechanism, 2024. https://source.android.com/docs/core/graphics/implement-vsync.

[19] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proc. of ACM SOSP*, pages 277–290, 2005.

[20] Boosteroid. Boosteroid Cloud Gaming, 2024. https://boosteroid.com/.

[21] Canonical. Anbox Cloud - Scalable Android in the Cloud, 2024. https://anbox-cloud.io.

[22] Qasim M. Chaudhari, Erchin Serpedin, and Khalid Qaraqe. On Maximum Likelihood Estimation of Clock Offset and Skew in Networks With Exponential Delays. *IEEE Transactions on Signal Processing*, 56:1685–1697, 2008.

[23] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. T-Gaming: A Cost-Efficient Cloud Gaming System at Scale. *IEEE Transactions on Parallel and Distributed Systems*, 30:2849–2865, 2019.

[24] China Mobile and ZTE. Powered by Sa: 5g Mec-Based Cloud Game Innovation Practice. Technical report, 2020.

[25] Shen Ciao, Zhongyue Guan, Qianxi Liu, Li-Yi Wei, and Zeyu Wang. Ciallo: GPU-Accelerated Rendering of Vector Brush Strokes. In *Proc. of ACM SIGGRAPH*, pages 1–11, 2024.

[26] CloudMoon. CloudMoon - Cloud Gaming, 2024. https://play.google.com/store/apps/details?id=com.nianwei.cloudphone&hl=en_US.

[27] Micah Dowty and Jeremy Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43:73–82, 2009.

[28] Epic Games. Frame Pacing for Mobile Devices, 2024. https://docs.unrealengine.com/5.3/en-US/frame-pacing-for-mobile-devices-in-unreal-engine/.

[29] Epic Games. Unreal Engine Official Website, 2024. https://www.unrealengine.com/en-US.

[30] Richard Farebrother. *L1-Norm and L∞-Norm Estimation: An Introduction to the Least Absolute Residuals, the Minimax Absolute Residual and Related Fitting Procedures*. Springer Science & Business Media, April 2013.

[31] Ilja T Feldstein and Stephen R Ellis. A Simple Video-based Technique for Measuring Latency in Virtual Reality or Teleoperation. *IEEE Transactions on Visualization and Computer Graphics*, 27(9):3611–3625, 2020.

[32] James D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1996.

[33] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. Trinity: High-Performance Mobile Emulation through Graphics Projection. In *Proc. of USENIX OSDI*, pages 285–301, 2022.

[34] Sumanta Guha. *Computer Graphics Through OpenGL®: From Theory to Experiments*. CRC Press, December 2018.

[35] Pouya Hamadanian, Doug Gallatin, Mohammad Alizadeh, and Krishna Chintalapudi. Ekho: Synchronizing Cloud Gaming Media across Multiple Endpoints. In *Proc. of ACM SIGCOMM*, pages 533–549, 2023.

[36] Luke Hsiao, Brooke Krajancich, Philip Levis, Gordon Wetzstein, and Keith Winstein. Towards Retina-quality VR Video Streaming: 15ms Could Save You 80% of Your Bandwidth. *ACM SIGCOMM Computer Communication Review*, 52(1):10–19, 2022.

[37] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. GamingAnywhere: An Open Cloud Gaming System. In *Proc. of ACM MMSys*, pages 36–47, 2013.

[38] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud GPUs. In *Proc. of USENIX NSDI*, pages 817–833, 2020.

[39] Rob J. Hyndman, Roman A. Ahmed, George Athanasopoulos, and Han Lin Shang. Optimal Combination Forecasts for Hierarchical Time Series. *Computational Statistics & Data Analysis*, 55:2579–2589, 2011.

[40] Iryanto Jaya, Yusen Li, and Wentong Cai. Improving Scalability, Sustainability and Availability via Workload Distribution in Edge-Cloud Gaming. In *Proc. of ACM MM*, pages 2987–2995, 2022.

[41] JoyArk. JoyArk – Explore and Play Games Instantly, 2024. https://joyark.com/.

[42] Viktor Kelkkanen, Markus Fiedler, and David Lindero. Synchronous Remote Rendering for VR. *International Journal of Computer Games Technology*, 2021(1):6676644, 2021.

[43] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment Anything. *arXiv:2304.02643*, 2023.

[44] David Large and James Farmer. *Modern Cable Television Technology*. Elsevier, 2004.

[45] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *Proc. of ACM MobiSys*, pages 151–165, 2015.

[46] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, Chen Sun, Gareth Tyson, and Hongqiang Harry Liu. LiveNet: A Low-Latency Video Transport Network for Large-Scale Live Streaming. In *Proc. of ACM SIGCOMM*, pages 812–825, 2022.

[47] Linsheng Li, Bin Yang, Cathy Bao, Shuo Liu, Randy Xu, Yong Yao, Mohammad R. Haghighat, Jerry W. Hu, Shoumeng Yan, and Zhengwei Qi. DroidCloud: Scalable High Density AndroidTM Cloud Rendering. In *Proc. of ACM MM*, pages 3348–3356, 2020.

[48] Yusen Li, Haoyuan Liu, Xiwei Wang, Lingjun Pu, Trent Marbach, Shanjiang Tang, Gang Wang, and Xiaoguang Liu. Themis: Efficient and Adaptive Resource Partitioning for Reducing Response Delay in Cloud Gaming. In *Proc. of ACM MM*, pages 491–499, 2019.

[49] Li Lin, Xiaofei Liao, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. In *Proc. of ACM MM*, pages 347–356, 2014.

[50] Yuhua Lin and Haiying Shen. CloudFog: Leveraging Fog to Extend Cloud Gaming for Thin-Client MMOG with High Quality of Service. *IEEE Transactions on Parallel and Distributed Systems*, 28:431–445, 2017.

[51] LizardByte. Sunshine Official Website, 2023. https://docs.lizardbyte.dev/projects/sunshine/en/latest/about/overview.html.

[52] Wei-Yin Loh. Classification and Regression Trees. *WIREs Data Mining and Knowledge Discovery*, 1:14–23, 2011.

[53] Sanjay Madhav. *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach*. Pearson Education, 2014.

[54] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling High Quality Real-Time Communications with Adaptive Frame-Rate. In *Proc. of USENIX NSDI*, 2023.

[55] Microsoft. Xbox Cloud Gaming, 2024. https://www.xbox.com/en-US/cloud-gaming.

[56] S.B. Moon, P. Skelly, and D. Towsley. Estimation and Removal of Clock Skew from Network Delay Measurements. In *Proc. of IEEE INFOCOM*, pages 227–234 vol.1, 1999.

[57] Itay Nave, Haggai David, Alex Shani, Yoav Tzruya, Arto Laikari, Peter Eisert, and Philipp Fechteler. Games@large Graphics Streaming Architecture. In *Proc. of IEEE ISCE*, pages 1–4, 2008.

[58] Netease. Netease Cloud Gaming, 2024. https://cg.163.com/.

[59] NVIDIA. NVIDIA GameStream, 2022. https://www.nvidia.com/en-us/support/gamestream/.

[60] NVIDIA. NVENC Video Codec SDK, 2024. https://developer.nvidia.com/video-codec-sdk.

[61] NVIDIA. NVIDIA GeForce NOW, 2024. https://www.nvidia.com/en-us/geforce-now/.

[62] NVIDIA DOCS HUB. Single Root IO Virtualization (SR-IOV), 2024. https://docs.nvidia.com/doca/sdk/single+root+io+virtualization+(sr-iov)/index.html.

[63] Open-Stream Developers. Open-Stream Game Streaming Platform, 2024. https://open-stream.net/.

[64] Mikko Pitkänen, Marko Viitanen, Alexandre Mercat, and Jarno Vanne. Remote VR Gaming on Mobile Devices. In *Proc. of ACM MM*, pages 2191–2193, 2019.

[65] Qemu Developers. Documentation of Virtio-Gpu, 2023. https://www.qemu.org/docs/master/system/devices/virtio-gpu.html.

[66] Jiaxing Qiu, Zijie Zhou, Yang Li, Zhenhua Li, Feng Qian, Hao Lin, Di Gao, Haitao Su, Xin Miao, Yunhao Liu, and Tianyin Xu. vSoC: Efficient Virtual System-on-Chip on Heterogeneous Hardware. In *Proc. of ACM SOSP*, 2024.

[67] Sai Swaroop Ratakonda and Sreela Sasi. Seasonal Trend Analysis on Multi-Variate Time Series Data. In *Proc of IEEE ICDSE*, pages 1–6, 2018.

[68] Eric Risser. Rendering 3D Volumes Using Per-pixel Displacement Mapping. In *Proc. of ACM SIGGRAPH Symposium on Video Games*, pages 81–87, 2007.

[69] Rusty Russell. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42:95–103, 2008.

[70] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, and Bruce Maggs. DChannel: Accelerating Mobile Applications With Parallel High-Bandwidth and Low-Latency Channels. In *Proc. of USENIX NSDI*, pages 419–436, 2023.

[71] Alexander Shapiro. Monte Carlo Sampling Methods. In *Handbooks in Operations Research and Management Science*, volume 10 of *Stochastic Programming*, pages 353–425. Elsevier, January 2003.

[72] Ryan Shea, Di Fu, and Jiangchuan Liu. Cloud Gaming: Understanding the Support From Advanced Virtualization and Hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 25:2026–2037, 2015.

[73] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. Using Graphics Rendering Contexts to Enhance the Real-Time Video Coding for Mobile Cloud Gaming. In *Proc. of ACM MM*, pages 103–112, 2011.

[74] W. C. Siu. *Multimedia Information Retrieval and Management: Technological Fundamentals and Applications*. Springer Science & Business Media, January 2003.

[75] Sony. PlayStation Plus Cloud Gaming, 2024. https://www.playstation.com/en-us/games/playstation-plus-essential-1-month-subscription.

[76] Dongjie Tang, Cathy Bao, Yong Yao, Chao Xie, Qiming Shi, Marc Mao, Randy Xu, Linsheng Li, Mohammad R. Haghighat, Zhengwei Qi, and Haibing Guan. CARE: Cloudified Android OSes on the Cloud Rendering. In *Proc. of ACM MM*, pages 4582–4590, 2021.

[77] Dongjie Tang, Yun Wang, Linsheng Li, Jiacheng Ma, Xue Liu, Zhengwei Qi, and Haibing Guan. gRemote: API-Forwarding Powered Cloud Rendering. In *Proc. of ACM HPDC*, pages 197–201, 2020.

[78] Tencent. Tencent Pioneer, 2024. https://gamer.qq.com/.

[79] Tencent. Tencent Start, 2024. https://start.qq.com.

[80] Tencent Cloud. Tencent Cloud Gaming Product Documentation, 2023. https://cloud.tencent.com/document/product/1162/46097.

[81] Unity Technologies. Unity Frame Pacing for Android, 2024. https://docs.unity3d.com/ScriptReference/Application-targetFrameRate.html.

[82] Unity Technologies. Unity Official Website, 2024. https://unity.com.

[83] Alexander Van't Hof, Hani Jamjoom, Jason Nieh, and Dan Williams. Flux: Multi-Surface Computing in Android. In *Proc. of ACM EuroSys*, pages 1–17, 2015.

[84] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, Changxi Zheng, Jing Wang, and Honghao Liu. Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming. In *Proc. of USENIX NSDI*, pages 113–129, 2024.

[85] Shanika L. Wickramasuriya, George Athanasopoulos, and Rob J. Hyndman. Optimal Forecast Reconciliation for Hierarchical and Grouped Time Series Through Trace Minimization. *Journal of the American Statistical Association*, 114:804–819, 2019.

[86] Jiangkai Wu, Yu Guan, Qi Mao, Yong Cui, Zongming Guo, and Xinggong Zhang. ZGaming: Zero-Latency 3D Cloud Gaming by Image Prediction. In *Proc. of ACM SIGCOMM*, pages 710–723, 2023.

[87] XPosed Developers. Xposed Framework API, 2023. https://api.xposed.info/reference/packages.html.

[88] Lingfeng Xu, Xun Guo, Yan Lu, Shipeng Li, Oscar C. Au, and Lu Fang. A Low Latency Cloud Gaming System Using Edge Preserved Image Homography. In *Proc. of IEEE ICME*, pages 1–6, 2014.

[89] Qiang Xu, Sanjeev Mehrotra, Zhuoqing Mao, and Jin Li. PROTEUS: Network Performance Forecast for Real-Time, Interactive Mobile Applications. In *Proc. of ACM MobiSys*, pages 347–360, 2013.

[90] Jinyu Yang, Mingqi Gao, Zhe Li, Shang Gao, Fangjing Wang, and Feng Zheng. Track Anything: Segment Anything Meets Videos. *arXiv.2304.11968*, 2023.

[91] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. Mobile Gaming on Personal Computers with Direct Android Emulation. In *Proc. of ACM MobiCom*, pages 1–15, 2019.

[92] Zongxin Yang and Yi Yang. Decoupling Features in Hierarchical Propagation for Video Object Segmentation. *Advances in Neural Information Processing Systems*, 35:36324–36336, 2022.

[93] Roy D. Yates, Mehrnaz Tavan, Yi Hu, and Dipankar Raychaudhuri. Timely Cloud Gaming. In *Proc. of IEEE INFOCOM*, pages 1–9, 2017.

[94] Yuhan Zhou, Tingfeng Wang, Liying Wang, Nian Wen, Rui Han, Jing Wang, Chenglei Wu, Jiafeng Chen, Longwei Jiang, Shibo Wang, Honghao Liu, and Chenren Xu. AUGUR: Practical Mobile Multipath Transport Service for Low Tail Latency in Real-Time Streaming. In *Proc. of USENIX NSDI*, pages 1901–1916, 2024.

Region-based Content Enhancement for Efficient Video Analytics at the Edge

Weijun Wang[1*] Liang Mi[2*†] Shaowei Cen[2†] Haipeng Dai[2] Yuanchun Li[1] Xiaoming Fu[3] Yunxin Liu[1‡]
[1]*Institute for AI Industry Research (AIR), Tsinghua University*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University*
[3]*University of Göttingen*

Abstract

Video analytics is widespread in various applications serving our society. Recent advances of content enhancement in video analytics offer significant benefits for the bandwidth saving and accuracy improvement. However, existing content-enhanced video analytics systems are excessively computationally expensive and provide extremely low throughput. In this paper, we present region-based content enhancement, that enhances only the important regions in videos, to improve analytical accuracy. Our system, RegenHance, enables high-accuracy and high-throughput video analytics at the edge by 1) a macroblock-based region importance predictor that identifies the important regions fast and precisely, 2) a region-aware enhancer that stitches sparsely distributed regions into dense tensors and enhances them efficiently, and 3) a profile-based execution planer that allocates appropriate resources for enhancement and analytics components. We prototype RegenHance on five heterogeneous edge devices. Experiments on two analytical tasks reveal that region-based enhancement improves the overall accuracy of 10-19% and achieves 2-3× throughput compared to the state-of-the-art frame-based enhancement methods.

1 Introduction

Video cameras are widespread in our society, with numerous installations in major cities and organizations [3, 4, 12, 13, 16]. These cameras continuously collect vast amounts of video data for various applications such as traffic control [25], school security [42], crime investigation [15, 83], sports refereeing [5, 14], and more [17, 86]. Significant advances in deep neural networks (DNNs) for vision processing offer a tremendous opportunity to AI-powered automatic *video analytics* [44,45,56,89,92,101]. A typical video analytics pipeline follows the distributed paradigm as AI methods often require high computing power that cameras cannot offer [1, 18]. Cameras capture and deliver the live videos to the edge in proximity for real-time processing. However, the outdated hardware of cameras in use [62] and extremely limited uplink bandwidth between the camera and the edge [21, 43, 101] results in limited video quality and thus low analytical accuracy.

Content enhancement offers a promising solution to tackle this issue. It delivers a remarkable accuracy improvement [68,70,80,82,84,96,100] and bandwidth saving [41,60,79,93–95,97] by leveraging neural enhancement models (*e.g.*, super-resolution [24, 64], generative adversarial network [30, 48], image restoration [63]) to enhance the informative details of video frames prior to feeding them into the final analytical models. Different from previous model optimization methods (*e.g.*, model merging [55, 71], model updating [60, 74], and model switching [59, 95]), data-optimization content enhancement does not require modifying user-provide models [23, 31, 73]. Besides, even if the city government updates current low-end cameras to the latest ones that can offer high-quality video, content enhancement can still improve the details of small objects or blurred content.

Unfortunately, naively employing content enhancement in practice is excessively computationally expensive. Such straightforward ways of enhancements not only cause high latency but also compete for computational resources with final analytical models. For example, applying enhancement on one tail-accuracy frame or executing generative adversarial networks on hard-recognized human faces causes hundreds to thousands of milliseconds of latency [84, 96]. The state-of-the-art method, *selective enhancement* [93, 95], yields some throughput improvement by enhancing only sampled frames but results in substantial accuracy reduction (more in § 2.2).

Goal and observations. This paper asks the following research question: Can we integrate content enhancement into video analytics at the edge in a high-throughput manner? We argue that yes, but it requires a rethink of enhancement. Our key observations are that 1) the time cost of enhancement positively correlates to the input size, and 2) the regions after enhancement that benefit analytical accuracy only occupy a small portion in each frame (more details in §2.3). Therefore, we aim to develop a *region-based content-enhancement* method which only enhances the beneficial regions.

To this goal, we face three non-trivial challenges.

(C1) How to identify beneficial regions fast and accurately? Simply relying on DNN-based identification is too slow to the extent that it even exceeds the time required for the entire frame enhancement.

(C2) How to efficiently enhance regions? Characteristics of enhancement and heterogeneous accuracy gain of regions necessitate a novel enhancement mechanism that differs from the previous frame-based enhancement [60, 93–95].

*Weijun Wang and Liang Mi contributed equally to this work.
†This work was done while Liang Mi and Shaowei Cen were interns at the Institute for AI Industry Research (AIR), Tsinghua University.
‡Corresponding author: Yunxin Liu<liuyunxin@air.tsinghua.edu.cn>.

(C3) How to allocate resources among components? To maximize the overall performance, limited resources on edge must be best allocated among system components; however, typical schedulers cause imbalances among components, which in turn leads to noticeable throughput degradation.

RegenHance. This paper presents RegenHance, a system that efficiently identifies and enhances the most beneficial regions in video, enabling high-accuracy and high-throughput analytics at the edge. RegenHance addresses the above challenges by following techniques:

Macroblock-based region importance prediction. We propose a predictor to fast and accurately identify the regions that yield the highest accuracy improvements in videos (§3.2). We first analyze the advantages of predicting the region importance at the macroblock (MB, the video encoding unit) level and establish a metric to precisely measure the accuracy gain (importance) of each MB after enhancement in each frame. Next, we develop an ultra-lightweight prediction model and a prediction results reuse algorithm, achieving high throughput. Experiments show our method can predict the MB importance in 30 frames per second on a single CPU thread.

Region-aware enhancement. We design an enhancer that efficiently enhances the Tetris-like irregular regions composed of important macroblocks (§3.3). It prioritizes and selects the Top-K MBs of all video streams in order of their importance scores (K is estimated by §3.4). Then, considering the unique characteristic of the time cost of enhancement (§2.3) and the sparse distribution of important MBs (§3.2), the region-aware enhancer can be formulated as a two-dimensional bin-packing problem that minimizes the input size of the enhancement model to optimize the throughput. We propose a greedy algorithm that fast stitches the irregular regions into dense tensors before forwarding them into the enhancement model.

Profile-based execution planning. To balance the limited resources among components on the edge device, we propose an execution planer (§3.4), that profiles the capacity of the given edge device, and allocates resources among components by determining the parameters (*e.g.*, batch size of the input) for each component (*e.g.*, decoder, MB-based region importance prediction, region-aware enhancer, and analytical models) to maximize the end-to-end throughput.

We summarize our key contributions as follows:

- To the best of our knowledge, we are the first to identify the bottleneck of content-enhanced video analytics and propose a new idea: region-based content enhancement.
- We prototype RegenHance that enables region-based content enhancement video analytics at the edge. It involves three components, MB-level region importance prediction, region-aware enhancement, and profile-based execution planning.
- We implement RegenHance and conduct evaluations for two popular analytical tasks with real-world videos on five heterogeneous devices. Experimental results show that RegenHance improves 10-19% accuracy and achieves 2-3× throughput compared to the state-of-the-art methods. Our code is now available at https://github.com/mi150/RegenHance.

This work does not raise any ethical issues.

2 Motivation and Challenges

This section explores three questions: (1) How well are selective enhancement techniques in video analytics (§2.2)? (2) How much potential improvement can region-based content enhancement achieve (§2.3)? (3) What challenges must be tackled (§2.4)?

2.1 Background

Typical video analytics platforms [6, 9, 39], to support real-time applications, provide services on edge servers close to cameras that allow users to register custom video analytics jobs. To register a job, users specify video sources, analytical tasks, performance targets, and optionally upload their analytical models [23, 31, 73]. The camera captures and encodes multiple images into a video stream (*e.g.*, 30 images into a 1-second video chunk set up in prior studies [44, 62, 101]), and then transmits to the edge for analytics. Following the same scope, this paper also concentrates on optimizing the analytics platforms on edge servers.

Content-enhanced video analytics [41, 60, 68, 85, 93–97, 100] is a mainstream data optimization method, that utilizes DNN models to enhance low-resolution frames to high-definition ones. Pre-trained DNNs learn a pixel generator mapping neighboring pixels' values to the generated one from training data, then deployed in runtime systems analyzing real-world videos for accuracy improvement and bandwidth saving.

Selective enhancement is the state-of-the-art method to improve the throughput of content-enhanced Internet video streaming [93, 95, 104]. In this method, a few sampled frames (called anchors) are enhanced by the super-resolution model, while the remaining frames are quickly up-scaled by reusing enhanced ones via codec information. Such reuse causes video quality loss, due to the rate-distortion problem in codec [78], accumulated across consecutive non-enhanced frames, and thus the sampled frames need to be selected carefully.

2.2 Limitation of Frame-based Enhancement

Motivational study. To measure the performance of selective enhancement, we benchmark three analytics methods. Taking object detection as an example, 1) *Only infer* method purely detects objects in each frame, 2) *Per-frame SR* method applies super-resolution on each frame and then detects objects on them (as the ground truth). 3) *Selective SR* [95] applies super-resolution on a set of selected frames and reuses on other ones, then detects objects on all frames; this method will select enough frames until meeting preset target accuracy (*e.g.*, 90%). For fairness, all methods run on an edge

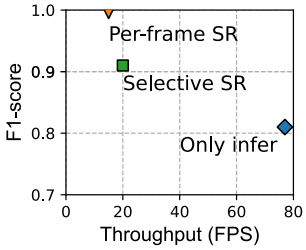


Figure 1: Performance of the frame-based methods. The state-of-the-art selective SR provides poor E2E throughput.

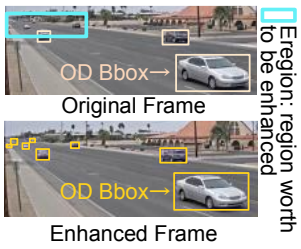


Figure 2: An example eregion worth to be enhanced for object detection (OD) bounding with a rectangle box.

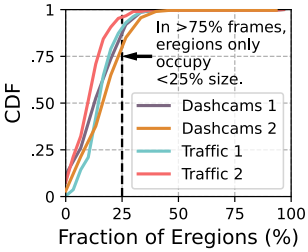


Figure 3: Distribution of eregions in various videos. In large amounts of frames, eregions occupy only a small portion.

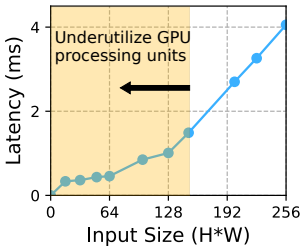


Figure 4: Latency of enhancement. The same H*W (*e.g.*, 64*64) input, no matter pixel values, yields the same latency.

device equipped with an NVIDIA T4 GPU [10] that upscales a 640 × 360 video to a 1920 × 1080 version via EDSR model [64] and detects objects with YOLO [46]. Both models compile with TensorRT [11].

Result: selective enhancement is too heavy for video analytics. As shown in Fig. 1, per-frame SR improves >10% accuracy but reduces >76% end-to-end throughput than only infer. Normalized with per-frame SR, selective SR indeed improves throughput from 15 to 20 (>33%) fps, but still far from the throughput of only infer and results in an undeniable accuracy drop. This is because, compared to human perception, analytical models are more sensitive to blur and distortion caused by accumulated loss of reuse. Small changes in several pixel values may flip the analytics result inversely [29]. So when presetting the accuracy target as 90%, selective SR methods choose 27-61 anchor frames in each 120-frame chunk on average. Such 24-51% fraction is much higher than that select and enhance only 2-13% frames in video applications for human vision.

2.3 Potential Improvement of Region-based Content Enhancement

Above *frame-based* optimizations fail to offer efficient analytics because they evenly enhance every pixel in each frame. We argue that this is unnecessary and causes a significant waste of computational resources.

Our first observation is that *the regions that are worth being enhanced in each frame occupy only a small portion.* In this paper, we name the region that provides higher analytical accuracy after enhancement, *Eregion*. As visualized in Fig. 2, the frame after enhancement (bottom) can provide more detected objects, and the eregion (blue rectangle) is a simple region (discussed more in Appx. C.1). Fig. 3 analyzes the distribution of eregions in experimental videos for Fig. 1. In >75% of frames over videos recording various scenarios, eregions for object detection task only occupy 10-25% of the spatial area of a frame; while for semantic segmentation, as illustrated in Appx. C.1, only 10-15% area in 70% frames is eregions. With careful design, eregions could minimize the enhanced area while offering comparable accuracy to per-frame SR. So ideally, only eregions should be enhanced in video analytics.

The second observation is that *the time cost of enhancement DNNs is positively correlated with input size.* As shown in Fig. 4, along with the enlarging input size, the enhancement latency initially experiences a gradual rise and then scales proportionally with the input size after making full use of processing units. This unique characteristic originates from the nature of enhancement models that generate new pixels from neighboring ones with the learned mapping. Thus, we must minimize the size to decrease enhancement latency.

Our idea. With the above two observations in mind, we propose *region-based content enhancement* that aims to only enhance the eregions to maximize the throughput of content enhancement while providing comparable accuracy to per-frame enhancement. The question next is how to fully explore the potential room for improvement.

2.4 Challenges to Achieve Region-based Content Enhancement

Building an effective system based on region-based content enhancement involves three challenges.

C1: How to fast and accurately identify eregions on original frames? Enhancing only the eregions, as shown in Fig. 5, does save significant (2.4×) time cost. Unfortunately, such an oracle cannot be reached as eregions are calculated from the already-enhanced frames. During real-time analytics, only the original frames are accessible, thus we have to predict eregions on the original frames. Naively using a DNN-based method to identify regions, *e.g.*, DDS [44] identifies the Region of Interest (RoI) with a Region Proposal Network (RPN), can not meet our requirements. The imprecise identification spends too much time on the enhancement of unimportant regions in Fig. 5, and the high computing cost of the selection methods themselves (*e.g.*, the RPN), in turn, harms the possible time saving of region-based enhancement.

C2: How to enhance eregions in high throughput? High-throughput region-based enhancement is hardly achieved by existing frame-based enhancers [84, 93–97]. Three new contradictions that frame-based methods have never encountered

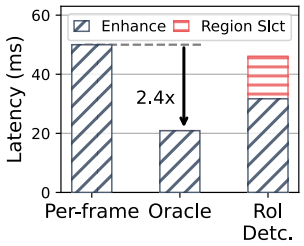
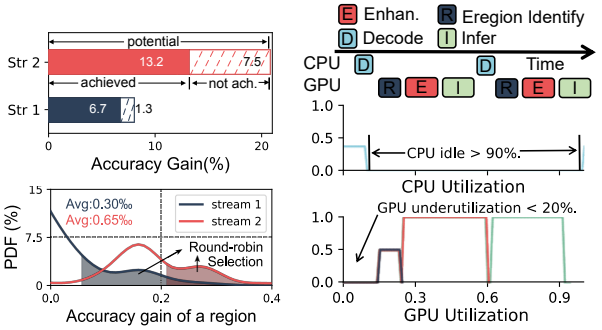


Figure 5: Region-based enhancement saves remarkable (2.4×) latency, but prior region selection methods themselves (*e.g.*, RoI detection [44]) produces too high computing cost.



(a) Round robin across streams (b) Sequential exec. of components

Figure 6: Limitation of region-agnostic resource management.

obstruct region-based enhancement. 1) Eregions are often in irregular size and sparsely distributed in each frame, but the enhancement models (DNNs) only accept rectangle (matrix) input. 2) One natural option, as DDS [44] used, is setting non-eregion pixels to black (zero value in the matrix); and yet it leads to the same latency as enhancing the original frame due to the pixel-value-agnostic characteristic of enhancement latency as demonstrated in Fig. 4. 3) Batch execution of enhancement DNNs benefits higher throughput [76], but requires every input matrix in the same size.

C3: How to allocate resources among components? To optimize the overall performance (accuracy and throughput), particularly for all video streams, 1) eregions must be carefully allocated across streams, and 2) computational resources must be balanced among computing components. However, this is challenging due to the heterogeneity of regions and components. First, the accuracy gains and enhancement overhead of regions are heterogeneous. Second, all runtime components on the edge server, including decoding, eregion identification, enhancement, and analytical inference, compete for computational resources, and their throughputs are heterogeneous. Using previous schedulers [8, 95] causes noticeable degradation in accuracy and throughput.

To demonstrate this, we benchmark a strawman scheduler that 1) parallelizes per-stream decoding on multiple CPU threads, 2) forwards decoded frames to GPU in a round-robin manner, 3) applies RegenHance's region enhancement strategy but pipelines computing components at batch size of four. Two streams that contain different sizes of eregions are delivered to an edge server equipped with an NVIDIA T4 GPU [10]. We measure the accuracy gain per stream and the resource utilization of computing units.

This strawman region-agnostic scheduler fails to achieve high accuracy gain and throughput. Fig. 6(a) (top) shows the potential (the accuracy of per-frame SR minus only infer), achieved (solid part), and not achieved (dotted part) accuracy gain of two streams. There is a remarkable gain (7.5%) not achieved in Stream 2 as the round-robin manner results in an even chance for enhancement across streams. The distributions of the region's accuracy gain in each stream are highly heterogeneous as the two curves shown in Fig. 6(a) (bottom). Suppose a scheduler enables Stream 2 to enhance more regions that provide higher gain (namely, in Fig. 6(a) (bottom), decrease some grey area under Stream 1 to fill the blank under Stream 2 into red), it can improve the overall accuracy. On the other hand, inappropriate execution plans underutilize the hardware, leading to low end-to-end throughput and, in turn, hindering higher accuracy improvement. As illustrated in Fig. 6(b), the region-agnostic scheduler leaves >90% CPU and >15% GPU idle time. The unsuitable batch size results in the Eregion Identification utilizing only <50% computing resource GPU, hence blocking the subsequent enhancement and analytics.

3 RegenHance Design

RegenHance aims to enhance only the regions (Eregions) that best benefit analytical accuracy, so as to maximize the overall accuracy gain among all streams in the optimal end-to-end throughput on a given edge server. To do so, it addresses the above challenges by three techniques: 1) a lightweight but precise algorithm to identify eregions on macroblock (MB) granularity, 2) a high-throughput enhancer that together considers the heterogeneous accuracy gain across streams and unique characteristics of enhancement, and 3) a holistic resource manager generating execution plan for all components on given edge servers. Fig. 7 depicts the design of RegenHance. We implement the techniques with three new components: MB-based region Importance Prediction, Region-aware Enhancement, and Profile-based Execution Planning.

3.1 Overview Workflow

RegenHance takes the original ingest videos and outputs high-resolution frames for downstream analytical tasks. During the offline phase, Profile-based Execution Planning ❶ profiles the budget of processors on the given edge server with the registered analytical tasks and optional uploaded corresponding models by users; next, ❷ generates the execution plan, *i.e.*, specifies every runtime component's execution hardware and their allocated resource meeting best pipelining and parallelization, to maximize the end-to-end throughput under the

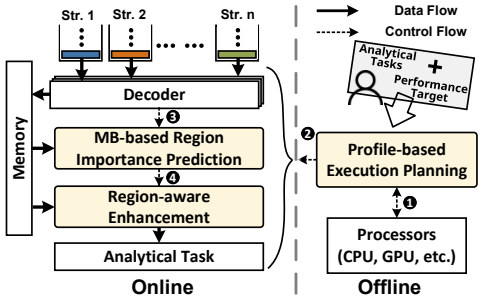


Figure 7: RegenHance overview.

constraints of performance targets (*e.g.*, accuracy and latency) that users specify (§3.4).

During the online phase, the decoder decodes compressed video streams in parallel to RGB frames and stores them in memory. MB-based region importance prediction ❸ selects frames during decoding, loads them from memory, predicts their MBs importance, and appends importance into MBs' indexes (§3.2). Region-aware enhancement ❹ aggregates and sorts all MBs over all streams based on their importance, constructs top N ones into regions, stitches them into dense tensors, and efficiently executes enhancement (§3.3). Lastly, forward the enhanced frames to analytical models for inference.

3.2 MB-based Region Importance Prediction

In order to identify eregions quickly and accurately, we propose an MB-based region importance prediction. It contains a lightweight predictor estimating MB importance in each frame (in spatial dimension) and an MB importance reuse algorithm among continuous frames (in temporal dimension).

3.2.1 Spatial MB Importance Predictor in Each Frame. Eregions can be arbitrary shapes, we need to determine the granularity to construct these regions first. One natural way is pixel granularity. The pixel-grain method, of course, provides the most precise importance prediction but costs too much computing resources (as demonstrated in Fig. 5).

Inspired by the video codec knowledge [45, 53], we argue that setting macroblock (MB) as the elementary unit of eregions is both efficient and accurate. Macroblock serves as the elementary unit for applying the quantization parameter (QP) to control the compression level of video quality. For example, in H.264 [7], frames are divided into an array of 16×16-pixel MBs, and each MB is assigned with different QPs to distribute more bits to regions demanding higher visual quality and fewer bits to less crucial regions.

After setting MB as a unit, our problem transformed to: Given a video frame f containing a set of macroblocks MB, select partial MBs MB_s to be enhanced such that the accuracy of downstream analytical task is maximized:

$$max \quad Acc(I(SR(MB_s)+IN(\overline{MB_s})), I(SR(f))),$$

where $Acc(\cdot)$ is the accuracy of analytical task $I(\cdot)$ (*e.g.*, F1-score of object detection), $SR(\cdot)$ and $IN(\cdot)$ are the super-resolution and the bi-linear interpolation with the same enlarge factor, $\overline{MB_s}$ indicates the unselected MBs.

To select beneficial MBs, we need to quantify their importance. Ideally, MBs after enhancement leading to larger variations in inference results and greater differences in pixel values are more important. With this intuition, we utilize the following "importance" metric looking at the gradient of the accuracy with respect to changes in the pixels in each MB, and the magnitude of change in those pixel values due to enhancement.

$$\sum_{i \in MB} || \underbrace{\frac{\partial Acc(I(IN(f)), I(SR(f)))}{\partial IN(f)}\Big|_i}_{\text{accuracy's gradients at pixel } i} ||_1 \cdot || \underbrace{(SR(f)_i - IN(f)_i)}_{\text{pixel value distance at } i} ||_1,$$

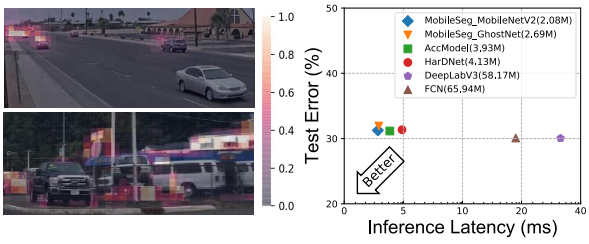
where i is a pixel within the certain MB and $||\cdot||_1$ is the L1-norm. This metric assigns an importance score to each MB. Fig. 8(a) shows the heat map of MB importance on the same frame with Fig. 2. Their similarity demonstrates the MB importance is a good representation of eregions*.

It seems straightforward to calculate the MB importance according to the above importance metric. Unfortunately, it needs the frames already enhanced. Such a chicken-egg paradox makes us predict the MB importance in the original frames with a learning-based method. We construct the training set by enhancing all video frames and calculating the importance metrics on each of them with one forward and backward propagation of the final analytical model. This resulting importance value of each MB is one item in the ground truth matrix of each frame, Mask*.

Next, we consider this problem a segmentation task and draw inspiration from their model designs. Our key observation is that the MB importance prediction resembles the problem of image segmentation. Image segmentation aims to semantically segment an image by assigning each pixel one predefined label, while MB importance prediction gives each MB an importance score. With this in mind, the MB importance prediction can be approximated as an image segmentation problem by boiling the importance value down to multiple importance levels. Suppose setting ten levels in this paper, MB importance prediction targets to assign each MB an importance level, like classification in image segmentation. Appx. B demonstrates such approximation yields good performance. This observation enables us to harness various techniques tailored for MB importance prediction, including learning-based semantic segmentation.

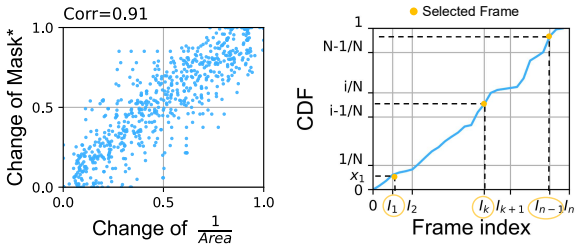
Aiming to predict MB importance precisely at high throughput, we train an ultra-lightweight segmentation model with the above importance metric. We retrained six models using

*Unlike the saliency map in computer vision that captures which pixel values have more influence on the DNN output, our loss function captures how the enhancement or non-enhancement of an MB (its content quality change) specifically changes the DNN inference accuracy.



(a) Heat map of MB importance (b) Model selection

Figure 8: Macroblock-level region importance prediction.



(a) Correlation ratio (b) Frame selection with CDF

Figure 9: Temporal MB importance reuse across frames.

the cross-entropy loss with piecewise Mask* (as importance level) to support MB importance prediction. Six models are an ultra-lightweight model MobileSeg [81] with two backbones, two lightweight models AccModel [45] and HarDNet [32], and two heavyweight models FCN [66] and DeepLabV3 [33]. As shown in Fig. 8(b), ultra-lightweight models provide almost the same accuracy as heavyweight ones while offering 4-18× throughput. This can be attributed to the significantly relaxed complexity of the MB-grained segmentation compared to traditional image segmentation. Predefined MB size in video codec supports this point. The 16×16-MB H.264 [7] codec makes 1920×1080 labels that output in traditional image segmentation models decrease to 120×68 ones in MB-grained. As the best performance of MobileSeg, we select it as our MB importance predictor. At the offline phase, RegenHance fine-tunes the predictor with the Mask* generated by user-uploaded analytical models.

3.2.2 Temporal MB Importance Reuse among Continuous Frames. Reusing DNN outputs on similar frames is popular [45, 90, 100] as discussed in §5. Reusing content of enhanced frames in region-based enhancement like selective SR will cause dramatic accuracy loss (demonstrated in §2.2), but the MB importance value is reusable. Hence, RegenHance predicts the MB importance of a set of frames and reuses their outputs on other frames, to offer the best approximation of per-frame MB importance prediction.

Our key choice is using an ultra-lightweight operator to represent the MB importance change then only predicts MB importance of the frames with large changes. We compared many lightweight features and proposed $\frac{1}{Area}$ operator (details in Appx. C.2). Area operator captures the large blocks in images [52, 62], whereas $\frac{1}{Area}$ captures the change of small objects just as the important MBs in Fig. 8(a) needed. Statistical analysis in Fig. 9(a) shows it is a nice representation to estimate the change of Mask* (with 0.91 correlation).

With the $\frac{1}{Area}$ operator, denoted by Φ, RegenHance selects frames to be predicted by the cumulative distribution function (CDF) of $\Delta\Phi$ between continuous frames in each chunk. It first accumulates the feature change with the following function during decoding a n-frame chunk.

$$\mathbf{S} = Norm(\Delta\Phi(Res_{Y_1}), \cdots, \Delta\Phi(Res_{Y_{n-1}})),$$

where Res_{Y_i} is Y-channel of each frame's residual[†], $\Delta\Phi(Res_{Y_i}) = \Phi(Res_{Y_{i+1}}) - \Phi(Res_{Y_i})$, $Norm(\cdot)$ is L1-normalization. Then, it selects N frames based on the CDF $\mathbf{M}$, where $\sum_i M_i = 1$, calculated from $\mathbf{S}$. As illustrated in Fig. 9(b), the y-axis is divided into N even intervals; and in each interval, it selects a value, *e.g.*, x_1, and then its corresponding frame index, *e.g.*, I_1, on the x-axis is a selected frame. The other frames in each interval reuse the predicted result of this frame. In multiple streams, the number of selected frames for given stream j is allocated by the ratio $\frac{\sum_i \Delta\Phi_{i,j}}{\sum_j \sum_i \Delta\Phi_{i,j}}$, and its total number is determined by the profile-based execution planning (§3.4). Fig 10 depicts the MB-based importance prediction first selects frames and then predicts their MB importance.

3.2.3 Discussion. *Generality of importance metric.* In this paper, each analytical task must retrain a specific MobileSeg for importance prediction, as the metric equation relies on the downstream models. We omit a general importance metric design in this paper because 1) the current specific metric offers good results (as shown in Fig. 8(a)) and 2) the fine-tuning time only costs 4 minutes on eight RTX3090 GPUs. We will explore a general metric in future work.

3.3 Region-aware Enhancement

To achieve optimal overall accuracy over all streams and end-to-end throughput, we propose a region-aware enhancement consisting of a cross-stream MB selection and a region-aware bin packing algorithm.

3.3.1 Cross-stream MB Selection. To maximize the overall accuracy improvement, RegenHance chooses the best MB st among all streams that offer the highest total accuracy gain. As shown in Fig. 10, it constructs a global queue that aggregates and sorts MBs from all streams in order of the importance (level) in the MB index $\{stream_{id}, frame_{id}, loc_x, loc_y, importance\}$, where loc_x, loc_y indicates coordination of the MB in the frame.

Next, it selects the top N MBs and delivers their indexes to the region-aware bin packing algorithm for further processing. The number of selected MBs is estimated as follows,

$$\max_N MB_{size} \cdot N \leq H \cdot W \cdot B,$$

[†]Residual is data existing in video codec representing the difference data between the original and compressed video frames.

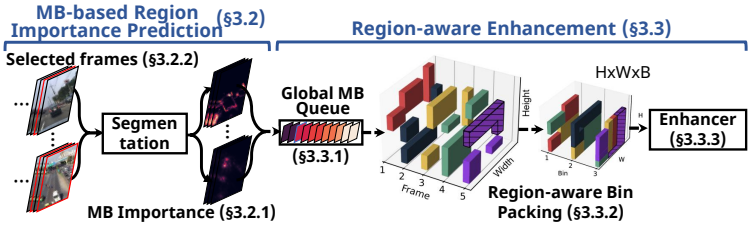


Figure 10: Runtime components of RegenHance.

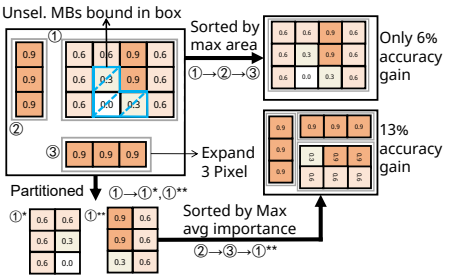


Figure 11: Large region is not always desirable. Classic large-item-first policy packs the MBs that are unselected but bounded in the large box into the bin (upper half), leading to a 7% accuracy drop compared to our importance-first policy (lower half).

Algorithm 1 Region-aware Bin Packing

Input: Indexes of MBs, B Bins of $H \times W$ size
Output: Packing plan of MBs

```
1:  function PACKING(MBs, Bins)
2:      freeareas = Bins                    ▷ Initialize the list of free area as Bins
3:      regions = REGIONPROPS(MBs)
4:      boxes = BOUND(regions)
5:      boxes = PARTITION(boxes)            ▷ Partition big boxes to small ones
6:      SORT(boxes, reverse = True, order = Σ_{MB∈box} MB.importance / |{MB|MB∈box}|)
7:      for box in boxes do
8:          for farea in freeareas do
9:              if ROTATEPACKING(box, farea) then
10:                 UPDATE(farea, freeareas); break
11:         boxes.POP(box)
12: function ROTATEPACKING(box, farea)
13:     if farea.w≥box.w and farea.h≥box.h then return True
14:     else if farea.w≥box.h and farea.h≥box.w then return True
15:     else return False
16: function UPDATE(farea, freeareas, box)
17:     box.APPEND(loc)                     ▷ Append the placed location of box in Bins
18:     ifareas = INNERFREE(farea, box)     ▷ Find rest free areas in box after
                                              packing farea (more in Appx. A)
19:     freeareas = freeareas ∪ ifareas \ {farea}
```

where MB_{size} is the size of a MB, *e.g.*, 16×16 in H.264. H, W, B are the optimal height, width, and batch size for the enhancement model preset in the execution plan (§3.4).

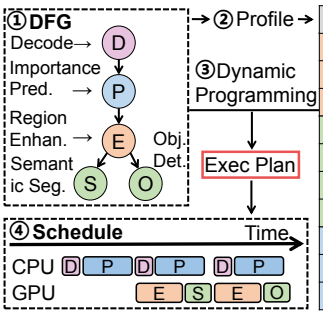
3.3.2 Region-aware Bin Packing. Considering the sparse distribution of selected MBs, the unique characteristics of enhancement models (requires rectangle input and latency is pixel-value-agnostic proportional to the input size), and the helpful but complex batch execution, we propose a region-aware bin packing algorithm to construct selected MBs into irregular regions and stitch them into dense tensors.

The problem is formulated as a *two-dimensional bin-packing problem* [28] that packs the maximum number of selected MBs into given bins. The input are MB indexes and the number of bins B and their size $H \times W$, the output is the packing plan of MBs. We process MB indexes, not real images, to avoid frequent memory I/O. It is known to be NP-hard. Prior methods [37, 38, 76] batch the standard rectangle DNN inputs and thus can not handle irregular regions. To motivate our design, we first analyze two strawmen: 1) MB packing: directly packing selected MBs after expanding three pixels in each direction[‡] into bins, and 2) Irregular region packing: packing the (irregular) regions consisting of connected and selected MBs after extension into bins [67]. MB packing packs too many unimportant or repeated pixels due to extension, thus under-utilizing given bins compared to the irregular manner; whereas the high complexity of existing irregular region packing algorithms causes more than one order of magnitude of time cost than MB packing (more in Appx. C.4).

To strike a better balance between bin utilization and algorithm efficiency, we propose region-aware bin packing in Alg. 1. Its key design choices are: 1) bounding the irregular regions in rectangle boxes for efficient packing plan search (line #3-5); 2) cutting large boxes into small ones and sorting boxes in order of importance density, *i.e.*, the average importance of all MBs in it, for high bin utilization (line #6). Fig. 11 exemplifies that this prioritization leads to a higher (13%) accuracy gain, compared to 6% of traditional large-item-first (max-area-first) policies [57] (more results refer to Fig. 23). Region-aware bin packing first constructs regions by calculating the connected components of selected MB in line #3, and bounds each region in a rectangular box with extension (line #4), *e.g.*, ①②③ in Fig. 11. Next, it partitions the boxes larger than the preset size (line #5) in case of importing too many unimportant MBs, *i.e.*, box ① cut to ①* and ①**. Then, line #6 sorts boxes according to the priority we propose, *i.e.*, not (①,②,③) of max-area-first policy but (②,③,①**). Lastly, it iteratively packs them into bins. In each iteration (line #7-11), it rotates and packs a box into a free area in a bin, updates the box's placing location and the list of free areas, and then deletes the box from the boxes. For example, the "L" region in Fig. 10 and ①** in Fig. 11 is rotated and packed into bins.

3.3.3 Enhancement with Super-resolution Model. Based on the execution plan across various devices (§4.2), except for Jetson AGX Orin equipped with unified memory, the other devices need to transfer real frames from main memory to GPU memory. To save time, we hide this transfer at the same time as the MB selection and bin packing procedure. This idea works because, after processing the real frames by MB-

[‡]Such expansion can avoid the MB/region boundaries causing too many jagged edges and blocky artifacts [45] when pasting enhanced content back to the bi-linear-interpolated frames. For more results refer to Appendix C.3.



Model@HW	Bat	Cos	TPS
SR@CPU	1	270	3
SR@GPU	1	7	142
	4	27	148
OD@GPU	1	2	500
	4	7.8	512
SS@GPU	1	2.7	370
	4	8.3	481
P@CPU	16	27	592
P@GPU	1	1.7	588
	4	4.2	952

Figure 12: An example of execution planning process seven 30-fps streams s.t. <1s latency & >0.9 accuracy performance target. The throughputs (TPS) of one model executing on one hardware (Model@HW) in the right table are profiled with batch size (Bat) and time cost (Cos). The red boxes highlight the execution plan of each component of the given dataflow graph (DFG).

based region importance prediction, as shown in Fig. 7, all modules solely deal with MB indexes before super-resolution. Consequently, we stitch the real-content regions into tensors (bins) following the packing plan on the GPU, subsequently enhancing and stitching them back to bi-linear-interpolated non-regions for final analytics.

3.4 Profile-based Execution Planning

Given an edge server equipped with R computational resource (*i.e.*, R processing time of processors under 100% utilization) and the user's analytical jobs, profile-based executing planning aims to allocate the most appropriate resources to each component, maximizing the end-to-end throughput subject to performance targets. The problem is formulated as follows,

$$\max \ \ T_{e2e}, \ \ s.t. \sum R_u \leq R \ \ \forall u \in V(G).$$

where G is the dataflow graph (DFG) of components; u is the node in graph G and R_u indicates its allocated resource.

We present a profile-based execution planning, as the example of one CPU plus one GPU depicted in Fig. 12, that ① parses the DFG of user-specified analytical tasks, ② runs workload (*e.g.*, one-minute user-specified streams) on all components, including user-uploaded models to profile their capacity on all accessible hardware, ③ generates the execution plan that satisfies user-specified performance targets, and ④ loads each component into corresponding hardware. Our crucial choice is to assign each component's input tensor size (*e.g.*, batch size) for resource allocation. Batching execution, *i.e.*, grouping input matrices into one, is well known to achieve high processor utilization by DNNs; it also allows the inference engines to achieve various throughputs by adjusting different batch sizes [36, 37, 76].

As the DFG of any job, naturally, is a directed acyclic graph (DAG), we can always leverage *dynamic programming* to solve this optimization problem [75]. Define $T_u(r)$ as the maximum throughput of components represented by u and the subtree at u within the resource budget r, which numerically equals the minimum node in each path. For non-leaf node u, the algorithm allocates a resource budget r' for node u and at most $r-r'$ for the subtree, and it then enumerates all $r \leq R$ to find the optimal allocations that assign optimal batch size b for each node. Formally, $\forall (u,v) \in E(G)$,

$$T_u(r) = \max_{r:r\leq R} \left\{ \min(\max_{b:c_u(b)\leq r'} \frac{b}{c_u(b)}, T_v(r-r')) \right\}.$$

where $c_u(b)$ is the resource cost of node u at b batches. RegenHance allocates the least resources for analytical models that satisfy the user's latency target and then assigns other components' batch sizes with this equation.

In general, the optimal solution always converges to the allocation that won't be bottlenecked by any node; in other words, each node in the graph generates the same throughput. Therefore, when users' registrations change frequently, the execution plan must be generated quickly. To this end, we will explore learning-related methods like online deep reinforcement learning [26] and combinatorial-optimization-related methods like local search [50] in future work.

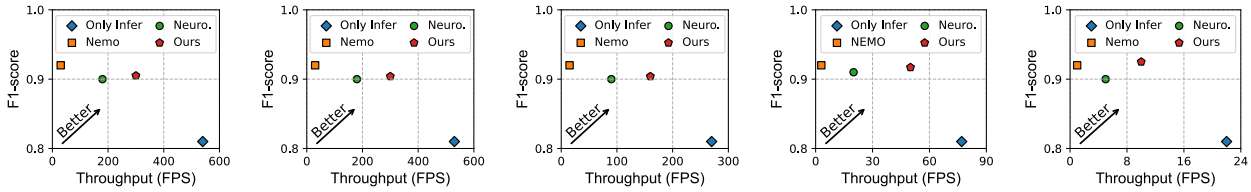
4 Evaluation

We evaluate RegenHance with two video analytics tasks on five heterogeneous devices. Our key findings are:

- RegenHance improves 10-19% accuracy and achieves 2-3× end-to-end throughput compared to the state-of-the-art frame-based enhancement methods. (Sec. 4.3)
- RegenHance shows robust effectiveness on various devices with heterogeneous computational resources, different analytical tasks and models, diverse workloads and performance targets, and various resolutions. (Sec. 4.3)
- MB-based Region Importance Prediction and Region-aware Enhancement bring remarkable accuracy and throughput benefit, and Profile-based Execution Planning boosts great resource utilization and throughput. (Sec. 4.4)

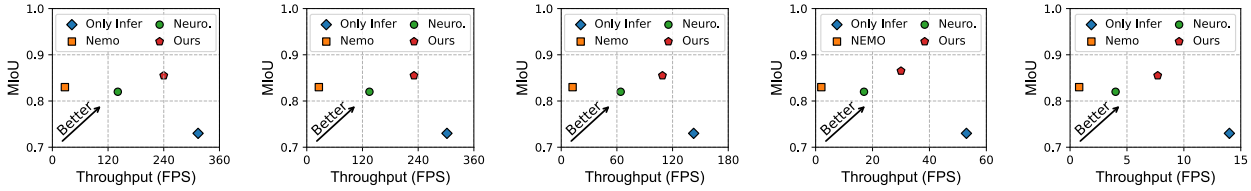
4.1 Implementation

We implement RegenHance upon commercial frameworks including FFmpeg (v4.4.2) [19], Pytorch (v1.8.2) [20], Paddleseg (v2.7.0) [27], ONNX, OpenVINO (v2023.0.1) [22], TensorRT (v8.4.2.4) [11], and the mostly code is implemented by Python (v3.8). The MB-based region importance prediction and the region-aware enhancer are implemented as follows. (1) RegenHance implements the MB importance predictor by retraining MobileSeg (MobileNetV2 backbone) with importance metric and prunes its 50% parameters by L1-Norm pruner. It is further exported to the ONNX version (using `paddle.onnx.export` API) for efficient running on Intel CPU with OpenVINO Runtime, and exported to the TensorRT version for NVIDIA GPU (using `trtexec` library). If not mentioned, all TensorRT models are set as `FP16` in dynamic shape version, and the engine files are exported and inferred by PyCUDA (v2022.2). (2) We modify `ff_h264_idct_add` API in



(a) 4090 (b) A100 (c) 3090Ti (d) T4 (e) Jonson AGX Orin

Figure 13: Accuracy and throughput comparison over various devices (Object detection).



(a) 4090 (b) A100 (c) 3090Ti (d) T4 (e) Jetson AGX Orin

Figure 14: Accuracy and throughput comparison over various devices (Semantic segmentation).

Tasks	Metric	Models	Dataset
Object Detection	F1-score Stream #	Mask R-CNN (Swin) YOLO	YODA VideoClips
Semantic Segmentation	mIoU Stream #	FCN HarDNet	Cityscape BDD100K

Table 1: Summary of downstream tasks and datasets.

FFmpeg to extract the residual for temporal MB importance reuse. (3) RegenHance uses a pre-trained super-resolution model [64] as enhancer, and also converts it to ONNX version and TensorRT version for efficiency.

Loading models into memory can cost hundreds of milliseconds to seconds. Therefore, at runtime, RegenHance preloads the DNN into the specified processors and then invokes it. RegenHance consists of ∼5.3K lines of code.

4.2 Experimental Setup

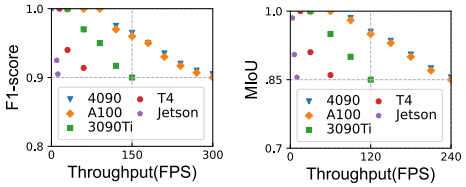
Downstream tasks and dataset. As summarized in Tab.1, we select two downstream analytical tasks, object detection and semantic segmentation, to evaluate the performance of RegenHance as they play the core role in a wide range of high-level tasks. Object detection aims to identify objects of interest (*i.e.*, locations and classes) on each video frame; its accuracy is measured by average *F1-score* in each stream with Intersection over Union (IoU) threshold at 0.5. Semantic segmentation labels each pixel with one class, and we measure its accuracy with *mIoU*. We also evaluate their *throughput*, *i.e.*, number of streams that can be processed in real-time.

Both of the tasks were separately tested with two models (light- and heavyweight) on two video sets. For the object detection, we used the Yoda dataset [88] and gathered 120 video clips from YouTube containing various scenes with diverse characteristics, *i.e.*, time, illumination, objects' density and speed, and road type. The labels to train importance predictor are generated with Mask R-CNN (Swin backbone) on per-frame enhancement since its SOTA performance; if users provide their self-tailored analytical models, we use their output as labels. Here, we use YOLO as an example. We will release this dataset after this paper is accepted. For semantic segmentation, we utilized the BDD100K [2] and Cityscape [35] public dataset. We re-encoded these datasets into 360P resolution, 30 fps, and 1024kbps bitrate videos in H.264 to avoid the impacts of various video codecs.

Devices. We conducted experiments on five heterogeneous devices divided into four categories. (1) As a comparison, we deployed and tested RegenHance on a cloud server with an NVIDIA A100 GPU and Intel(R) i9-12900K CPU. (2) As one of the most popular configurations of edge servers, we conducted experiments on an edge equipped with an NVIDIA Tesla T4 and Intel i7-8700 CPU. (3) To explore the capabilities of gaming cards and their generation gap, we tested an NVIDIA RTX4090 and an NVIDIA RTX3090Ti, respectively, with an Intel i9-13900K. (3) For the embedded edge, we used an NVIDIA Jetson AGX Orin 64GB as the platform.

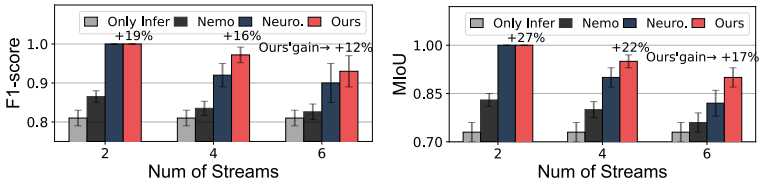
Baselines. To show the superiority brought by RegenHance, we compare it with four different baselines:

- **Only infer** directly applies analytical DNNs on each original frame without enhancement.
- **NeuroScaler [95]** is the state-of-the-art frame-based enhancement method. It first enhances the anchors and reuses their quality gain on non-anchors, then infers all frames. It fast selects anchors in a heuristic manner.
- **Nemo [93]** also only enhances the anchors and reuses their quality gain, then infers all frames; but it iteratively selects the best anchors based on enhancement results.



(a) Object detection (b) Semantic segmentation

Figure 15: TPT-ACC tradeoff on different devices.



(a) Object detection (b) Semantic segmentation

Figure 16: Accuracy comparison over various stream numbers.

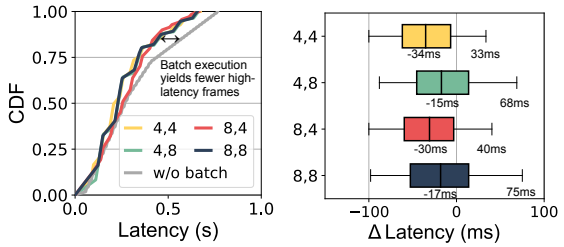


Figure 17: *Left*: Latency of each frame with diverse batch sizes (Infer#, SR#). *Right*: The latency diff between execution w/ and w/o batch of each frame ($\Delta Lat = Lat_{batch}(i) - Lat_{w/o\ batch}(i)$).

Metric	360P	720P
BW (Mbps)	0.96	3.00
Max Stream #	11	10
GPU Usage (SR)	0.23	0.17
Acc Gain	9%	7%

Table 2: Performance trade-off under different video resolutions.

Method	TPT (FPS)
Per-frame SR	95
PF + Plan.	111
PF + Pred. + Plan.	111
PF + Pred. + Enhan.	179
RegenHance	300

Table 3: End-to-end throughput (FPS) breakdown of RegenHance.

4.3 End-to-End Performance

This section reports the E2E performance of RegenHance on various devices. If not mentioned, all results are tested on RTX4090 with 1s latency, 90% object detection, and 88% semantic segmentation (10% gain compared to only infer).

Performance on heterogeneous devices. As shown in Fig. 13 and Fig. 14, RegenHance achieves high accuracy and high throughput simultaneously on various devices. It of course cannot reach the throughput of only infer due to the additional enhancement time cost but provides significant throughput gain compared to SOTA NEMO and NeuroScale. On average, RegenHance outperforms their throughput by 12× and 2.1× in object detection and 11× and 1.9× in semantic segmentation, respectively. This larger performance gain in semantic segmentation stems from its heightened sensitivity to visual details. RegenHance can keep this superiority on all five heterogeneous devices because profile-based execution planning always generates the optimal throughput plans.

Trade-off between throughput and accuracy. RegenHance creates a trade-off space between accuracy and throughput, as shown in Fig. 15; it can offer service on edge servers with heterogeneous resources. Edge servers equipped with higher resources produce larger trade-off space. For example, RegenHance supports object detection for ten streams (300fps) with 91% accuracy on RTX4090 or A100 GPU; and if the accuracy constraint is more strict, RegenHance will make corresponding adjustments and serve the maximum number of streams, *e.g.*, six 95%-accuracy streams. On devices with lower available resources, *e.g.*, on NVIDIA T4 and Jetson AGX Orin, although the maximum frame rate decreases, RegenHance still delivers remarkable throughput under different accuracy targets.

Performance gain with multiple streams. With the increasing number of competing streams, RegenHance always achieves the highest accuracy compared to the other three frame-based enhancement methods. For example, in Fig. 16 tests on RTX4090, RegenHance improves accuracy by 8-14% compared to the selective enhancement in six streams. This is because, in such a highly competitive scenario where each stream is allocated limited resources, our method consistently enhances the most valuable regions; in contrast, the selective baseline and the per-frame baseline waste excessive resources on unimportant content. RegenHance significantly improves the throughput of content-enhanced video analytics.

Frame latency under diverse batch sizes. The latency is defined the same as prior studies (*e.g.*, DDS [44], AWStream [101], and Reducto [62]), *i.e.*, the time from encoding 1s video chunk (30 frames) on cameras to all 30 frames' inference results are available on edge. The left figure of Fig. 17 plots the latency of each frame with different batch sizes under the same workflow. The results imply that batch execution yields fewer high-latency frames because its average latency of 30 frames in each chunk is much lower than executed without batch, which stems from the batch's higher GPU utilization. For a deeper insight, the right figure of Fig. 17 evaluates the latency difference of each frame with and without batch execution. Compared to without batch, execution with batch may cause 75ms latency at most, which is the earliest frame (in video play order) in one batch; but on average, batch execution saves time cost.

Performance gain under different resolutions. When streaming 720p in object detection while 360p for semantic segmentation under a target of 90% accuracy, as shown in Table 2, 360p video costs only 31% bandwidth (0.96Mbps *v.s.*

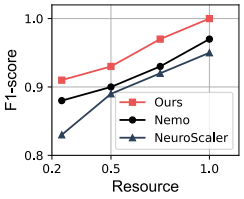


Figure 18: Accuracy on different resources.

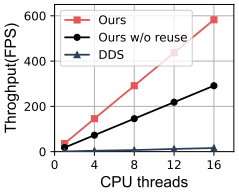


Figure 19: Throughput of region prediction.

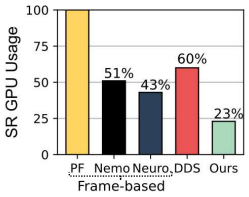


Figure 20: GPU resources usage.

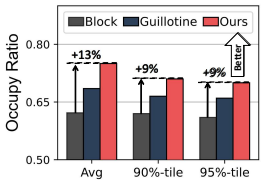


Figure 21: Perf. of different packing policies.

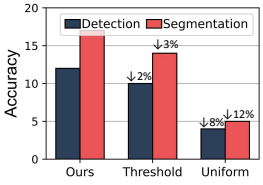


Figure 22: Acc gain of Cross-stream MB sel.

3Mbps) compared to 720p. RegenHance boosts accuracy gain of 360p video from 81% to 90%, while 720p video from 83% to 90%. Higher resolution yields higher accuracy; however, content enhancement can still improve the details of small objects or blurred content. The end-to-end throughput of 720p is almost the same as 360p (11 *v.s.* 10 streams). The reason is that although 720p needs to enhance fewer regions compared to 360p (17% *v.s.* 23% GPU usage for SR), its time cost of other runtime components (*e.g.*, MB importance prediction) is larger than 360p due to the larger input size.

Contributions by individual components. Table 3 shows the throughput breakdown of RegenHance. We tested multiple versions of RegenHance by selectively applying each component. The results show that each component significantly contributes to the improvements in the throughput. The execution planning enables a 1.2× throughput compared to per-frame enhancement by reasonable resource allocation (1st→2nd). Incorporating MB-based region importance prediction without region-aware enhancement fails to improve throughput because filling unimportant regions to black does not alter the enhancement latency (2nd→3rd row). The throughput is increased by 1.6× due to the region-aware enhancement (3rd→4th row) and by 1.7× from the best execution plan of profile-based execution planning (4th→5th row).

4.4 Comprehensive Component-wise Analysis

We provide an in-depth performance analysis of individual system components.

4.4.1 MB-based region Importance Prediction (§3.2) helps RegenHance achieve great throughput improvement while keeping high accuracy.

Accuracy: Fig. 18 illustrates the accuracy gain obtained by different enhancement methods, when assigned the same computational resource to complete object detection on six streams. RegenHance (region-based enhancement) can offer a 3-4% and 4-8% higher gain compared to (frame-based) Nemo and Neuralscaler, respectively, because our predictor identifies the most beneficial region precisely.

Thoughput: As shown in Fig. 19, our ultra-lightweight MB importance predictor can be executed at 30 fps on one single i7-8700 CPU core, which outperforms the RoI selection by RPN in DDS more than 60 times. On GPU, it achieves 973 fps, which outperforms DDS more than 12 times. The reuse contributes 2 times throughput improvement.

Resource saving: Fig. 20 shows that our method reduces 77%, 28%, and 20% GPU usage compared to the frame-based Per-frame enhancement, Nemo, and NeuroScaler, respectively, when enhancing a single stream (30FPS) to achieve an accuracy exceeding 90%. Compared to the RoI selection of DDS, our method saves 37% GPU usage as it identifies the most beneficial region for analytical accuracy more precisely.

4.4.2 Region-aware Enhancement (§3.3) yields significant throughput improvement and overall accuracy gain.

Throughput: Region-aware bin packing maximizes RegenHance's throughput with a high stability and occupy ratio, and thus effectively reducing the additional overhead of SR. We conducted experiments repeated 1,000 times by randomly shuffling the order of six video streams, and comparing the occupy ratio, *i.e.*, the ratio of selected MBs occupying all enhanced content, with the classic Guillotine policy [57] and the Block policy, *i.e.*, MB packing. Fig. 21 shows that the average, 90%-tile, and 95%-tile occupy ratio difference; our packing policy gets the highest occupy ratio of 75% that outperforms the comparisons by up to 13%, 9%, and 9%.

Accuracy: Cross-stream MB selection, especially our custom sorting order, leads to remarkable accuracy gain by considering the heterogeneous region accuracy gain across streams. We compared our MB selection with the Uniform method, which allocates the same MB number to each stream, and the Threshold method sets a fixed threshold, 0.5, to select MB importance for all streams. As shown in Fig. 22, our method outperforms the Uniform and Threshold methods by 8-12% and 2-3% accuracy gain, respectively. Fig 23 demonstrates the superiority of our custom sorting priority. Compared to the traditional large-item-first (max-region-first in our context) packing policy, our method yields a 50% accuracy gain.

4.4.3 Profile-based Execution Planning (§3.4) profiles different devices and allocates devices and resources to different components and models based on the results, thereby maximizing throughput.

Resources allocation: Profile-based execution planning makes RegenHance achieve the best performance on given devices and workloads by avoiding bottlenecked by any component. Fig. 24 visualizes the computational resource allocation of two different object detection models on i9-13900K+RTX4090 providing satisfied performance as Fig. 13(a). With distinct workloads of YOLOv5s (16.9GFLOPs)

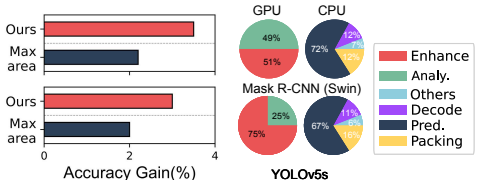


Figure 23: Acc gain of our MB selection.

Figure 24: Execution plan for different workload.

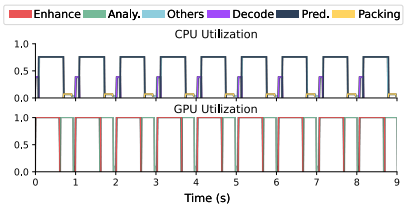


Figure 25: GPU & CPU usage.

Component	RRobin	Ours
MB Prediction	194	533
Enhancement	80	186
Analytics	220	186
Throughput	**80**	**186**

Table 4: Throughput (FPS) gain to region-agnostic strawman in §2.4.

and Mask R-CNN (Swin backbone, 267GFLOPS), it allocates more resources for the analytical task.

Resource usage: Fig. 25 shows the real-time utilization of CPU and GPU when detecting objects on six video streams. We used Nvidia Nsight to monitor GPU utilization and HTOP for CPU utilization. The GPU can reach full load approximately 95-99% of the time, while the CPU can reach high load around 81% of the time. This result indicates that the execution plan achieves an efficient GPU-CPU corporation.

Throughput: We avoid the enhancement bottleneck introduced by the round-robin manner (§2.4), which allocates equal resources to each component. The results in Table 4 indicate that our approach achieves 2.3× throughput.

Scalability and initialization: There are two types of preparation time in our system. If only the ingest video streams change, it needs approximately 0.6-2 seconds for initialization depending on the specific device and model requirements. When given a new device, it needs to take approximately 1-3 minutes to execute the Profile-based Execution Planning.

5 Related Work

Video analytics is facilitated by the advances of deep learning. However, the high analytical accuracy from deep learning comes at a prohibitive computing cost that most end devices cannot afford [87]. For example, today's deployed traffic cameras cost only $40 to $200, equipped with a single-core CPU that provides very scarce computational resources. To tackle this issue, today's video analytics apply a *distributed* architecture [45, 51, 58, 62, 65, 69, 91, 98–100, 102, 103].

Video compression saves bandwidth resources by adjusting video encoding configurations (*e.g.*, resolution, bitrate, and frame rate). Like traditional video streaming protocols like WebRTC [47] and DASH [77], distributed video analytics must tackle bandwidth constraints. Prior studies, no matter server-feedback accurate configuration setup [56, 69, 101, 103] or fast autonomy by video source (camera) itself [45, 65, 89, 91], addressed this issue well, saving considerable bandwidth resources but causing accuracy drop.

Selective inference, also called frame/image filtering, filters similar frames/images out to meet low resource cost and high throughput. Cameras leverage pixel-level difference [34, 40, 62] to filter frames for bandwidth saving; edge servers use light DNNs to select the frames to be detected by heavy DNNs, then reuse their output on filtered frames for throughput increment [51, 58, 98, 100, 102]. Selective inference achieves satisfactory throughput and resource-saving but lower accuracy. DDS [44] iteratively detects the selected images to avoid accuracy drop, but its twice cross-network transmission and detection imports too much delay. Turbo [68] seeks the same goal by enhancing frames on idle GPU slots. Differing from it, RegenHance boosts the enhancement method and fully uses heterogeneous resources on edge.

Model optimization meets high throughput by network pruning [61] and weights quantization [54], and achieves high accuracy by knowledge distillation [49]. However, the compressed model is easily affected by data drift, where the live video data diverges from the training data, leading to accuracy degradation [74]. To tackle this, continuous learning [60, 74], model switching [59, 95], and model merging [55, 71] are proposed by previous studies. Intuitively, continuous learning and model switching leverage small DNNs for high throughput and improve accuracy by matching the weights in DNN to the changed real-world videos. Model merging differs from traditional parameter sharing in one model [72]. It explores structural similarities between concurrent models and enables multiple models to share the same network components, *e.g.*, the backbone [71], by retraining.

6 Conclusion

In this paper, we look at content enhancement in video analytics applications. We found that frame-based content enhancement wastes too much computation on the analytics-agnostic image. We presented the region-based content enhancement technique and well-matched region-aware resource scheduler and implemented RegenHance. In our evaluation using five heterogeneous devices, we show that RegenHance can deliver an order of magnitude than frame-based content-enhanced video analytics.

Acknowledgements. We thank the anonymous NSDI reviewers for their constructive comments. This work was supported by Carbon Neutrality and Energy System Transformation (CNEST) Program, Tsinghua University (AIR)-AsiaInfo Technologies (China), Inc. Joint Research Center for 6G Network and Intelligent Computing, NSFC (No. 62272261, 62402280, U22A2031) the Fundamental Research Funds for the Central Universities (No. 2024300349), EU Horizon CODECO projects (No. 101092696), Shuimu Tsinghua Scholar Program (No. 2023SM201).

References

[1] Axis. axis for a safety touch at the grey cup festival. https://www.axis.com/files/success_stories/ss_stad_greycup_festival_58769_en_1407_lo.pdf.

[2] Bdd100k dataset. https://www.vis.xyz/bdd100k/.

[3] British transport police: Cctv. http://www.btp.police.uk/adviceandinformation/safetyonandneartherailway/cctv.aspx.

[4] Can 30,000 cameras help solve chicago's crime problem? https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html.

[5] Global sports analytics market size report, 2021-2028. https://www.grandviewresearch.com/industry-analysis/sports-analytics-market.

[6] Google cloud vision. https://cloud.google.com/vision.

[7] H.264 specification. https://www.itu.int/rec/T-REC-H.264.

[8] Kubeedge. https://github.com/kubeedge/kubeedge.

[9] Microsoft rocket. https://www.microsoft.com/en-us/research/project/live-video-analytics/.

[10] NVIDIA T4. https://www.nvidia.com/en-us/data-center/tesla-t4/.

[11] NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[12] One legacy of tiananmen: China's 100 million surveillance cameras. https://www.wsj.com/articles/BL-CJB-22562.

[13] One surveillance camera for every 11 people in britain, says cctv survey. https://www.telegraph.co.uk/technology/10172298/One-surveillance-camera-for-every-11-\people-in-Britain-says-CCTV-survey.html.

[14] Robo scores world cup to have 'robot linesmen' for first time in history as fifa ready to say trials have worked. https://www.thesun.co.uk/sport/18807051/world-cup-robot-linesman-fifa-qatar/.

[15] Video analytics applications in retail - beyond security. https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html/.

[16] Video analytics market - growth, trends, covid-19 impact, and forecasts (2022 - 2027). https://www.mordorintelligence.com/industry-reports/video-analytics-market.

[17] Video analytics traffic study creates baseline for change. https://www.govtech.com/analytics/Video-Analytics-Traffic-Study-Creates\-Baseline-for-Change.html.

[18] Wyze. wyze camera. https://www.safehome.org/homesecurity-cameras/wyze/.

[19] Ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/, 2021.

[20] Pytorch: Official website of the python platform. https://pytorch.org/, 2021.

[21] Speedtest market report. https://www.speedtest.net/reports/zh/, 2021.

[22] Openvino™ toolkit: An open source toolkit that makes it easier to write once, deploy anywhere. https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html, 2023.

[23] Neil Agarwal and Ravi Netravali. Boggart: Towards General-Purpose acceleration of retrospective video analytics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 933–951, 2023.

[24] Namhyuk Ahn, Byungkon Kang, and Kyung-Ah Sohn. Fast, accurate, and lightweight super-resolution with cascading residual network. In *European Conference on Computer Vision (ECCV)*, 2018.

[25] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodik, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 50(10):58–67, 2017.

[26] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[27] PaddlePaddle Authors. Paddleseg, end-to-end image segmentation kit based on paddlepaddle. https://github.com/PaddlePaddle/PaddleSeg, 2019.

[28] Judith O Berkey and Pearl Y Wang. Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society*, 38:423–429, 1987.

[29] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *arXiv preprint arXiv:1712.09665*, 2017.

[30] Adrian Bulat and Georgios Tzimiropoulos. Super-fan: Integrated facial landmark localization and super-resolution of real-world low resolution faces in arbitrary poses with gans. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[31] F Cangialosi, N Agarwal, V Arun, J Jiang, S Narayana, A Saarwate, and R Netravali. Privid: Practical, privacy-preserving video analytics queries. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[32] Ping Chao, Chao-Yang Kao, Yu-Shan Ruan, Chien-Hsiang Huang, and Youn-Long Lin. Hardnet: A low memory traffic network. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 3552–3561, 2019.

[33] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[34] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 155–168, Seoul, South Korea, 2015.

[35] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[36] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *IEEE International Conference on Computer Design (ICCD)*, pages 497–505, 2019.

[37] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. DVABatch: Diversity-aware Multi-Entry Multi-Exit batching for efficient processing of DNN services on GPUs. In *USENIX Annual Technical Conference (ATC)*, pages 183–198, 2022.

[38] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15, 2021.

[39] Jason Jinquan Dai, Ding Ding, Dongjie Shi, Shengsheng Huang, Jiao Wang, Xin Qiu, Kai Huang, Guoqiong Song, Yang Wang, Qiyuan Gong, Jiaming Song, Shan Yu, Le Zheng, Yina Chen, Junwei Deng, and Ge Song. Bigdl 2.0: Seamless scaling of ai pipelines from laptops to distributed cluster. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 21407–21414, 2022.

[40] Xin Dai, Xiangnan Kong, Tian Guo, and Yixian Huang. Cinet: Redesigning deep neural networks for efficient mobile-cloud collaborative inference. In *SIAM International Conference on Data Mining (ICDM)*, pages 459–467, 2021.

[41] Mallesham Dasari, Arani Bhattacharya, Santiago Vargas, Pranjal Sahu, Aruna Balasubramanian, and Samir R Das. Streaming 360-degree videos using super-resolution. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1977–1986, 2020.

[42] Sokemi Rene Emmanuel Datondji, Yohan Dupuis, Peggy Subirats, and Pascal Vasseur. A survey of vision-based traffic monitoring of road intersections. *IEEE Transactions on Intelligent Transportation Systems*, 17(10):2681–2698, 2016.

[43] Sina G. Davani and Nabil J. Sarhan. Experimental analysis of optimal bandwidth allocation in computer vision systems. *TCSVT*, 31(10):4121–4130, 2021.

[44] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-Driven Video Streaming for Deep Learning Inference. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 557–570. ACM, 2020.

[45] Kuntai Du, Qizheng Zhang, Anton Arapin, Haodong Wang, Zhengxu Xia, and Junchen Jiang. Accmpeg: Optimizing video encoding for video analytics. In *Machine Learning and Systems*, 2022.

[46] Glenn Jocher et. al. ultralytics/yolov5: v6.0 - YOLOv5n 'Nano' models, Roboflow integration, TensorFlow export, OpenCV DNN support, October 2021.

[47] Google. Webrtc official website. https://webrtc.org/, 2011.

[48] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[49] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[50] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[51] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, page 19, 2018.

[52] Weiming Hu, Nianhua Xie, Li Li, Xianglin Zeng, and Stephen Maybank. A survey on visual content-based video indexing and retrieval. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(6):797–819, 2011.

[53] Jinwoo Hwang, Minsu Kim, Daeun Kim, Seungho Nam, Yoonsung Kim, Dohee Kim, Hardik Sharma, and Jongse Park. Cova: Exploiting compressed-domain analysis to accelerate video analytics. In *USENIX Annual Technical Conference (ATC)*, pages 707–722, 2022.

[54] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, 2018.

[55] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, 2018.

[56] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 253–266, August 2018.

[57] J. Jylänki. A thousand ways to pack the bin-a practical approach to two-dimensional rectangle bin packing. *http://clb. demon. fi/files/RectangleBinPack. pdf.*, 2010.

[58] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, August 2017.

[59] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. RECL: Responsive resource-efficient continuous learning for video analytics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 917–932, 2023.

[60] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 107–125, 2020.

[61] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[62] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 359–376, 2020.

[63] Jingyun Liang, Jiezhang Cao, Guolei Sun, Kai Zhang, Luc Van Gool, and Radu Timofte. Swinir: Image restoration using swin transformer. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1833–1844, 2021.

[64] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1132–1140, 2017.

[65] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 1–16, August 2019.

[66] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, 2015.

[67] Eunice López-Camacho, Gabriela Ochoa, Hugo Terashima-Marín, and Edmund K Burke. An effective heuristic for the two-dimensional irregular bin packing problem. *Annals of Operations Research*, 206:241–264, 2013.

[68] Yan Lu, Shiqi Jiang, Ting Cao, and Yuanchao Shu. Turbo: Opportunistic enhancement for edge video analytics. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 263–276, 2022.

[69] Marwa Meddeb. Region-of-interest-based video coding for video conference applications. page 172.

[70] Junhyug Noh, Wonho Bae, Wonhee Lee, Jinhwan Seo, and Gunhee Kim. Better to follow, follow to be better: Towards precise supervision of feature super-resolution for small object detection. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9724–9733, 2019.

[71] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge. In *USENIX Symposium on Network System Design and Implementation (NSDI)*, 2023.

[72] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning (ICML)*, pages 4095–4104. PMLR, 2018.

[73] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor:Privacy-Preserving video analytics as a cloud service. In *USENIX Security Symposium (Security)*, pages 1039–1056, 2020.

[74] Bhardwaj Romil, Xia Zhengxu, Ananthanarayanan Ganesh, Jiang Junchen, Shu Yuanchao, Karianakis Nikolaos, Hsieh Kevin, Bahl Paramvir, and Stoica Ion. Ekya: Continuous learning of video analytics models on edge compute servers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[75] Dasgupta Sanjoy, Papadimitriou Christos, and Vazirani Umesh. *Algorithms*. 2008.

[76] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 322–337, 2019.

[77] Thomas Stockhammer. Dynamic adaptive streaming over http– standards and design principles. In *Pro. of MMSys*, pages 133–144, 2011.

[78] Gary J Sullivan and Thomas Wiegand. Rate-distortion optimization for video compression. *IEEE Signal Processing Magazine*, 15(6):74–90, 1998.

[79] Lin Sun, Weijun Wang, Tingting Yuan, Liang Mi, Haipeng Dai, Yunxin Liu, and Xiaoming Fu. Biswift: Bandwidth orchestrator for multi-stream video analytics on edge. *arXiv preprint arXiv:2312.15740*, 2024.

[80] Weimin Tan, Bo Yan, and Bahetiyaer Bare. Feature super-resolution: Make machine see more clearly. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3994–4002, 2018.

[81] Shiyu Tang, Ting Sun, Juncai Peng, Guowei Chen, Yuying Hao, Manhui Lin, Zhihong Xiao, Jiangbin You, and Yi Liu. Pp-mobileseg: Explore the fast and accurate semantic segmentation model on mobile devices. *arXiv preprint arXiv:2304.05152*, 2023.

[82] Li Wang, Dong Li, Yousong Zhu, Lu Tian, and Yi Shan. Dual super-resolution learning for semantic segmentation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3774–3783, 2020.

[83] Weijun Wang, Tingting Yuan, Minghao Han, Meng Li, Han Zhang, Yu Ma, Sripriya Srikant Adhatarao, and Xiaoming Fu. Poster: A real-time social distance measurement and record system for covid-19. In *ACM International Conference on Embedded Wireless Systems and Networks*, page 179–180, 2021.

[84] Yiding Wang, Weiyan Wang, Duowen Liu, Xin Jin, Junchen Jiang, and Kai Chen. Enabling edge-cloud video analytics for robotics applications. *IEEE Transactions on Cloud Computing*, 2022.

[85] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the {Edge-Cloud} barrier for real-time advanced vision analytics. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[86] Fabian Wölk, Tingting Yuan, Krisztina Kis-Katos, and Xiaoming Fu. Measuring consumption changes in rural villages based on satellite image data—a case study for thailand and vietnam. In *IEEE International Conference on Mobility, Sensing and Networking (MSN)*, pages 600–607, 2021.

[87] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming

Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at facebook: Understanding inference at the edge. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[88] Zhujun Xiao, Zhengxu Xia, Haitao Zheng, Ben Y Zhao, and Junchen Jiang. Towards performance clarity of edge video analytics. In *IEEE/ACM Symposium on Edge Computing (SEC)*, pages 148–164, 2021.

[89] Xiufeng Xie and Kyu-Han Kim. Source Compression with Bounded DNN Perception Loss for IoT Edge Computer Vision. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 1–16, 2019.

[90] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. DeepCache: Principled Cache for Mobile Deep Vision. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 129–144, 2018.

[91] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *ACM conference on Embedded Networked Sensor Systems (SenSys)*, pages 476–488, 2020.

[92] Juncheol Ye, Hyunho Yeo, Jinwoo Park, and Dongsu Han. Accelir: Task-aware image compression for accelerating neural restoration. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 18216–18226, 2023.

[93] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In *ACM International Conference on Mobile Computing and Networking, (MobiCom)*, pages 1–14, 2020.

[94] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 645–661, 2018.

[95] Hyunho Yeo, Hwijoon Lim, Jaehong Kim, Youngmok Jung, Juncheol Ye, and Dongsu Han. Neuroscaler: Neural video enhancement at scale. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 795–811, 2022.

[96] Juheon Yi, Sunghyun Choi, and Youngki Lee. EagleEye: wearable camera-based person identification in crowded urban spaces. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 1–14, April 2020.

[97] Juheon Yi, Seongwon Kim, Joongheon Kim, and Sunghyun Choi. Supremo: Cloud-assisted low-latency super-resolution in mobile devices. *IEEE Transactions on Mobile Computing*, 21(5):1847–1860, 2020.

[98] Mu Yuan, Lan Zhang, Fengxiang He, Xueting Tong, and Xiang-Yang Li. Infi: End-to-end learnable input filter for resource-efficient mobile-centric inference. In *ACM International Conference on Mobile Computing And Networking (MobiCom)*, page 228–241, 2022.

[99] Mu Yuan, Lan Zhang, Xuanke You, and Xiang-Yang Li. Packetgame: Multi-stream packet gating for concurrent video inference at scale. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2023.

[100] Tingting Yuan, Liang Mi, Weijun Wang, Haipeng Dai, and Xiaoming Fu. Accdecoder: Accelerated decoding for neural-enhanced video analytics. In *IEEE Conference on Computer Communications (INFOCOM)*, 2023.

[101] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. AWStream: adaptive wide-area streaming analytics. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 236–252. ACM, 2018.

[102] Tan Zhang, Aakanksha Chowdhery, Paramvir Victor Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In *ACM Conference on Mobile Computing and Networking (MobiCom)*, pages 426–438. ACM, 2015.

[103] Wuyang Zhang, Zhezhi He, Luyang Liu, Zhenhua Jia, Yunxin Liu, Marco Gruteser, Dipankar Raychaudhuri, and Yanyong Zhang. Elf: Accelerate high-resolution mobile deep vision with content-aware parallel offloading. In *ACMInternational Conference on Mobile Computing and Networking (MobiCom)*, pages 201–214, 2021.

[104] Zhengdong Zhang and Vivienne Sze. Fast: A framework to accelerate super-resolution processing on compressed videos. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 19–28, 2017.

[105] Yuan Zhi, Zhan Tong, Limin Wang, and Gangshan Wu. Mgsampler: An explainable sampling strategy for video action recognition. In *IEEE/CVF International conference on Computer Vision (ICCV)*, pages 1513–1522, 2021.

A Functions invoked in Algorithm 1

Algorithm 2 InnerFree(farea,box)

Input: farea,box
Output: the rest free area in box

```
function InnerFree(farea, box)
    m,n,dp=box.w,box.h,List[m][n]
    Init(dp)
    for j in n do
        up,down,stk=List[m],List[m],Stack()
        for i in m do
            if stk is not None and left[stk.top()][j] >= left[i][j] then
                down[stk.pop()] = i
            if stk is not None then
                up[i] = stk.top()
            stk.push(i)
        for i in m do
            height = down[i] - up[i] - 1
            area = height * left[i][j]
            get the free area with max area
    return max free area
```

B Performance of Importance Level Approximation

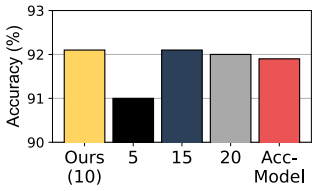


Figure 26: Importance level classification of MBs (*i.e.*, MB-grained image segmentation assigns each MB an importance level) yields comparable (even better) accuracy than AccModel, which predicts exact importance value.

Approximating the MB importance prediction as an image segmentation problem to assign each MB an importance level (class) is effective and produces good performance. We have trained four MB importance predictors (in MobileSeg architecture) with 5, 10, 15, and 20 important levels, respectively. Fig. 26 compares the final accuracy of inference task when predicting the exact MB importance (AccModel) and important levels. Experiment results demonstrate that important level classification yields precise MB importance prediction and hence high accuracy provided the level number is not very coarse, saying 5. The efficiency of this importance level classification is because such shallow model architecture is better at more manageable classification tasks (*e.g.*, segmentation) than regression tasks (*e.g.*, AccModel predicts the exact importance value). RegenHance set level number as 10.

C Additional Results

C.1 Example Eregions and the Distribution

Example eregions of object detection in §2.3 and semantic segmentation in Fig. 27 are generated by simply bounding the different analytical results between original and enhanced frames with a rectangle. Note that these example eregions are not the eregions used in RegenHance as they still contain much unnecessary content; this is why we propose to set fine-grained MB as the construction unit of eregions in RegenHance.

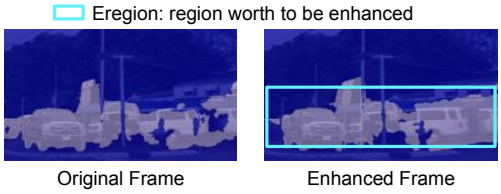


Figure 27: Example eregion worth to be enhanced for semantic segmentation (SS) bounding with a rectangle box.

In semantic segmentation, as illustrated in Fig. 28, only 10-15% area in 70% frames are eregions.

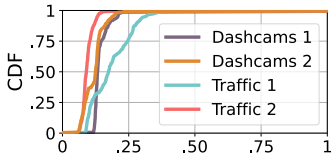
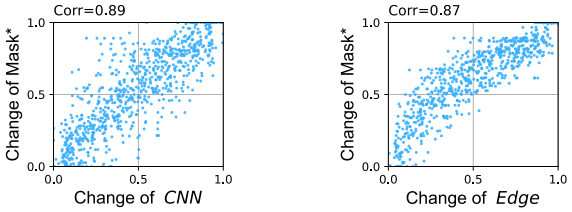


Figure 28: Distribution of eregions in Semantic segmentation.

C.2 Performance of Diverse Operators



(a) CNN [105] (b) Edge

Figure 29: Correlation ratio between the change of features.

As shown in Fig. 29, one-layer CNN [105] and the Edge operator (*i.e.*, edge detector in computer vision community) offer lower correlation ratios compared to the $\frac{1}{Area}$ operator, *i.e.*, 0.91. Fig. 30 exemplifies the characteristics of Area and $\frac{1}{Area}$. Images in the top raw demonstrate that the Area operator captures the change of large blocks, while the values of $\frac{1}{Area}$

change very small. On the contrary, $\frac{1}{Area}$ captures the change of small objects in two bottom images as RegenHance needs.



(a) $\frac{1}{Area}$: 0.002, Area: 0.665



(b) $\frac{1}{Area}$: 0.307, Area: 0.035

Figure 30: Correlation ratio between the change of features.

C.3 Performance *v.s.* Expanding Pixels

Pixel expanding in each direction can avoid the MB/region boundaries causing too many jagged edges and blocky artifacts when pasting enhanced content back to the bi-linear-interpolated frames. But it also introduces extra enhancement costs. We measure the accuracy gain and extra latency of the object detection task, as shown in Fig. 31. To balance the accuracy and the enhancement cost, we expand three pixels in this paper because the MB selection (§3.3.1) can control the number of enhanced MB to adjust the performance of RegenHance.

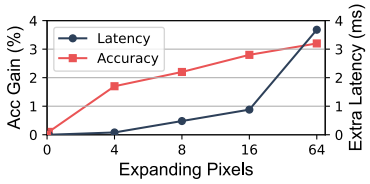


Figure 31: Both accuracy gain and enhancement cost increase with more expanding pixels. RegenHance expands 3 pixels in each direction surrounding every region.

C.4 Performance of MB Packing and Irregular Packing

Our region-aware packing algorithm spends almost the same time as MB packing and provides a close occupy ratio of irregular packing. As shown in Fig. 32, MB packing yields a low occupy ratio as packing too many unimportant or repeated pixels, while irregular packing leads to a large time cost of packing plan search.

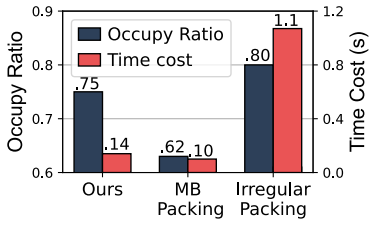


Figure 32: Our packing algorithm achieves a very good balance between bin utilization (occupy ratio) and the time cost of packing plan search.

C.5 An Example of Intermediate Results of RegenHance in Object Detection Task

Fig. 34 shows an example of packing five frames into one bin and pasting enhanced regions in the stitching image back to the interpolated frames. In particular, the region ④ is cut into two smaller ones by our region-aware bin-packing algorithm for better bin utilization. The green boxes in the left column bound the objects can be detected after enhancing the stitching image, *i.e.*, enhancing the selected regions. To the right column, region-based enhancement offers comparable accuracy with entire-frame enhancement.

C.6 Performance under Various Workloads and Users' Latency Targets

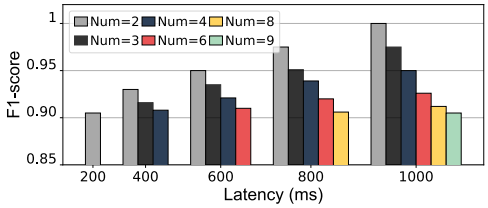


Figure 33: RegenHance supports various user-specified latency targets with adaptive batch sizes under different workloads (*i.e.*, stream numbers)

RegenHance supports various latency targets by automatically adjusting every component's batch size. Fig. 33 demonstrates that RegenHance can well analyze two 30-fps streams (60 frames) under 200ms latency budget and nine streams under 1s. The batch sizes of components change under different workloads. For example, under a 400ms budget, when stream numbers increase from 2 to 4, the batch sizes of (enhanced model, analytical model) are (8,4), (4,8), and (4,8); this implies more resources are allocated from region enhancement to final inference when workload increases, and hence lower accuracy. Note that the batch sizes of any components are no more than 8 among all latency targets; namely, in each batch, the earliest input won't wait for the last one more than 75ms as shown in Fig. 17.

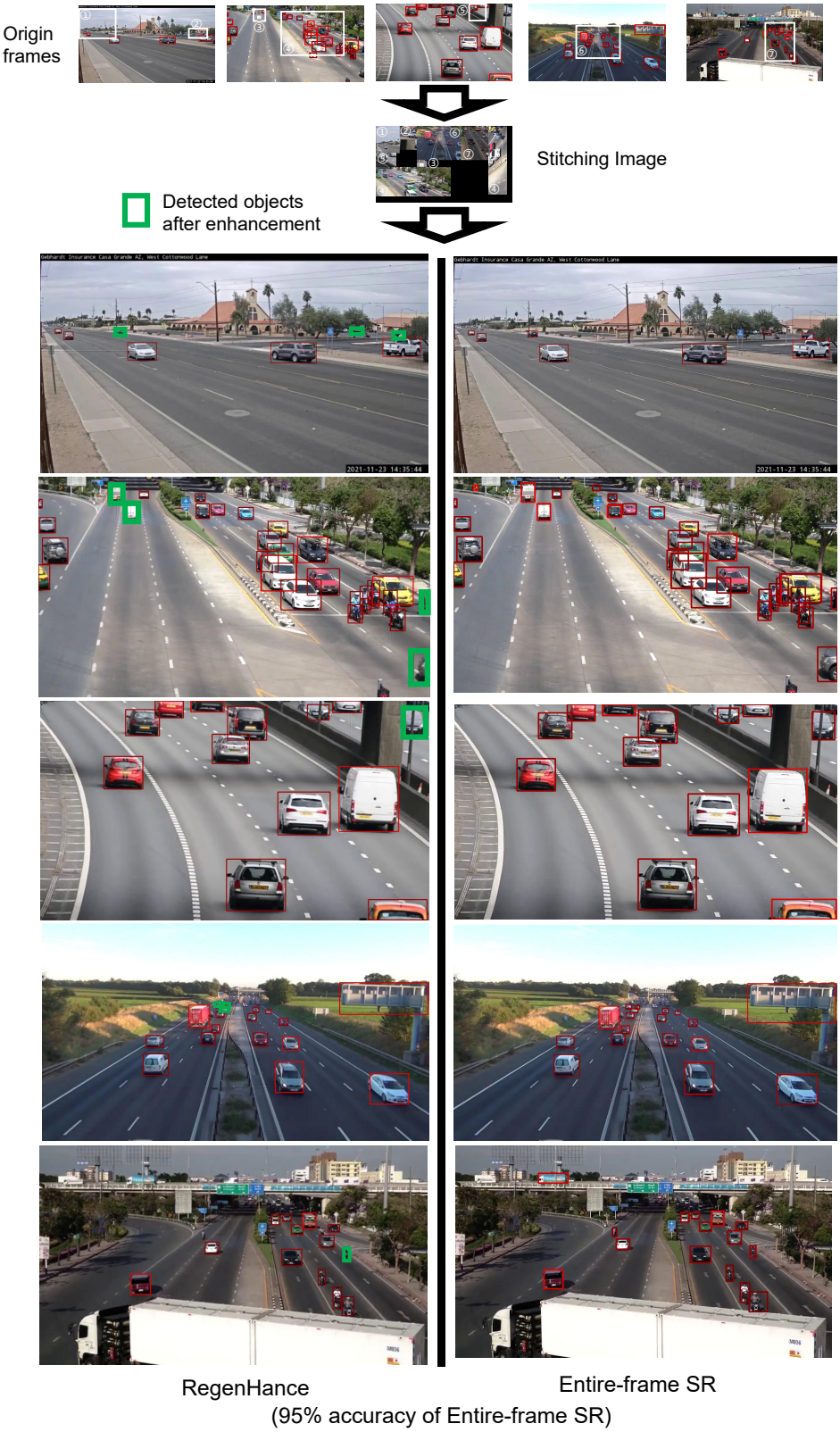


Figure 34: Examples of running super-resolution on stitched regions and their inference results compared to entire-frame enhancement.



Figure 35: Representative mask*.

Tooth: Toward Optimal Balance of Video QoE and Redundancy Cost by Fine-Grained FEC in Cloud Gaming Streaming

Congkai An[+], Huanhuan Zhang[+], Shibo Wang[+], Jingyang Kang[+], Anfu Zhou[+], Liang Liu[+], Huadong Ma[+], Zili Meng[◇], Delei Ma[°], Yusheng Dong[°], Xiaogang Lei[°]
[+]*Beijing University of Posts and Telecommunications*,
[◇]*Hong Kong University of Science and Technology*, [°]*Well-Link Times Inc.*

Abstract

Despite the rapid rise of cloud gaming, real-world evaluations of its quality of experience (QoE) remain scarce. To fill this gap, we conduct a large-scale measurement campaign, analyzing over 60,000 sessions on an operational cloud gaming platform. We find that current cloud gaming streaming suffers from substantial bandwidth wastage and severe interaction stalls simultaneously. In-depth investigation reveals the underlying reason, *i.e.*, existing streaming adopts coarse-grained Forward Error Correction (FEC) encoding, without considering the adverse impact of frame length variation, which results in over-protection of large frames (*i.e.*, bandwidth waste) and under-protection of smaller ones (*i.e.*, interaction stalls). To remedy the problem, we propose Tooth, a per-frame adaptive FEC that aims to achieve the optimal balance between satisfactory QoE and efficient bandwidth usage. To build Tooth, we design a dual-module FEC encoding strategy, which takes full consideration of both frame length variation and network dynamics, and hence determines an appropriate FEC redundancy rate for each frame. Moreover, we also circumvent the formidable per-frame FEC computational overhead by designing a lightweight Tooth, so as to meet the rigid latency bound of real-time cloud gaming. We implement, deploy, and evaluate Tooth in the operational cloud gaming system. Extensive field tests demonstrate that Tooth significantly outperforms existing state-of-the-art FEC methods, reducing stall rates by 40.2% to 85.2%, enhancing video bitrates by 11.4% to 29.2%, and lowering bandwidth costs by 54.9% to 75.0%.

1 Introduction

Cloud gaming, as an increasingly embraced application, involves high-quality video streaming from remote servers to diverse devices, thereby obviating the need for advanced local hardware and enabling pervasive gameplay across all connected devices, particularly mobile devices with limited computation and storage capacity. In recent years, several gaming platforms have launched acclaimed cloud-based versions of games, such as Cloud Genshin Impact [12] and Tower of Fantasy [13]. Market analyses reveal that the global cloud gaming market, which stood at US$ 1,905.6 million in 2023, is projected to reach US$ 48,552.2 million by 2032, demonstrating a compound annual growth rate of 42% [5].

To ensure a satisfactory interaction QoE, cloud gaming requires consistently ultra-low latency from player request to game response, typically less than 100 milliseconds [29]. To better understand the real-world QoE of cloud gaming, we conduct a comprehensive field measurement involving 66,128 players across 22 cities worldwide, as detailed in Table 1. Our measurements reveal that the downlink delivery of game video content (*i.e.*, from the remote server to the end devices) is the primary bottleneck impacting interactive QoE. To mitigate this, besides usual retransmission mechanisms [21,36], cloud gaming commonly adopts Forward Error Correction (FEC) from the current video streaming works [18, 20, 31, 33, 42] to recover lost data packets. However, our analysis indicates that existing FEC algorithms significantly increase bandwidth usage, adding an additional 0.26× to 1.0× redundancy than the original data streaming. We further conduct an in-depth exploration of existing FEC schemes and identified their two primary limitations.

- *Frame length affects the redundancy rate for recovering a frame.* Specifically, FEC can recover the lost packets within a frame when the redundancy rate can cover the Loss Rate In Frame (LRIF). However, we observe that the video frame length (number of packets) in cloud gaming varies largely, *i.e.*, spanning from 2 packets to over 80 packets, and frame LRIF is a random variable that follows a binomial-like distribution with a function of the frame length. Especially, the LRIF of a small frame is as high as 30+% and that of a large frame is as low as 2%, differing by over 10× and requiring completely different redundancy rates (§2.3).
- *Existing FEC algorithms are coarse-grained, ignoring frame LRIF differences but applying a uniform redundancy rate across almost all video frames.* As a result, they tend to over-protect large frames by adding significantly more redundancy than necessary, with median values ranging from 5.2× to 9.2×. Worse still, they under-protect small frames, failing to recover their lost packets and resulting in frequent interaction stalls, occurring more than once per minute. Furthermore, our findings indicate that coarse-grained FECs exacerbate stall frequency in operational cloud gaming systems due to excessive redundancy, which leads to increased network congestion bursts. This, in turn, elevates the LRIF for video frames, particularly for smaller frames, making them more difficult to recover.

In this paper, we aim to address a key problem: *How to follow frame LRIF to ensure successful recovery in small frames meanwhile avoiding wasting redundant bandwidth in*

large frames, so as to propel cost-effective cloud gaming in the real world? To accomplish this, we propose Tooth, a novel fine-grained FEC tailored for cloud gaming at the frame level. It precisely determines the delivery probability of each frame through the frame length distribution, and adaptively applies the per-frame redundancy rates for different frames to align the delivery probability. Although conceptually simple, building Tooth in practice introduces two significant challenges.

Firstly, the realistic LRIF pattern is more complicated than the binomial distribution, and identifying the critical factors that influence the binomial distribution of per-frame LRIF is essential for enabling precise FEC. Our field measurements reveal that both the application-layer frame length and the transport-layer network loss patterns significantly impact LRIF. In particular, the frame length can be readily acquired from the video encoder, while inferring the network loss pattern is a non-trivial task. Traditionally, loss patterns are simply characterized by the loss rate [18,20,27,31,38]. However, our evaluations show that this metric does not accurately depict the distribution of network loss events on a per-video-frame basis. To address this deficiency, we introduce a novel metric–network loss aggregation, which quantifies whether network packet loss events are more dispersed across multiple video frames or concentrated within a few frames, thereby aligning more effectively with fine-grained FEC encoding strategies.

Secondly, effectively mapping LRIF-related factors to per-frame redundancy rate, while ensuring Tooth lightweight enough for practical deployment. In particular, the relationship between per-frame redundancy rate and LRIF-related factors is non-linear. Straightforward mapping methods like linear function or neural network models, cannot guarantee the video QoE and running cost simultaneously (as will be validated in §5.3). Furthermore, the key factors we investigated, such as network loss rate and loss aggregation, are highly volatile due to network dynamics, *i.e.*, only historical network loss pattern cannot reliably reflect future conditions. Therefore, we design a dual-module FEC encoding for Tooth, which consists of *(i)* a *slow-module*, which learns network volatility to estimate the future network loss pattern (loss rate lr and loss aggregation la) based on historical loss patterns and packet-by-packet reception. It utilizes a compressed neural network model to improve prediction accuracy and operates with lower frequency, *i.e.*, once for each network RTCP feedback cycle. *(ii)* a *fast-module*, which formulates the non-linear and discontinuous mapping between LRIF-related factors and redundancy of each frame as a regression problem, and then uses a lightweight random forest method to model the relationship. The *fast-module* runs every time a video frame is encoded, thus ensuring that redundancy is immediately determined and applied to each newly generated video frame. As a result, the above decoupled dual modules avoid implementing frame-level FEC through heavy neural network models and also reduce the execution frequency, which allows Tooth agile enough to operate in real-world scenarios.

We have implemented Tooth in a commercial cloud gaming system, one of the popular cloud gaming platforms (anonymized as Company W), and deployed it in its production environments, involving about 2,300 sessions and lasting 6 weeks in total. Compared to state-of-the-art FEC solutions such as RL-AFEC [18] and RTC-FEC [9,28], Tooth achieves an ideal balance between optimized video QoE and efficient redundancy cost. For instance, it significantly reduces stall frequency by 40.2%-85.2%, increases perceptual video bitrate by 11.4%-29.2%, and reduces bandwidth costs by 54.9%-75.0%. Moreover, we also evaluate Tooth's effectiveness in different *(i)* network types like WiFi, 4G, and 5G; *(ii)* game types like 2D games and 3D games; *(iii)* Internet Service Providers (ISP); *(iv)* distances of game server to end devices, *i.e.*, cross-city sessions and intra-city sessions. The above extensive experiments (§5.2) validate that Tooth can achieve excellent QoE consistently in real-world cloud gaming scenarios. We also conduct a series of ablation studies to verify the key design modules of Tooth.

To summarize, our main contributions are:

- We conduct a large-scale measurement on an operational cloud gaming system and uncover underlying reasons for the inefficiencies of existing loss recovery solutions (§2).
- We propose Tooth, a lightweight dual-module approach that can map the application-layer frame length and transmission-layer network loss pattern to per-frame redundancy in real-time, thus achieving a fine-grained FEC for enhancing cloud gaming streaming (§3).
- We implement and deploy Tooth in the operational cloud gaming system and conduct real-world evaluation, which outperforms state-of-the-art solutions significantly and consistently across environment variation (§5).

2 Background and Motivation

2.1 Background

2.1.1 Packet Loss Recovery

In conventional real-time applications such as video conferencing, playback delays of 200 to 500 milliseconds are typically tolerable [28, 43, 47]. However, in cloud gaming scenarios, stronger interactivity is critical, with significantly stricter latency constraints, *i.e.*, the application-layer interaction latency must consistently remain below 100 milliseconds to ensure optimal user QoE [22–24]. Here we clarify that the application-layer interaction latency refers to the duration between a user's operation input (*e.g.*, run and hit) in the game client and the moment the user receives the corresponding screen feedback. It contains the system processing delay and data round-trip network delay, the measurement method is given in Appendix A. Given the stringent latency requirements of cloud gaming, it significantly influences the design considerations for packet loss recovery. In particular, there are typically two loss recovery mechanisms, which are usually adopted jointly in cloud gaming:

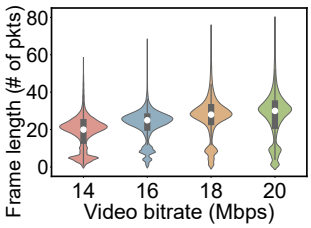


Figure 1: Frame length details at different video bitrates.

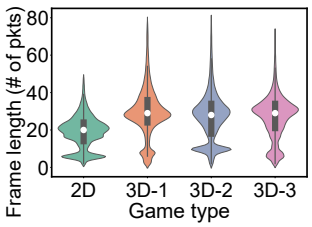


Figure 2: Frame length details in 2D and 3D games.

Feature	Hairpin [36]	Tambur [42]	Xu *et al.* [51]	Ours
User sessions	3,308	9,700	2,430	69,426
Playtime (hours)	600+	-	121.5	31,316
Players	3,308	9,700	1	66,168
Player's City	-	-	1	22
Edge server's City	1	-	1	17
ISP	3	-	-	4
User device models	-	-	1	83
Cloud game products	2	2	3	4

Table 1: Datasets statistics of our large-scale measurements on an operational cloud gaming system, compared with other existing measurement efforts.

- *Forward Error Correction (FEC)* [18, 20, 31, 42], which proactively supplements original data with additional encoded data as redundancy before each transmission. In this case, the lost original data packets will very likely be recovered by the receiver. FEC reduces the delay by not requiring retransmission but increases the bandwidth cost since future packet losses are uncertain.
- *Retransmission* [21, 36]. There are also recent proposals that call for the revitalization of retransmission due to the reduction of RTT. When the packet is lost and detected, the sender will transmit the lost data packets again. This will introduce additional RTT delays to the data delivery, but will only consume the bandwidth of the lost packets.

2.1.2 Variation in Frame Length

A key observation in this paper is the variation of frame length in cloud gaming, for the following two reasons:

- *Rate adaptation.* The available network bandwidth is highly dynamic, sometimes dropping by a factor of 50× in wireless real-time communications [35], so the encoding bitrate will also change accordingly. This is determined by the rate adaptation algorithms such as GCC [17]. Such an adaptive bitrate will result in the variation in frame length (number of packets). We emphasize that the frame length variation here is different from the segment length variation reported in [45], which has a far loose latency requirement.
- *Variable bitrate (VBR).* Meanwhile, due to the internal design of video codecs, even if the bitrate is fixed, the frame length can also vary due to the VBR encoding. For example, under the same video bitrate, the frame length of a static scene will be shorter than that of a dynamic scene.

We conduct a measurement campaign over our operational cloud gaming system (later described in §2.2), and the statistics are shown in Table 1. The video codec we use is NVENC [1, 8] following H.265 [7] and it adopt infinite GOP encoding manner to meet the low-latency transmission characteristics of cloud gaming. Unlike the traditional finite GOP manner sets a fixed GOP length, such as 30 in RL-AFEC [18], infinite GOP disables the server to send I frames periodically, to reduce latency surges, more explanations in Appendix B. The congestion control algorithm is GCC [17]. We present the distribution of frame lengths when the encoding bitrates are fixed to different levels in Fig. 1, and this length difference exists in both 2D and 3D cloud games, as shown in Fig. 2.

2.2 Inefficacy of Existing Solutions

Unfortunately, when deploying FEC and retransmission into production-scale systems, we find that existing solutions suffer from a fundamental trade-off between bandwidth cost and interaction latency.

Measurement Setup. We conduct a large-scale measurement campaign over our operational cloud gaming system. The measurement covers 22 cities across China and 4 ISPs, with other statistics shown in Table 1. It is worth noting that the dataset we collect is the largest dataset in the field of cloud gaming to the best of our knowledge.

FEC algorithms often over-protect data packets with drastically high bandwidth costs. An interesting observation is that many existing FEC solutions often add much more redundant packets than necessary. For example, when there are 5 packet losses out of 25 data packets, ideally only 5 additional redundant packets will protect the data packets. However, existing solutions [18, 36, 42] often add much more than 5 packets. This reason is quite intuitive – more redundancy is added to get prepared for unforeseen packet losses, however, which leads to drastic bandwidth waste. Especially considering that bandwidth cost is the major contributor to the operational expenses for streaming applications such as cloud gaming, this will directly lead to a revenue loss [2, 3, 42].

We validate the observation by deploying two representative FECs in our cloud gaming platform:

- *RTC-FEC* [28], the de-facto rule-based solution in WebRTC [9] based on the current video bitrate and network loss rate.
- *RL-AFEC* [18], a reinforcement learning driven FEC solution. It defines the first I frame and the following K-1 P frame of one GOP (length is N) as the K "critical" frames[1], and the remaining N-K P frames as "non-critical" frames. RL-AFEC finds that non-recoverable critical frames have a more serious negative impact on video quality, so it sets different redundancy rates[2] for the critical frames of each

[1] In *RL-AFEC* [18], there are 6 candidate values for K, *i.e.*, 5, 10, 15, 20, 25, and 30. Here we use the recommended optimal value, *i.e.*, K = 15.

[2] Here we set 10 candidate redundancy rates for RL-AFEC, *i.e.*, 10%, 20%,...100%, completely consistent with [18].

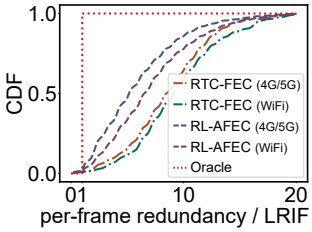


Figure 3: CDF of the ratio of per-frame redundancy rate to LRIF for coarse-grained FEC.

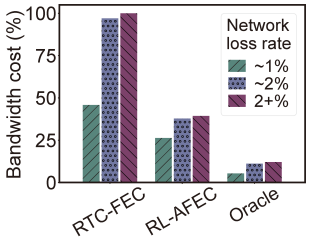


Figure 4: Bandwidth costs of the representative FECs and the ideal Oracle solution.

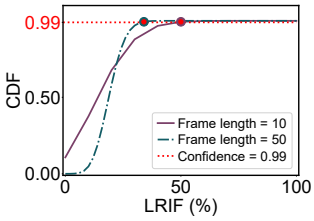


Figure 5: The CDF of video frame LRIF binomial distributions with different numbers of trials.

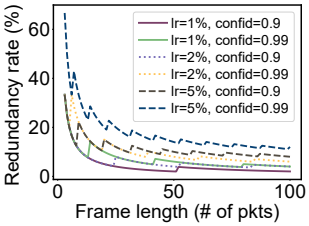


Figure 6: Lines: when the loss rate (lr) is 1%, 2%, and 5%. Y-axis: the redundancy rate. X-axis: the frame length.

GOP. As for the remaining non-critical frames, it still uses the same uniform redundancy rate.

As shown in Fig. 3, both algorithms add much more redundant packets to frames than necessary, with a median number of 5.2×-8.6× in 4G/5G networks and 6.6×-9.2× in WiFi networks, resulting in serious bandwidth waste. In Fig. 4, we show the overall redundancy bandwidth costs of two representative FECs, both are very high, *i.e.*, 45.9%-100% and 26.3%-39.4% respectively. To show the potential optimization space, we compare the existing FECs with an "Oracle" approach, *i.e.*, we count the actual data loss rates of all video frames as the ideal bandwidth cost. As shown in Fig. 4, only 5.4%-12.1% redundancy is required.

Retransmission-oriented solutions still suffer from severe stalls. Including the recent effort, Hairpin [36], there has been an understanding that since the RTT is low enough, merely relying on retransmission might be sufficient to recover the lost packets. However, our experiments show the contrary:

- The network RTT measured in our cloud gaming system is 23 ms for WiFi users and 35 ms for 4G/5G users. This is because as the scale of the product expands, the users with worse network conditions are naturally included.
- The detection of packet loss also takes time in the commonly used negative acknowledgement (NACK) methods. For example, in WebRTC, it will take the receiver up to 20 ms before sending NACK through RTCP protocol.
- Affected by the congestion control and pacing rate of the sender, the transmission duration of video frame data packets often exceeds 10 ms.

Therefore, considering the stringent deadline requirement of around 100 ms [22–24, 36, 37, 48], it is still insufficient to rely on retransmission to recover the lost packets, and the stall frequency even reaches 41.9 times per minute. We will provide the detailed experimental results in §5.2.

2.3 Observation – Frame length matters

In Fig. 3, we have shown that existing FEC algorithms often over-protect many video frames while under-protecting some others (the ratio < 1), leading to both bandwidth waste and interaction stalls. In this subsection, we will show that the root cause of this inefficiency is the neglect of the frame length in the FEC algorithms. For the sake of clarity, we first denote the loss rate in each frame as LRIF, which is defined as $\frac{\#\ of\ lost\ packets\ in\ frame}{\#\ of\ all\ packets\ in\ frame}$. We later have the following findings:

Finding 1: The distribution of video LRIFs vary significantly with the frame length. As we all know, network loss events are random and bursty [33, 36, 42]. Now let's begin with a simple probability question: assume that the loss rate of each packet is 10% and they are independent from each other, if we want to protect the delivery of the frame with the confidence of 99%, how many redundant packets do we need to add when (a) the frame has 10 data packets; and (b) the frame has 50 data packets? In this case, we need to calculate the per-frame LRIF as it is the necessary redundancy rate to be set. The LRIF in this example follows the Binomial Distribution [11], with the probability of each trial being 0.9, and we want the minimum number of trials such as $P(X \geq framelength) \geq 0.99$. The answer is: when the frame length is 10 packets, we need to add 4 redundant packets; when the frame length is 50 packets, we need to add 12 redundant packets. The redundancy rate for smaller frames (40%) is much higher than that for the larger frames (24%)[3].

The reason behind this phenomenon is a basic law in probability theory – the law of large numbers. As shown in Fig. 5, when the number of trials (in our case, frame length) increases, it requires a lower redundancy rate to successfully recover 99% of all frames. Therefore, the redundancy rate needed to protect the frame will decrease as the frame length increases. Here we clarify that a lower redundancy rate does not directly represent adding fewer redundant packets. For a video frame, the ideal redundancy rate is its LRIF and the ideal number of the redundant packets is $framelength \times LRIF$. Next, we further simulate the necessary redundancy rates with different confidence levels and frame lengths, and the results are shown in Fig. 6. This phenomenon is general – the redundancy rate for small frames is much higher than that for large frames.

Finding 2: Existing FEC algorithms ignore the LRIF variation caused by frame length, either over-protect large frames or under-protect small frames. We delve anew into existing FEC algorithms and find the underlying

[3] One can validate the result at https://stattrek.com/online-calculator/binomial.aspx. The parameters we use are (0.9, 14, 10) and (0.9, 62, 50).

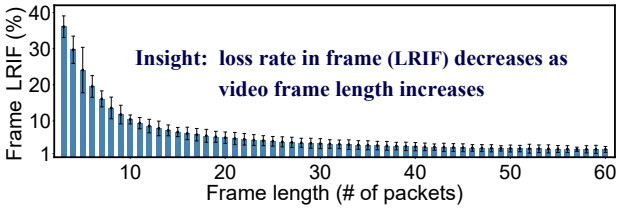
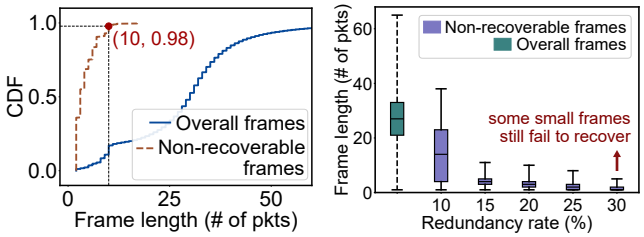
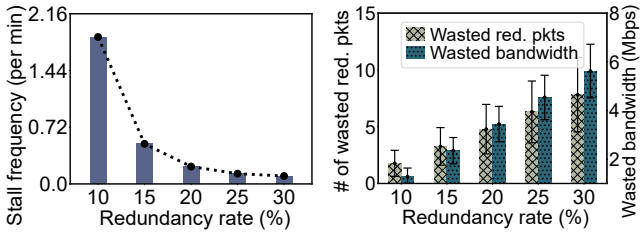


Figure 7: Relationship between video frame length and LRIF. The statistical results come from video frames that suffered packet loss in our commercial cloud gaming platform. Only these frames can reflect the optimal redundancy rate to protect each video frame, more explanations in Appendix D.



(a) Recovery of video frames when redundancy rate is 20%.

(b) Recovery of video frames under different redundancy rates.



(c) Stall frequency under different redundancy rates.

(d) Wasted redundancy (original streams are average 19.3Mbps).

Figure 8: Video frame recovery and wasted bandwidth under coarse-grained redundancy rates. The results are obtained from the controlled analysis to FEC. We replay real-world stream traces in the simulator and don't consider the damage to network condition caused by excessive data transmission.

reason: most of them are coarse-grained, ignoring the impact of frame length, and set a uniform redundancy rate r (defined as $\frac{redundant\ data}{original\ data}$) for all frames based on network conditions. While RL-AFEC [18] proposes setting varying redundancy rates for each of the first K critical frames in one GOP, it overlooks the remaining N-K non-critical frames. This is especially unsuitable in the context of cloud gaming, where N-K non-critical frames can comprise up to 99.9% of all frames in one GOP (details in Appendix B). Thus the vast majority of frames will still be set the uniform redundancy rate, even though these frames behave with significant LRIF variations. Therefore, all existing FEC methods tend to set a higher redundancy rate, as we presented above. Unfortunately, even if the redundancy rate is high enough for most frames, it might still be insufficient for small frames. For example, in the line of (lr=10%, conf=99%) in Fig. 5, when setting the redundancy rate to 40% (4× the average loss rate), frames

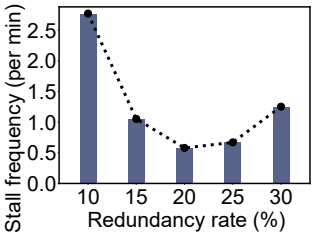


Figure 9: As redundant rate increases, the over-sent redundancy even exacerbates interaction stalls.

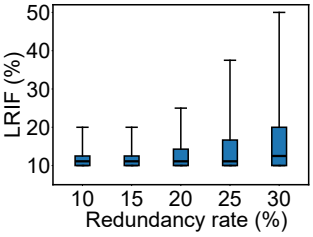


Figure 10: As redundant rate increases, the over-sent redundancy exacerbates small frames' LRIF.

smaller than 10 packets are still likely to fail to recover even though for frames larger than 10 packets the redundancy will be significantly wasted. We further validate this observation by our online measurements as well. As shown in Fig. 7, as the frame length increases, the LRIF mean of the frames that suffered packet loss decreases significantly. This indicates that large frames will easily be over-protected while small frames will be under-protected by the coarse-grained FECs.

The adverse impact of coarse-grained FECs on operational cloud gaming systems. Here we demonstrate how existing coarse-grained FECs impact the performance of cloud gaming in practice, for which we use all traces as listed in Table 1. Firstly, we apply a uniform redundancy rate of 20% to each gaming stream and observe the frame recovery. As shown in Fig. 8a, the non-recoverable frames are relatively small, *i.e.*, 98% of them are small frames (length ≤10). Furthermore, we also apply other 4 redundancy rates to each video stream in turn, *i.e.*, 10%, 15%, 25%, 30%. Fig. 8b plots the recovery results under all five redundancy rates. We can observe that: increasing the redundancy rate can indeed recover more video frames, but some small frames still fail to be recovered, which requires a higher redundancy rate, exceeding 30%. To examine the impact on QoE, we present the interaction stall frequency in Fig. 8c and the wasted redundancy in Fig. 8d. Obviously, the stall frequency decreases along with the increase of redundancy rate, but the marginal effect diminishes very fast, *e.g.*, we barely have extra gain when the redundancy rate rises from 25% to 30% but the wasted bandwidth keeps increasing substantially, from 4.5 Mbps to 5.6 Mbps.

More importantly, in the real world, *our measurement results show the counter-intuitive fact: more redundancy instead exacerbates cloud gaming's interaction stall*. Specifically, we measure the interaction gains of FEC under different redundancy rates of 10%, 15%, 20%, 25%, 30%. As shown in Fig. 9, when the redundancy rate is relatively low, *i.e.*, 10%, 15%, 20%, FEC can continuously reduce the interaction stall frequency. However, as redundancy rate keeps increasing, FEC fails to further optimize and even worsens the stall performance. We investigate the traces and find the root reason: adding too much redundancy will lead to more bursting network congestion, thus the LRIF of a video frame especially a small frame will be even higher, as shown in Fig. 10.

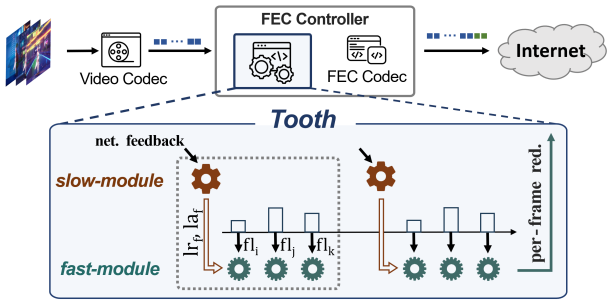


Figure 11: Design overview of Tooth.

To sum up, existing FEC algorithms are commonly coarse-grained and ignore the LRIF differences across video frames. Hence they are trapped in the dilemma of improving frame recovery and avoiding bandwidth waste. Nowadays, cloud gaming applications are becoming more bandwidth-demanding (with higher image quality), it is more imperative to break the dilemma with a novel FEC design.

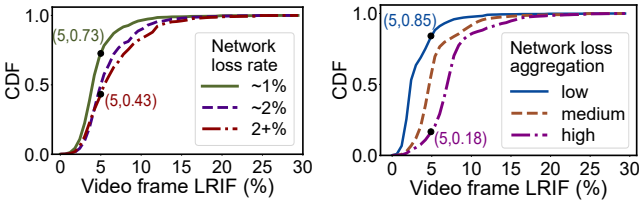
2.4 Design Challenges

Although we have revealed the significant impact of frame length on per-frame redundancy rate, adapting it to achieve fine-grained FEC is non-trivial, which faces 3 key challenges: *(i)* Besides frame length, LRIF also intertwines with the transport-layer network loss pattern. Identifying the critical factors that influence the per-frame redundancy rate is essential for enabling precise FEC at the fine-grained level. *(ii)* The relationship between per-frame redundancy rate and the related factors is non-linear and discontinuous, making it difficult to represent using a straightforward mapping function. Meanwhile, network loss pattern fluctuates, meaning the historical observations cannot reliably reflect the future network conditions. *(iii)* Fine-grained FEC necessitates immediate determining and executing redundancy rate upon generating a new video frame (*e.g.*, 16.7 ms at FPS=60). The high frequency and stringent time constraints make existing learning-driven approaches impractical.

3 Design

3.1 Design Overview of Tooth

We first perform the field measurements to identify critical factors influencing the per-frame redundancy rate, including application-layer frame length and transmission-layer network loss pattern. Notably, in addition to the loss rate introduced in the conventional approaches [16], we based on the FEC operation to supplement loss pattern with loss aggregation, which further describes the temporal distribution of network packet loss events (§3.2). Then, we design Tooth as a decoupled dual-module architecture (Fig. 11), which can map the LRIF-related factors to per-frame redundancy rate and meanwhile minimize computational and inference time overheads. It consists of a *slow-module* and a *fast-module*, which circumvents the need for bulky large neural networks but



(a) Network loss rate *lr*. (b) Network loss aggregation *la*.

Figure 12: Other related factors affecting video frame LRIF. Fig. 12b is obtained by analyzing the data with a network loss rate of ~1% in Fig. 12a. The network loss rate is analyzed from the RTCP feedback (every 100ms in our system).

also lowers the execution frequency, effectively tackling the computational complexities and inference time constraints in real-world environments. Specifically,

- *Slow-module* utilizes a compressed neural network model to discern network fluctuations and estimate future loss pattern. It does not need to execute frame by frame but is activated only upon receiving new RTCP network feedback to avoid unnecessary computational overhead (§3.3).
- *Fast-module* incorporates an exceedingly lightweight machine learning model to determine per-frame redundancy based on the frame length and future network loss patterns obtained from the *slow-module* (§3.4).

3.2 Investigating the Impact Factors of LRIF and Evaluating Strawman Solutions

We make an in-depth analysis of the large-scale measurement dataset in §2.2, and find that *besides the frame length, transport-layer network loss rate (lr) and loss aggregation (la) also significantly influence over video frame LRIF.* Next, we give more analysis: *(i)* in Fig. 12a, we plot the CDF of the video frame LRIFs related to network loss rate *lr*. The result clearly demonstrates a trend where frames experience increasing LRIF as the network loss rate escalates. For instance, when *lr* is 1%, 73% of the frames have LRIFs below 5%. However, when *lr* increases to 2% or higher, only 43% of the frames maintain LRIFs below 5%, marking a significant difference of 30%. *(ii)* Previous study [42] has noted that continuous packet loss severely impacts video frame recovery. However, we argue that for advanced FEC methods like Reed-Solomon codes [41], there is no distinction between losing n packets continuously or discretely within a single frame. Therefore, we introduce a novel metric, *i.e.*, network loss aggregation *la*, to reflect whether packet loss events will be more dispersed in many frames or concentrated in a few frames. *la* can be represented as follows,

$$la = \begin{cases} 0 & n \le 1 \\ \dfrac{n}{\sum^{n} |t_n - centroid| + \alpha} & n > 1 \end{cases} \quad (1)$$

Here n represents the number of packets lost within a network feedback period, and t_n denotes the ideal arrival time of the n^{th} lost packet, measured in milliseconds. We define the timing

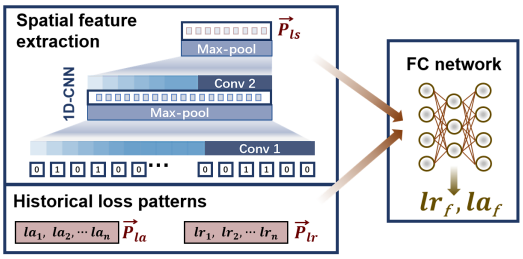


Figure 13: *Slow-module*'s model design.

centroid of all lost packets as $\frac{\sum^n l_n}{n}$, and α is set to 0.5 to avoid $\sum^n |t_n - centroid|$ being 0, which means packets are lost within 1ms. To intuitively illustrate the impact of la on LRIF, we briefly divide all la values into 3 levels according to the numerical distribution (*i.e.*, low: 0-1, medium: 1-5, high: 5-max) and categorize the video LRIFs. As shown in Fig. 12b, elevated la will increase the video frame LRIF. For instance, when la is low, 15% of frame LRIFs exceed 5%, but when la is high, this jumps to 82% of frame LRIFs exceed 5%.

Strawman solutions for fine-grained FEC. To assess the impact of video frame length (fl), network loss rate (lr), and loss aggregation (la) on per-frame redundancy rate, one preliminary approach is to manually establish the mapping relationship. However, this task proves to be non-trivial due to several challenges: *(i)* The relationships among the three factors and per-frame redundancy are not straightforwardly linear. Instead, they are implicit and complex, making manual modeling difficult. *(ii)* Both lr and la are subject to continual changes due to network fluctuations. Notably, there is also a dependent relationship where an increase in lr can sometimes lead to a higher la, though not invariably. Thus, it is imperative for Tooth to accurately determine per-frame redundancy amid varying network conditions.

Another strawman idea is to leverage powerful neural network models [18–20, 31]. Existing approaches generally operate at a coarser granularity, typically making decisions every 100 ms or longer [32, 42], which is inadequate for Tooth's fine-grained frame-by-frame optimization. On one hand, Tooth necessitates frequent redundancy execution within the tight frame encoding period of 16.7 ms at a frame rate of 60 FPS. On the other hand, Tooth must maintain exceptionally low overheads in both computation and inference time to enable feasible deployment in production environments. However, even neural network models with optimized architectures and parameters, such as pruning, often fail to satisfy these stringent overhead requirements, which will be validated in §5.3.

3.3 *Slow-module*: Estimating Future Network Loss Pattern

Slow-module is tasked with learning network fluctuations to estimate future network loss rate lr_f and loss aggregation la_f, and update them to the *fast-module*. Obviously, estimating la is very challenging since it is potentially affected by lr in addition to its own intrinsic characteristics. Existing simple methods, like arithmetic average of the heuristic observations, will bring serious deviations to the judgment of per-frame redundancy, as verified in §5.3. Thus, we introduce a NN model to build *slow-module*, and we provide more details next.

Input and output. As illustrated in Fig. 13, *slow-module*'s input is the historical network loss pattern vectors: the network loss rate sequence in the past n RTCP feedback, *i.e.*, $\vec{P_{lr}} = (lr_1, lr_2, ..., lr_n)$, the loss aggregation sequence ($\vec{P_{la}} = la_1, la_2, ..., la_n$), and the spatial distribution features $\vec{P_{ls}}$ of historical packet loss events. $\vec{P_{ls}}$ is extracted from the packet loss events in the past n network feedback periods. In particular, we encode the arrivals of all N packets in the past n RTCP feedback periods into a 0, 1 vector $\vec{ls}$, *i.e.*:

$$\vec{ls} = (x_1, x_2, ..., x_N), \quad x_i = \begin{cases} 1 & \text{packet lost} \\ 0 & \text{packet received,} \end{cases} \tag{2}$$

Here we build a 1D-CNN block to process $\vec{ls}$, which consists of two cascaded one-dimensional convolution layers with kernel sizes of 100 and 4, respectively. Each convolution layer in the 1D-CNN module uses *ReLU* activation and a pooling layer with a kernel size of 2 to extract the spatial features $\vec{P_{ls}}$.

The output of Tooth's *slow-module* is the future network loss rate (lr_f) and loss aggregation (la_f).

Neural network model. We pass the concatenated sequences of $\vec{P_{lr}}$ (10×1), $\vec{P_{la}}$ (10×1), and $\vec{P_{ls}}$ (12×1) to the *slow-module*'s cascaded FC networks, with 64, 32 and 16 units respectively. For the loss function, we emphasize that the overestimated lr_f and la_f will guide Tooth add excessive redundancy, increasing bandwidth cost. But the underestimated future lr_f and la_f may lead to insufficient redundancy, thus incurring stalls, more unacceptable than wasting bandwidth. Therefore, we set a larger loss for the underestimated case and the final loss function $\mathcal{L}_S$ is defined as follows:

$$\mathcal{L}_S = \frac{(lr_f - \hat{lr})^2 + (la_f - \hat{la})^2}{2} \times \left(\frac{e^{(\hat{lr} - lr_f) + (\hat{la} - la_f)}}{|\hat{lr} - lr_f + \hat{la} - la_f| + \beta}\right), \tag{3}$$

where $\hat{lr}$ and $\hat{la}$ are the true values of network loss rate and loss aggregation. The first part calculates the difference between the estimated lr_f, la_f and the true $\hat{lr}$, $\hat{la}$. For the second part, if lr_f, la_f are larger than $\hat{lr}$, $\hat{la}$, the loss can be appropriately reduced and when lr_f and la_f are smaller than $\hat{lr}$, $\hat{la}$, the loss will be enlarged. β is a constant and is set to 1 to avoid $|\hat{lr} - lr_f + \hat{la} - la_f|$ being 0.

3.4 *Fast-module*: Determining Per-frame Redundancy

Fast-module takes the future network loss rate lr_f, loss aggregation la_f, frame (to be encoded) length fl as inputs, and outputs the per-frame redundancy decision. In crafting *fast-module*, we gather lr, la, fl, and *LRIF* from historical frames as label to build sample dataset. Next, we incorporate a machine learning approach different from complex NN models to reduce the computation complexity.

Incorporating Random Forest model. We clarify *fast-*

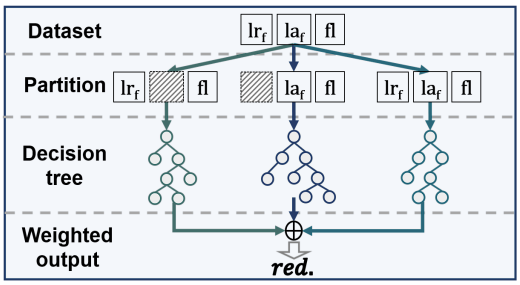


Figure 14: *Fast-module*'s model design.

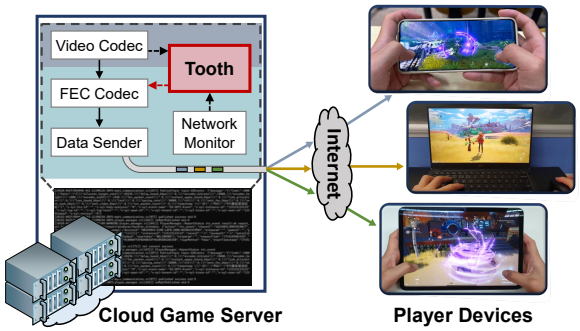


Figure 15: The implementation of Tooth.

module's inputs (lr, la, fl) exhibit the following characteristics: *(i)* High cardinality, especially lr and la boast hundreds of non-continuous potential values, while fl may also have dozens, thus rendering the feature space inherently variable and intricate. *(ii)* These features exert distinct and non-linear impacts on per-frame redundancy. For instance, a marginal 1% increase in lr may result in the loss of several more packets within a frame, whereas a similar increase in fl may not yield a substantial damage. In light of these issues, we leverage Random Forest (RF) model to construct *fast-module* as multiple LRIF decision trees. On one hand, RF model can effectively handle high-cardinality samples. Each redundancy decision tree of RF model will be built by randomly selecting training samples, ensuring strong generalization ability. On the other hand, each redundancy decision tree will split its nodes based on different input features during the establishment process, enabling to learn the degree to which each feature affects the per-frame redundancy. As shown in Fig. 14, we first partition the dataset based on lr,la, and fl, while ensuring that the elements in each dataset are evenly distributed to avoid situations where a large amount of data is concentrated near a certain value. Subsequently, we adjust the features carried by each subset, and the explicit features carried by each subset are divided into (lr, fl), (la, fl), and (lr,la, fl). This strategy prevents the decision trees from overly relying on a certain feature. Moreover, we choose MSE (Mean Squared Error) as the *criteria* of RF model to train the *fast-module*.

The final redundancy decision of the RF model is taken as the average of all decision trees' outputs. We emphasize that *(i)* the combination of multiple decision trees can help *fast-module* learn the non-linear relationship between lr, la, fl, and per-frame redundancy. *(ii)* Each tree examines distinct features and autonomously assesses the significance of each input feature, culminating in the integration of multiple decision trees' outputs. This can significantly enhance *fast-module*'s ability to generalize across varied network conditions and gaming environments, as validated in §5.2.

4 Implementation

Tooth's implementation workflow. As the workflow illustrated in Fig. 15, Tooth serves as a pivotal control module of the game server, positioned between Video Codec and FEC Codec. For each gaming session, Tooth acquires the receipt information of past data packets from Network Monitor and the frame length from Video Codec, subsequently controlling FEC Codec to add redundancy for each newly generated frame. In terms of data flow, the server generates the real-time game content, which is encoded by Video Codec and FEC Codec. Then, Data Sender transmits the data packets to players' devices (*e.g.*, smartphone, laptop, iPad) via the Internet.

Tooth's dual-module training methodology. In §2, we have collected extensive cloud gaming traces from the production environment, which records *(i)* transmission and reception details of transport-layer RTP packets. From which, we count the network loss rate and aggregation at a granularity of 100ms to construct the network loss pattern dataset; *(ii)* Each packet's sequence number and the associated video frame ID, locations of all lost packets. For each frame, we count its length, number of lost packets, and integrate its current network loss pattern to construct the video frame LRIF dataset. Based on these two datasets, we complete the training of Tooth.

System implementation details. *(i) Tooth's integration.* Our commercial cloud gaming system has been running online for nearly two years and has more than 100,000 players. It follows the RTP protocol stack [44] and is implemented based on WebRTC [9, 10]. On game server, there is a real-time streaming service (RTSS) module, responsible for handling the game video content and streaming it to the client. Tooth is integrated as an RTSS sub-module, embedded into RTSS's media and network stack to *1)* obtain the frame information and network information *2)* return to RTSS main process the redundancy rate decision. *(ii) Redundancy encoding method.* We leverage the Reed-Solomon (RS) codes [41] to build FEC Codec. Through our engineering refinement, it empowers an FEC block to handle up to 256 packets and the encoding time consistently remains below merely 1 ms. Also, we use WebRTC's FlexFEC codec to support existing FEC solutions. For protection of uplink client-to-server data flow, see Appendix C. *(iii) Network Monitor* discerns the RTP packets' arrival via the client's RTCP feedback (per 100 ms) then provides it for Tooth as the basis for FEC decision-making. Moreover, it calculates the frame LRIFs and archives comprehensive network metrics (*e.g.*, RTT, throughput) into the trace database based on the MQTT protocol [4]. These massive network traces facilitate continuously optimizing Tooth in the offline environment.

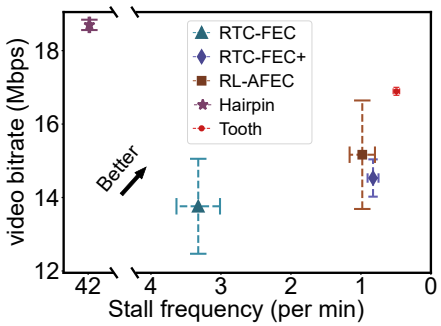


Figure 16: Interaction QoEs of all baseline methods. Tooth exhibits the optimized performance in stall frequency and video bitrate.

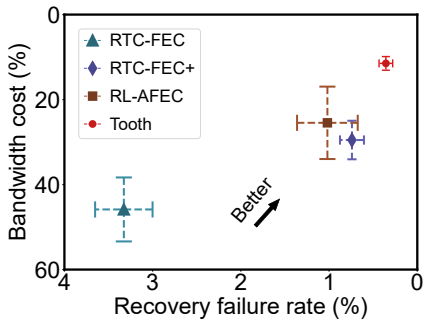


Figure 17: Redundancy rate and recovery failure rate of different FEC methods. Tooth recovers more frames with less redundancy.

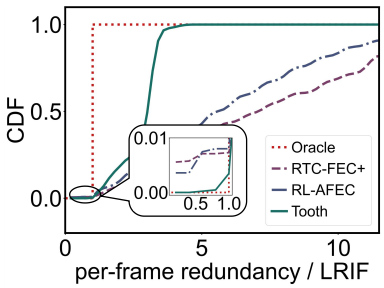


Figure 18: CDF of the ratio of per-frame redundancy rate to LRIF. A ratio of 1 means FEC adds the most appropriate redundancy to a frame.

5 Evaluation

5.1 Experimental Methodology

Comparison baselines. Here we compare Tooth with four state-of-the-art loss recovery baselines:

- *RL-AFEC* [18], which we have already introduced in detail in §2.2. Here we maintain its K as the recommended optimal value of 15 in [18] and provide the further ablation experimental results for other K values in the Appendix E.
- *RTC-FEC* [28], as the default FEC solution of WebRTC framework [9], is widely used in the industry. We integrate the m119 release from Chromium [10] in 2023 into our cloud gaming system.
- *RTC-FEC+*, which replaces *RTC-FEC*'s FlexFEC codec with more powerful RS codec [41]. We also tune *RTC-FEC+*'s redundancy table to suit the RS codec.
- *Hairpin* [36], which is the latest effort in interactive video streaming. It noticed that the loss of retransmission packets will bring multiple RTT overheads and ultimately incur stalls. So it adds redundancy to retransmission packets.

Evaluation metrics. For each cloud gaming session, we log the following metrics to conduct a comprehensive evaluation.

- Our direct optimization goals, *i.e.*, FEC performance, including *1)* bandwidth cost from FEC redundancy; *2)* recovery failure rate, which is the percentage of video frames that fail to recover when suffering packet loss. We clarify that adding redundancy will lower the video bitrate because production systems often have a bitrate cap (20 Mbps in our case) to control overall bandwidth costs, which is a common industry practice. Moreover, when FEC fails to recover a video frame, retransmission is used to ensure the data integrity of non-recoverable frames (whether I frame or P frame). Otherwise, the subsequent video frames won't play normally, leading to interaction stalls.
- Player interaction QoEs, including *3)* stall frequency, *i.e.*, how many times a player experiences stall per minute. We consider an interaction latency of more than 100ms as a stall, for which players will feel that the game is laggy; *4)* video bitrate, which does not include redundancy, represents better picture quality when it is higher; *5)* video PSNR and VMAF, which are close to the user-perceived feedback and have been widely adopted in the community and industry [18, 19, 31, 48].

To avoid the impact of differences in gaming session duration, we calculate these metrics at a one-minute granularity.

Field deployments. We deploy Tooth and baseline solutions to a production server, and conduct rigorous A/B tests in the real world. To ensure fair comparisons, we randomly apply an FEC solution to a gaming session instead of relying on manual allocation, and all solutions operate within the consistent system environment. The evaluation spans a duration of 6 weeks, and the players are in various activity scenarios (*e.g.*, library, office, cafe, home, park) with various wireless network conditions (*i.e.*, WiFi, 4G, 5G). We remove the first minute of each gaming session, during which the game is in the initial animation phase with no user interaction, and the system only monitors the user's network loss conditions.

5.2 System-level Evaluation

Tooth significantly enhances interaction QoE. Here we first focus on the two key application-layer QoE metrics, *i.e.*, stall frequency and video bitrate, to explore Tooth's QoE gains. As shown in Fig. 16, Tooth achieves the lowest stall frequency, *i.e.*, only 0.49 times per minute, reducing that of baseline FECs by 40.2%-85.2%. Furthermore, Tooth achieves the highest video bitrate, *i.e.*, 17.8 Mbps, which is 11.4%-29.2% higher than the baseline FECs. As a result, Tooth can enhance the state-of-the-art FECs' QoE in terms of both stall frequency and video bitrate. Next, we delve into all FECs' bandwidth costs and frame recovery results to further elucidate. As depicted in Fig. 17, Tooth can surpass existing FECs in reducing their bandwidth costs by 54.9%-75.0% while reducing the recovery failure rate by 51.9%-89.3%. This is attributed to Tooth sets higher redundancy rates for small frames to prevent recovery failures and lower redundancy rates for large frames to reduce unnecessary bandwidth costs. Since there are fewer small frames, the redundancy saved by large frames far exceeds the redundancy increased by small frames.

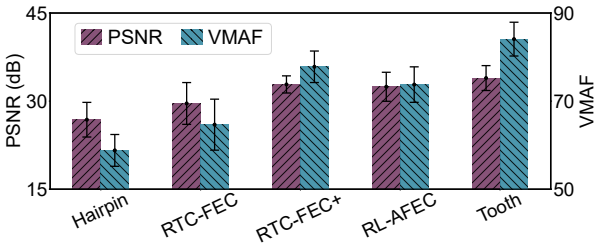
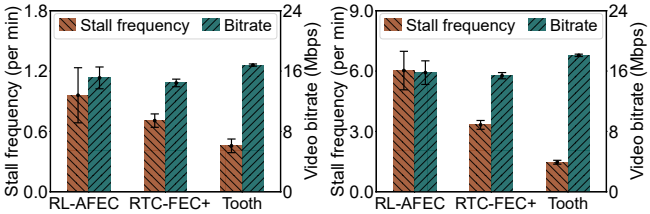


Figure 19: Video PSNR and VMAF results of different loss recovery methods.



(a) WiFi networks. (b) 4G/5G networks.

Figure 20: Interaction QoE under different network types. Tooth can decrease stall frequency more in 4G/5G networks.

To further clarify this, we calculate the ratio of per-frame redundancy rate and LRIF, where a ratio of 1 means that FEC adds the most appropriate redundancy to a frame. As shown in Fig. 18, Tooth is closest to the Oracle solution.

Regarding other baselines: *(i)* RTC-FEC, constrained by FlexFEC codec (mentioned in §5.1), behaves with a 45.9% bandwidth cost and 3.3% frame recovery failure rate. Consequently, RTC-FEC demonstrates a QoE of 3.3× stall frequency and a video bitrate of only 13.8 Mbps, inferior to Tooth. *(ii)* RTC-FEC+, optimizes RTC-FEC's bandwidth cost and frame recovery failure rate by 35.6% and 77.8%, respectively, thus improving QoE. *(iii)* RL-AFEC achieves a 13.7% lower bandwidth cost than RTC-FEC+ via reinforcement learning, but with a 35.2% degradation in recovery failure rate. These two competent coarse-grained FECs, RTC-FEC+ and RL-AFEC, can only fall into the inherent trade-off between saving bandwidth cost and ensuring recovery. *(iv)* For the retransmission-based Hairpin, its stall frequency is as high as 41.9 times per minute due to the additional RTT delays, despite improving video bitrate. Although Hairpin aims to recover all lost packets with only one round of retransmission, even this brings an additional RTT delay, which in our system will cause the interaction latency to be noticed. In the following sections, we will not delve further into the codec-limited RTC-FEC and retransmission-based Hairpin because they fail to meet the bitrate or stall frequency demands.

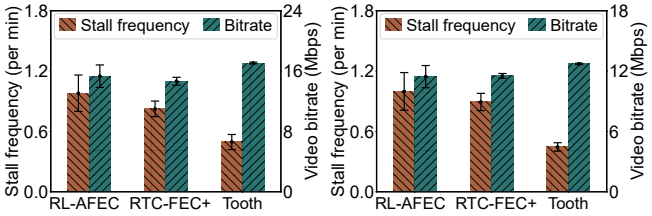
Tooth exhibits better video PSNR and VMAF performance. Note that video PSNR and VMAF require frame-by-frame comparison between the server-side source video and the client-side received video, which is not yet feasible in our current commercial platform. Therefore, we recruit 20 volunteers across campus to conduct a small-scale evaluation [4]. Specifically, we first build a local testbed using our commercial cloud gaming system, and randomly select 20 network traces (one per volunteer) from our network dataset to simulate the real network environments. Then, every volunteer is asked to initiate 5 cloud gaming sessions, with each session lasting 10 minutes and using different packet loss recovery methods. Finally, we collect 100 groups of gaming session videos and use the FFmpeg tool [6] to calculate their PSNR and VMAF results. As shown in Fig. 19, Tooth outperforms the baseline methods, with a 3.3% higher video PSNR and an 11.9% higher video VMAF than the next best RTC-FEC+.

Tooth is more effective in 4G/5G networks. Players on our game platform come from two network types, *i.e.*, WiFi and cellular[5], which represent different network conditions. WiFi networks exhibit lower RTT than 4G/5G networks, *i.e.*, 22.9ms vs. 35.1ms in average, which means the retransmission packets are more likely to timeout to incur stalls if FEC fails to recover the frames. Moreover, WiFi's network loss rate is 1.15% and 4G/5G's is 0.76%, *i.e.*, fewer frames will suffer packet loss in 4G/5G networks. In Fig. 20, we give the QoE results under these two networks. We conclude that: *(i)* From WiFi to 4G/5G networks, all three FECs' stall frequencies increase obviously due to higher RTT. For example, Rl-AFEC's stall frequency increases from 1.02 to stunningly 6.03. But Tooth can achieve more QoE gains, from reducing the baseline FECs' stall frequency by 35.5%-52.4% on WiFi to 55.9%-75.6% on 4G/5G. *(ii)* Due to the lower loss rate, RL-AFEC and RTC-FEC+ tend to decrease the redundancy rate, with their video bitrates increasing by 4.6% and 6.7%. Tooth can achieve a more significant bitrate increase of 7.7%. As proved in Fig. 7, the LRIF of large frames gradually approaches the network loss rate. So when the network loss rate greatly reduces, there are more bandwidth can be saved in large frames for Tooth, but which is something that uniform redundancy rate of coarse-grained FECs cannot achieve.

Tooth can still achieve significant QoE gains in 2D games whose frame length differences are smaller. As mentioned in Fig. 2, the frame length difference of 2D games is smaller overall than that of 3D games, with the maximum and minimum values being 50 and 80 respectively. An intuitive worry is: does Tooth still have QoE gains in 2D games? We break down the interaction QoE results of 3D games and 2D games to clarify the issue, as shown in Fig. 21. For 3D games, Tooth can achieve 39.9%, 49.5% lower stall frequency and 11.4%, 16.4% higher video bitrate than RTC-FEC and RL-AFEC, respectively (Fig. 21a). More importantly, for 2D games, Tooth still shows significant QoE gains over the baselines (Fig. 21b). In details, RL-AFEC and RTC-FEC+ both show slightly worsened stall frequency due to more small frames in 2D games,

[4]For campus volunteers, we only collect their operation instructions and video content from the cloud game client. This data collection process doesn't involve any user privacy information and has received approval from the ethical review committee.

[5]Our game APP cannot obtain detailed 4G and 5G access information from player devices, so they are unified into cellular players. In addition, the number ratio of WiFi players to cellular players is almost 8.5:1.



(a) 3D games. (b) 2D games.

Figure 21: Tooth can not only benefit 3D game but also 2D games whose frame length differences are smaller.

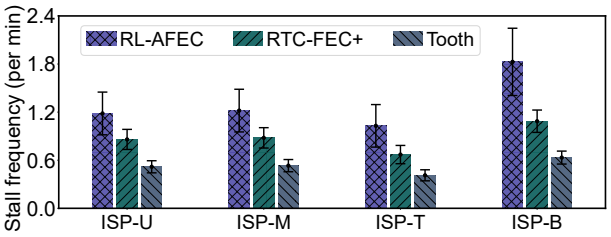


Figure 22: Tooth can achieve consistently lower stall frequency than baseline FECs under different ISP networks.

from 0.82 to 0.89 and from 0.97 to 0.99, respectively, while Tooth can further decrease the stall frequency from 0.49 to 0.44. In terms of video bitrate, Tooth still achieves considerable improvement, 10.4% and 11.1% higher than the baseline FECs. It can reduce the stall frequency by 76.2%, 77.8% and increases the video bitrate by 7.8%, 12.5% (Fig. 21b). Because in both 3D games and 2D games, frequent operations and dynamic game content will lead to inevitable variations in frame length. Thus, Tooth can exploit this difference to adaptively add appropriate redundancy to each frame.

Tooth exhibits exceptional stall performance in various ISP networks. Our cloud gaming platform encompasses players from all four major ISPs in China, *i.e.*, ISP-U (China Unicom), ISP-M (China Mobile), ISP-T (China Telecom), and ISP-B (China Broadnet), with player ratios of 1.1:3.1:1.8:0.01 respectively. Despite ISP variations, video bitrate gain remains largely unaffected, thanks to consistent CDN egress bandwidth set for each game stream (20Mbps in our system). Remarkably, there's a notable contrast in stall frequency. Illustrated in Fig. 22, players from ISP-T exhibit the lowest stall frequency, trailed by ISP-U and ISP-M, those from ISP-B encounter the highest stall frequency. Divergent stall performances stem from disparities in fundamental network performance and link load among ISPs: *(i)* ISP-B, in its nascent stage, is gradually enhancing its CDN links and base station infrastructure, showing relatively poor network RTT, average 28.0ms. *(ii)* The RTT conditions among ISP-M and ISP-U are comparable, 24.3ms and 23.8ms respectively, both are worse than ISP-T's 21.8ms. Fortunately, irrespective of the ISP networks, Tooth can achieve exceptional stall performance, reducing the stall frequency of the next best RTC-FEC+ by 1.63× to 1.72×.

Tooth can achieve consistent QoE gains in both intra-city and cross-city sessions. We divide gaming sessions into two

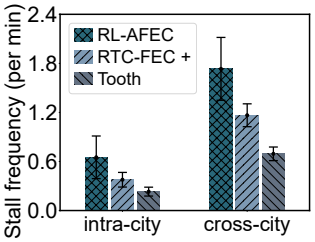


Figure 23: Stall frequency under different server distances.

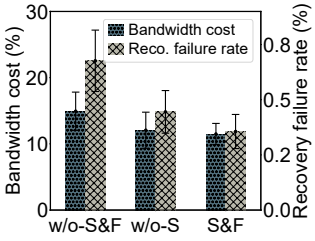


Figure 24: Ablation study of Tooth's two modules.

categories: intra-city sessions and cross-city sessions, contingent upon whether the player and game server are in the same city, accounting for 17.2% and 82.8% of sessions respectively. As illustrated in Fig. 23, Tooth can decrease the stall frequency of the baselines by 40.5% and 59.9% in cross-city sessions. More importantly, in cross-city sessions, Tooth's stall performance can even match that of RL-AFEC in intra-city sessions, and it demonstrates greater robustness (with smaller error bar). Additionally, for intra-city sessions boasting superior network conditions, Tooth shows noteworthy stall frequency gains, reducing by 38.6% and 64.5% compared to the baselines.

5.3 Micro-benchmark Comparisons

Ablation study of Tooth's two modules. Here we examine the separate utility of Tooth's two modules. For *slow-module*, we build its heuristic replica with the arithmetic average method, estimating network loss rate lr and loss aggregation la as the averages over the last three RTCP feedback periods. For *fast-module*'s heuristic replica, we look for a past RTCP feedback period where the network loss pattern is closest to lr and la, then use fl as the observation window to observe the maximum number N of lost packets in this period. The replica will add N redundant packets to the frame. Next, we develop three variants of Tooth: w/o-S&F, w/o-S, S&F, leveraging identical cloud gaming session traces in the simulator for a fair comparison. The results in Fig. 24 show that: *(i)* w/o-S&F can yield 34.3% lower recovery failure rate than w/o-S, and meanwhile reduce the bandwidth cost by 19.2%. The S&F, employing a Random Forest model, demonstrates significantly superior accuracy in determining per-frame redundancy compared to simple historical observations. *(ii)* Compared to w/o-S, *slow-module* provides a more precise estimation of network loss patterns through the NN model. This enhancement allows *slow-module* to further diminish the recovery failure rate from 0.41% in w/o-S to 0.32% in S&F, marking a noteworthy 22.0% improvement. Additionally, we also evaluate the inference accuracy of *slow-module*, more details in Appendix F.

Importance of LRIF-related factors. In §3.2, we expand the LRIF-related factors beyond just frame length (fl) to include network loss rate (lr) and loss aggregation (la). To justify this expansion, we evaluate various combinations of these factors as inputs to Tooth's *fast-module*, specifically A (fl, la), B (fl, lr), and C (fl, lr, la). We train the corresponding *fast-module* variants with these inputs, naming them *fast*-A,

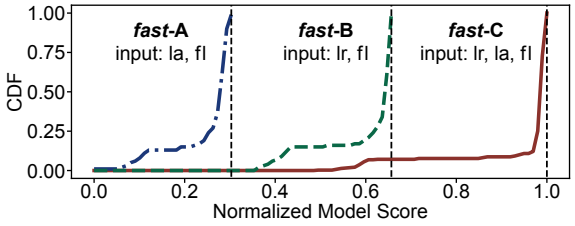


Figure 25: Importance of LRIF-related factors, *i.e.*, network loss rate (lr), loss aggregation (la), and frame length (fl).

Structure	GPU usage (%)	CPU usage (%)	Inference time (ms)	FEC encode time (ms)
Single-stage	7.9	7.1	3.6	0.5
Dual-module	1.4	1.8	0.7	0.5

Table 2: Computational and inference time overhead.

fast-B, and *fast*-C. To eliminate experimental contingency, each *fast-module* variant is trained with 400 different sets of model parameters, obtaining 1200 (3×400) Random Forest models. These models are then tested on the same dataset, with higher scores indicating more accurate LRIF judgment. Fig. 25 illustrates the CDF of model scores for each variant. It is evident that the maximum normalized scores of *fast*-A, *fast*-B, and *fast*-C are 0.30, 0.65, and 1.0 respectively. This highlights the importance of introducing lr and la as LRIF-related factors, especially la, which further emphasizes the subtle differences in network loss pattern that have not been considered by existing works [18, 20, 31, 49, 50].

Overhead comparison of Tooth's model structures. As discussed in §3.2, although it is feasible to directly use powerful neural networks to build Tooth as a single-stage structure, the high execution frequency (every 16.7ms) will inevitably lead to unacceptable computational and inference time overhead. Instead, the dual-module structure we designed decouples Tooth into two modules to reduce overhead. We test both structures on a game server (CPU: Xeon 8369HC; GPU: RTX 3060) and then present their detailed differences in Table 2. Obviously, by decoupling modules and reducing execution frequency, Tooth's dual-module structure can reduce GPU usage, CPU usage, and inference time by 5.6×, 3.9×, and 5.1× respectively. Both structures use RSFEC codec (mentioned in §4) and take 0.5ms for encoding the redundancy, which is a little lower than the decision-making time of Tooth's dual module, *i.e.*, 0.7ms. However, the single-stage decision-making time is 7.2× longer than the FEC encoding time, which obviously reduces its timeliness in commercial cloud gaming.

6 Related work

Network transmission optimization in cloud gaming. Traditional video streaming applications, such as video conferencing and live streaming [32, 34, 39, 46, 53], typically tolerate end-to-end network delays of several hundred milliseconds. In contrast, cloud gaming requires significantly lower latency, necessitating that the time from the player's action to the screen response consistently remains below 100 milliseconds [28, 43, 47]. Cloud gaming currently depends on congestion control algorithms [14, 15, 25, 26, 30, 52–54] to dynamically adjust the data transmission rate, aiming to minimize queuing delays of network packets. Our system employs a custom-optimized variant of GCC [17] for enhanced performance (see Appendix G). Despite these advanced measures, packet loss remains an inevitable challenge, consistently resulting in incomplete data delivery. To address this, cloud gaming systems have to implement effective loss recovery mechanisms, such as retransmission, represented by ART [33], Hairpin [36], and PTO [21]. However, retransmission mechanism commonly introduces additional delays—at least one RTT, and can lead to catastrophic interaction stalls (§5.2).

FEC-based loss recovery. FEC is initially employed to recover lost packets in audio transmissions [38, 40], and has recently been expanded to various video-related applications. For instance, CLOSET [50] and PATON [49] utilize FEC to reduce stalls during TCP-based video playback. DeepRS [20] and RL-AFEC [18] integrate FEC in real-time video communications to minimize end-to-end delays while optimizing FEC bandwidth costs. Additionally, R-FEC [31] and ABRF [19] combine FEC with congestion control to enhance the overall real-time video QoE. A recent advancement in video-conferencing, Tambur [42], introduces stream coding [27] to use multiple sequential frames to recover lost packets from earlier frames, but brings delay that spans multiple frame intervals. However, the above solutions all utilize coarse-grained redundancy mechanisms that do not satisfy the stringent interaction QoE requirements in cloud gaming scenarios (§5.2).

7 Conclusion

In this paper, we propose Tooth, an innovative fine-grained FEC method specifically designed for commercial cloud gaming. We highlight two main novelties: the first-of-its-kind large-scale measurements conducted in real-world environments to investigate the bottlenecks of existing loss recovery methods, and the design of fine-grained FEC encoding mechanisms to achieve an optimal balance between video QoE and redundancy bandwidth cost. Our online evaluations significantly demonstrate the advantages of Tooth. We believe that Tooth has the potential to benefit a wide range of latency-sensitive applications in the future.

Acknowledgements

We sincerely thank the anonymous reviewers and our shepherd, Quentin De Coninck for their insightful comments. Huanhuan Zhang, Anfu Zhou and Liang Liu are the corresponding authors. This work was supported in part by the NSFC Project (62225204, 62302058), the A3 Foresight Program of NSFC (62061146002), the Innovation Research Group Project of NSFC (61921003), the Beijing Major Science and Technology Project (Z231100007423014), and the Youth Top Talent Support Program.

References

[1] Nvidia video encoder programming guide v4.0. https://developer.download.nvidia.cn/compute/nvenc/v4.0/NVENC_VideoEncoder_API_ProgGuide.pdf, 2014.

[2] Alibaba cloud cdn pricing. http://docs-aliyun.cn-hangzhou.oss.aliyun-inc.com/pdf/cdn-billing-intl-en-2016-07-15.pdf, 2016.

[3] How much does it cost to maintain twitch infrastructure? https://trembit.com/blog/how-much-does-it-cost-to-maintain-twitch-infrastructure/, 2019.

[4] Mqtt protocol. [Online]. Available: https://mqtt.org, 2022.

[5] Cloud gaming market report. [Online]. Available: https://www.imarcgroup.com/cloud-gaming-market, 2023.

[6] Ffmpeg. https://www.ffmpeg.org/, 2023.

[7] H.265: High efficiency video coding. https://www.itu.int/rec/T-REC-H.265-202309-S/en, 2023.

[8] Nvidia video encoder programming guide v12.1. https://docs.nvidia.com/video-technologies/video-codec-sdk/12.1/nvenc-video-encoder-api-prog-guide/index.html#recommended-nvenc-settings, 2023.

[9] Webrtc homepage. [Online]. Available: https://webrtc.org/, 2023.

[10] Webrtc in chromium. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/e8f60fd53d98c1b1f67397209066e1453f6957ab/, 2023.

[11] Binomial distribution. [Online]. Available: https://en.wikipedia.org/wiki/Binomial_distribution, 2024.

[12] Genshin impact·cloud. [Online]. Available: https://cloudgenshin.hoyoverse.com/en-us, 2024.

[13] Tower of fantasy. [Online]. Available: https://cloudbase.gg/g/tower-of-fantasy/, 2024.

[14] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic meets modern: a pragmatic learning-based congestion control for the internet. In *SIGCOMM*, pages 632–647. ACM, 2020.

[15] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *NSDI*, pages 329–342. USENIX Association, 2018.

[16] Simone Basso, Michela Meo, Antonio Servetti, and Juan Carlos De Martin. Estimating packet loss rate in the access through application-level measurements. In *W-MUST@SIGCOMM*, pages 7–12. ACM, 2012.

[17] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *MMSys*, pages 13:1–13:12. ACM, 2016.

[18] Ke Chen, Han Wang, Shuwen Fang, Xiaotian Li, Minghao Ye, and H. Jonathan Chao. RL-AFEC: adaptive forward error correction for real-time video communication based on reinforcement learning. In *MMSys*, pages 96–108. ACM, 2022.

[19] Sheng Cheng, Han Hu, and Xinggong Zhang. ABRF: adaptive bitrate-fec joint control for real-time video streaming. *IEEE Trans. Circuits Syst. Video Technol.*, 33(9):5212–5226, 2023.

[20] Sheng Cheng, Han Hu, Xinggong Zhang, and Zongming Guo. Deeprs: Deep-learning based network-adaptive FEC for real-time video communications. In *ISCAS*, pages 1–5. IEEE, 2020.

[21] Sheng Cheng, Han Hu, Xinggong Zhang, and Zongming Guo. Rebuffering but not suffering: Exploring continuous-time quantitative qoe by user's exiting behaviors. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[22] Mark Claypool and Kajal T. Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006.

[23] Mark Claypool and Kajal T. Claypool. Latency can kill: precision and deadline in online games. In *MMSys*, pages 215–222. ACM, 2010.

[24] Matthias Dick, Oliver Wellnitz, and Lars C. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *NETGAMES*, pages 1–7. ACM, 2005.

[25] Mo Dong, Qingxi Li, Doron Zarchy, Philip Brighten Godfrey, and Michael Schapira. PCC: re-architecting congestion control for consistent high performance. In *NSDI*, pages 395–408. USENIX Association, 2015.

[26] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC vivace: Online-learning congestion control. In *NSDI*, pages 343–356. USENIX Association, 2018.

[27] Silas L. Fong, Ashish Khisti, Baochun Li, Wai-Tian Tan, Xiaoqing Zhu, and John G. Apostolopoulos. Optimal streaming codes for channels with burst and arbitrary

erasures. *IEEE Trans. Inf. Theory*, 65(7):4274–4292, 2019.

[28] Stefan Holmer, Mikhal Shemer, and Marco Paniconi. Handling packet loss in webrtc. In *ICIP*, pages 1860–1864. IEEE, 2013.

[29] Teemu Kämäräinen, Matti Siekkinen, Antti Ylä-Jääski, Wenxiao Zhang, and Pan Hui. A measurement study on achieving imperceptible latency in mobile cloud gaming. In *MMSys*, pages 88–99. ACM, 2017.

[30] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *SIGCOMM*, pages 107–125. ACM, 2020.

[31] Insoo Lee, Seyeon Kim, Sandesh Dhawaskar Sathyanarayana, Kyungmin Bin, Song Chong, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. R-FEC: rl-based FEC adjustment for better qoe in webrtc. In *ACM Multimedia*, pages 2948–2956. ACM, 2022.

[32] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, Chen Sun, Gareth Tyson, and Hongqiang Harry Liu. Livenet: a low-latency video transport network for large-scale live streaming. In *SIGCOMM*, pages 812–825. ACM, 2022.

[33] Tong Li, Wei Liu, Xinyu Ma, Shuaipeng Zhu, Jingkun Cao, Senzhen Liu, Taotao Zhang, Yinfeng Zhu, Bo Wu, and Ke Xu. ART: adaptive retransmission for wide-area loss recovery in the wild. In *ICNP*, pages 1–11. IEEE, 2023.

[34] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. Gso-simulcast: global stream orchestration in simulcast video conferencing systems. In *SIGCOMM*, pages 826–839. ACM, 2022.

[35] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *SIGCOMM*, pages 193–206. ACM, 2022.

[36] Zili Meng, Xiao Kong, Jing Chen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Hairpin: Rethinking packet loss recovery in edge-based interactive video streaming. In *USENIX NSDI*, 2024.

[37] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling high quality real-time communications with adaptive frame-rate. In *NSDI*, pages 1429–1450. USENIX Association, 2023.

[38] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion control using FEC for conversational multimedia communication. In *MMSys*, pages 191–202. ACM, 2014.

[39] Zipeng Pan, Yuan Zhang, Tao Lin, and Jinyao Yan. Liveae: Attention-based and edge-assisted viewport prediction for live 360° video streaming. In *EMS@SIGCOMM*, pages 28–33. ACM, 2023.

[40] Colin S. Perkins, Orion Hodson, and Vicky Hardman. A survey of packet loss recovery techniques for streaming audio. *IEEE Netw.*, 12(5):40–48, 1998.

[41] James S Plank. A tutorial on reed–solomon coding for fault-tolerance in raid-like systems. *Software: Practice and Experience*, 27(9):995–1012, 1997.

[42] Michael Rudow, Francis Y. Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and K. V. Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *NSDI*, pages 953–971. USENIX Association, 2023.

[43] Sandesh Dhawaskar Sathyanarayana, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Converge: Qoe-driven multipath video conferencing over webrtc. In *SIGCOMM*, pages 637–653. ACM, 2023.

[44] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A transport protocol for real-time applications. *RFC*, 3550:1–104, 2003.

[45] Susanna Schwarzmann, Nick Hainke, Thomas Zinner, Christian Sieber, Werner Robitza, and Alexander Raake. Comparing fixed and variable segment durations for adaptive video streaming: a holistic analysis. In *MMSys*, pages 38–53. ACM, 2020.

[46] Ziyi Wang, Yong Cui, Xiaoyu Hu, Xin Wang, Wei Tsang Ooi, and Yi Li. Multilive: Adaptive bitrate control for low-delay multi-party interactive live streaming. In *INFOCOM*, pages 1093–1102. IEEE, 2020.

[47] Bingyang Wu, Kun Qian, Bo Li, Yunfei Ma, Qi Zhang, Zhigang Jiang, Jiayu Zhao, Dennis Cai, Ennan Zhai, Xuanzhe Liu, and Xin Jin. XRON: A hybrid elastic cloud overlay network for video conferencing at planetary scale. In *SIGCOMM*, pages 696–709. ACM, 2023.

[48] Jiangkai Wu, Yu Guan, Qi Mao, Yong Cui, Zongming Guo, and Xinggong Zhang. Zgaming: Zero-latency 3d cloud gaming by image prediction. In *SIGCOMM*, pages 710–723. ACM, 2023.

[49] Jiyan Wu, Bo Cheng, Ming Wang, and Junliang Chen. Priority-aware FEC coding for high-definition mobile video delivery using TCP. *IEEE Trans. Mob. Comput.*, 16(4):1090–1106, 2017.

[50] Jiyan Wu, Chau Yuen, and Junliang Chen. Leveraging the delay-friendliness of TCP with FEC coding in real-time video communication. *IEEE Trans. Commun.*, 63(10):3584–3599, 2015.

[51] Xiaokun Xu and Mark Claypool. Measurement of cloud-based game streaming system response to competing TCP cubic or TCP BBR flows. In *IMC*, pages 305–316. ACM, 2022.

[52] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In *MobiCom*, pages 1–14, 2020.

[53] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. Onrl: improving mobile video telephony via online reinforcement learning. In *MobiCom*, pages 29:1–29:14. ACM, 2020.

[54] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. Learning to coordinate video codec with transport protocol for mobile video telephony. In *MobiCom*, pages 1–16, 2019.

Appendices

A Measurement Method for Application-layer Interaction Latency

We track user interaction delays frame by frame to identify whether a video frame is delayed and by how much. The interaction latency refers to the time it takes from when a user inputs an operation (*e.g.*, run and hit) into the game client to when the user receives the screen feedback. To measure this latency actually, we break down the cloud gaming application-layer latency into two parts:

(i) System processing delay, which involves tasks like capturing user input, video encoding, decoding, and playback on the game client and server. These contribute to a total average delay of 32ms, which is relatively stable.

(ii) Data round-trip network delay, which involves transmitting user operations to game server (uplink) and receiving all RTP packets of the responded video frame from the game server to the client (downlink) via the Internet. These delays depend on network transmission conditions, we record frame ID and the timestamps of all uplink and downlink packets to calculate them.

If the sum of these two parts of delays exceeds 100ms, we determine that the user interaction has timed out, in the same way, we can know how late the complete frame arrives.

B Applying Infinite GOP Video Encoding in Cloud Gaming

Traditional real-time video communication usually uses the finite GOP encoding manner and sets a fixed GOP length for the video codec, such as 30 set in RL-AFEC [18]. For cloud gaming, this ultra-low latency interactive video streaming application, a common practice in the industry is to adopt the infinite GOP video encoding [1, 8]. Infinite GOP video encoding disables the server to send I frames periodically, and the client is allowed to request a new I frame only for necessary error recovery, *e.g.*, the client frame buffer pool overflowed after multiple video frames suffering packet loss and failing to be decoded. As a result, our cloud gaming system sends an I frame every 4.7 minutes on average, which means our average GOP length is 16,920 and 564× higher than that of conventional video streaming in [18].

We explain that the P frame only needs to encode the difference between the current video frame and the previous one, while the I frame must encode the entire video frame. When encoding the next video frame, the size of a P frame will not exceed that of an I frame, even if the cloud gaming scene changes. Therefore, the infinite GOP encoding can avoid frequently sending high-load I frames and the accompanying latency surge, and is recommended by the popular NVENC video codec [1, 8] for low-latency cases, especially game-streaming, video conferencing, *etc*. We emphasize that multiple top gaming platforms, including our partner company W, adopt this encoding method to optimize the cloud gaming interaction latency. This is a common practice in industry although less frequently introduced in academic papers.

C Protection of uplink client-to-server data

The client-to-server data flow in cloud gaming, mainly discussing player operation data here, typically ranges from a few to several tens of Kbps, which is much smaller than the available network bandwidth. For such a small operation data flow, there are no multiple concurrent packets like server-to-client video flow. Therefore, the industry usually doesn't add redundancy using FEC rather than adopting some repeated delivery strategies (*i.e.*, sending data replicas) to ensure reliable transmission. Take our system as an instance, a single player operation amounts to no more than 10 bytes, and the total number of concurrent operations remains far below the maximum capacity of an RTP packet (MTU=1500). Thus, an operation RTP packet from the client will not only contain the latest player operations but also the previous n operations. Additionally, each operation RTP packet will be sent twice, with a 2ms interval, to further improve reliability.

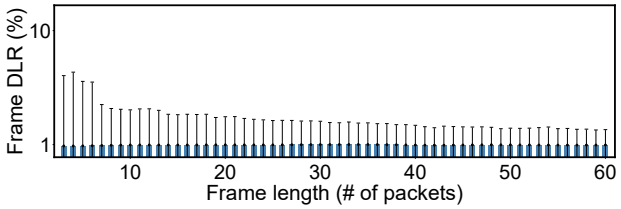


Figure 26: LRIF results of all video frames (including those experiencing no losses) observed in our commercial cloud gaming platform.

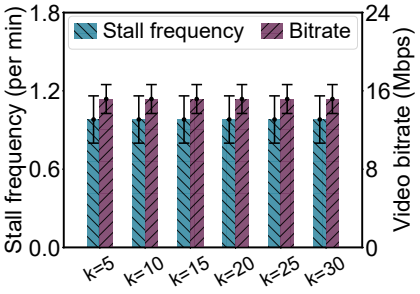


Figure 27: Ablation experiment results of the RL-AFEC model with different K values.

D A Deeper Dive into LRIF Results

We make the following explanations to clarify why in Fig. 7 we need analyze the LRIFs of frames that suffered packet loss rather than all GOP frames. *(i)* FEC in video streaming is more interested in higher percentiles of LRIF because only they can reflect how many redundant packets that FEC should add to protect all video frames as much as possible from network packet loss events; *(ii)* As shown in Fig.26, the average and standard deviation of all GOP frame LRIFs are actually consistent with the overall network loss rate (about 1%), but obviously, FEC should not adjust the redundancy rate based on this insight. So, in Fig. 7, we plot the LRIF average and std of video frames that suffered packet loss to reveal the relationship between non-zero LRIF and video frame length, which is more meaningful for guiding us in designing the fine-grained FEC in cloud gaming.

E Ablation Study of RL-AFEC Using Different K Values

For RL-AFEC [18], we further present ablation study experiments with the variation of K, *i.e.* setting K as 5, 10, 15, 20, 25, 30, consistent with [15]. As shown in Fig. 27, the six candidate values of K have little impact on the cloud gaming interaction QoE, which can be almost ignored. That's because the GOP length of cloud gaming streaming is very large, it is far from enough to optimize only a few K critical frames. We should set a fine-grained redundancy rate for each video frame.

More importantly, we emphasize that directly defining K to be GOP size N with RL-AFEC will not yield a similar outcome as Tooth. We elaborate on the key differences in the design principles of Tooth and RL-AFEC: *(i)* RL-AFEC is designed for common video call scenarios, where the length of the remaining P frames remains relatively uniform, apart from the initial I frame in each GOP. In other words, RL-AFEC does not account for significant frame length variations and the resulting huge LRIF differences in cloud gaming streaming. As a result, RL-AFEC may fall short of fully meeting the redundancy adaptation design required for specific cloud gaming scenarios. *(ii)* In RL-AFEC, the GOP length N is 30 and the authors conclude that the optimal K value is 15, the reason is that increasing K will cause its reinforcement learning model to explore huge action space, resulting in an obvious decrease in its model effectiveness. Especially considering that the GOP size N will be hundreds even thousands of times larger than K in cloud gaming scenarios, defining K to be N is even less feasible. Instead, Tooth, with its unique dual-module design, makes larger K or even infinite K feasible, and ultimately is fully capable of setting fine-grained redundancy rate for each frame.

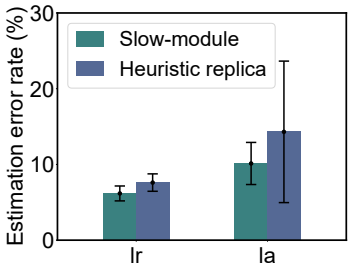


Figure 28: Estimation errors of network loss rate *lr* and loss aggregation *la*.

F Micro-benchmark Experiments on Tooth's Slow-module

To further evaluate Tooth's *slow-module*, we analyze Tooth online dataset to supplement the micro-benchmark experiment results, including the estimation errors (mean and std) of network loss rate *le* and loss aggregation *la*. Then, we calculate the mean and std of all estimation error rates. Moreover, we also replay these online traces in the simulator and evaluate *slow-module*'s heuristic replica (implementation details in §5.3) to compare their performance differences. As shown in Fig. 28, Tooth's *slow-module* behaves good inference accuracy, with an average error of 6.2% for *lr* and 10.1% for *la*, which are 18.8% and 29.2% lower than the heuristic replica, respectively.

G Fine-tuning of GCC's hyper-parameters in our system

The default GCC congestion control algorithm in WebRTC [10] isn't fully adapted for the streaming characteristics of cloud gaming. We follow the algorithm logic of GCC but

customize its hyper-parameters based on practical engineering experiments: (i) We increase GCC's upper bitrate limit to 20Mbps to meet the production streaming requirement of cloud gaming; (ii) We shorten the observation window length in GCC's trend-line estimator from 20 to 15, making the system more aware of delay increases; (iii) We slightly enlarge the rate increase factor in AIMD-based bitrate control from 1.08 to 1.11 for fast bitrate recovery after the delay stabilizes; (iv) We disable GCC's probe action during bitrate reduction since cloud gaming is more delay-sensitive than bandwidth-sensitive. Probing at this time can easily harm delay performance. (v) When the observed network loss rate is below 0.3%, we block GCC's loss-driven decision on reducing bitrate and rely solely on the delay-driven bitrate adjustments.

AsTree: An Audio Subscription Architecture Enabling Massive-Scale Multi-Party Conferencing

Operational Systems Track

Tong Meng Wenfeng Li Chao Yuan Changqing Yan Le Zhang
ByteDance Inc.

Abstract

While operating a multi-party video conferencing system (Lark) globally, we find that audio subscription alone may pose considerable challenges to the network, especially when scaling towards massive scales. Traditional strategy of subscribing to all remote participants suffers from issues such as signaling storm, excessive bandwidth and resource consumption on both server and client sides. Aimed at enhanced scalability, we share our design of AsTree, an audio subscription architecture. By a cascading tree topology and media plane-based audio selection, AsTree dramatically reduces the number of signaling messages and audio streams to forward. Practical deployment in Lark reduces audio and video stall ratios by more than 30% and 50%. We also receive 40% less negative client reviews, strongly proving the value of AsTree.

1 Introduction

Real-time multi-party video conferencing has been ubiquitous in our daily lives. Lark [1], as one of a rich set of heavily used services and products (*e.g.*, Zoom [2], Microsoft Teams [3], DingTalk [4]), serves tens of millions of users, and benefits vast many industries such as online education, international business, interactive social networking, *etc.*

Rather than focusing on simulcast-based video subscription like many existing literature [5, 6], we report our experiences of designing and deploying AsTree in Lark, an audio subscription architecture targeted at massive-scale conferences (*e.g.*, scaling up to hundreds and thousands of participants per room). Notably, there are several key differences between audio and video subscription, posing challenges to the architecture for the former correspondingly.

Whether to subscribe to a stream. Subscription and unsubscription to video streams can be the outcome of a subscriber's proactive operations, *e.g.*, when the subscriber scrolls through the participant list, or pins/unpins a specific publisher. In comparison, a subscriber mostly needs to be passively subscribed to an audio stream, as long as its publisher is actively speaking. That makes the audio subscription relations between conference participants unpredictable, because we cannot deterministically predict when and which participant will speak.

When to subscribe to a stream. Subscribing to a video stream requires 4 steps: (1) a publisher broadcasts a Publish signaling message when turning on camera, (2) a subscriber sends an Subscribe message (not necessarily closely follow previous step), (3) an overlay cascading path is established between the publisher and subscriber, (4) the video stream is cascaded and distributed. For an audio stream, however, the above steps cannot wait until a participant becomes unmuted. Considering that audio is generally more latency-sensitive than video, the signaling round trip and on-demand establishment of cascading path induce unwanted delay. What is worse, there would be loss of audible information if a participant starts talking right after clicking the unmute button.

How many streams to subscribe. Even if all participants open their cameras, each one subscribes to a bounded number of video streams, depending on the user interface layout. Yet when it comes to audio subscription, the number of audio streams for subscription may increase indefinitely, since every participant in a room may unmute. At a massive conference scale, subscribing to all unmuted participants may lead to considerable signaling overheads and bandwidth consumption.

In the early stage of our deployment, massive conferences where all participants are allowed to unmute themselves were uncommon. So we assumed audio consumes much lower bandwidth than video at that time, and employed a straightforward "subscribe-to-all" strategy for audio subscription (denoted as FullAud henceforth). To improve timeliness of audio signal, subscription to a participant is aggressively initiated when they joins a conference room, without waiting for them to unmute. Since most participants start muted by default, we expect the signaling round trip and cascading path establishment to finish before a participant actually starts talking. Along with that, each participant keeps publishing an audio stream while in the conference. When the microphone is muted, the audio stream simply contains a few Discontinuous Transmission (DTX) frames [7]. Then, the signaling message emitted from a participant clicking the mute/unmute button is used to toggle the displayed audio status seen by other participants. As far as we know, at least until the pandemic, many products also default to full audio subscription [8–10], though their realizations may differ.

Not surprisingly, as our DAU (daily active users) continuously grows and massive-scale conferences become increasingly frequent, FullAud starts to face scalability issues.

- **Signaling storm.** Influx of participants often happen at the start of a conference, inducing surging signaling messages for audio subscription. As a result, both audio and video are more likely to suffer from network and host congestion. Besides, the signaling broadcast generated every time a

participant mutes/unmutes the microphone interferes with cascading and distribution of media data to some extent.

- **Client-side resource overheads.** When a single participant has to fetch tens or hundreds of audio streams, the last-mile access link can be easily overwhelmed, increasing chances of choppy audio and video rebuffering. Meanwhile, CPU, memory, and battery power on user devices consistently operate under excessive loads, and thus, tend to cause performance issues such as overheating that eventually impair audio/video quality, as well.
- **Server-side resource overheads.** A room with N participants distributes $O(N^2)$ downlink audio streams at the edge. The quadratic scaling egress bandwidth makes FullAud substantially less affordable. In addition, the ever-expanding conference scale urges prohibitive investment in WAN bandwidth and media servers, considering that (1) individual media servers are more likely to be overloaded in densely populated regions, and (2) massive participants are more geographically dispersed and cascade more server clusters.

The AsTree architecture is designed to overcome the above issues. It is inspired by an empirical observation: at any time during a multi-party conference, almost all critical information comes from a small number of audio streams [10, 11]. Therefore, a majority of audio streams can be ignored from subscription. To effectively carry out that idea in a practical system, we should fulfill the following objectives.

- **Light-weight signaling.** For agile conference management, the selection of audio streams and notification to participants should not only avoid adding new signaling messages, but also significantly alleviate the existing signaling storm.
- **No single point of failure.** Audio stream selection needs to compare attributes of all audio streams. Gathering streams to a single server for that purpose should be avoided. Otherwise, that server becomes the computing and bandwidth bottleneck, whose failure impacts all participants in a room.
- **Compatibility with simulcast.** We want the new design for audio subscription to induce minimal changes to the already mature video simulcast architecture, and enable independent engineering development and modular upgrade.

As a large-scale system in a well-explored field, AsTree shares similarities with some existing works (*e.g.*, separation of media and control planes [5], cascaded media servers [12, 13]). Importantly, many conferencing service providers disclose limited information on how they scale the audio subscription architecture. To the best of our knowledge, we are the first to provide comprehensive experiences on that. Our main contributions are highlighted below.

- AsTree constructs a cascading tree between media servers for each conference room to lower WAN bandwidth consumption. It assigns a delegate server in each region connecting other intra-region media servers, and calculates an inter-region tree cascading region delegates.
- AsTree selects a subset of audio streams composed of dominant speakers at each media server, without control plane involvement. Effectively, a subscriber only subscribes to those audio streams containing valid vocal signal. Moreover, it eliminates the signaling broadcast when a participant mutes/unmutes the microphone.
- We present extensive results from benchmarking tests and practical deployment. AsTree is shown to significantly outperform FullAud. To be specific, negative reviews are reduced by 50%, audio stall is reduced by 30%, and video stall is reduced by 50%. Also, a media server can hold conferences with an order of magnitude more participants.

This work does not raise any ethical issues.

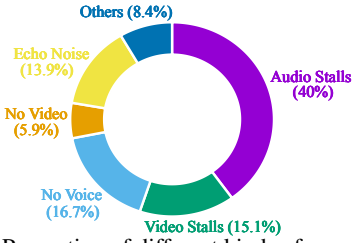


Figure 1: Proportion of different kinds of user complaints

2 Motivation

Thanks to the deployment of simulcast, Lark's video quality of experience (QoE) has been notably improved, making audio stall the most common category among millions of our user complaints (as in Figure 1, collected by post-call questionnaire with non-exclusive options). That confirms the importance of evolving the audio subscription architecture. To fully motivate AsTree and gain some insights, we explain the scalability issues of our early-stage FullAud implementation in more detail, and analyze the drawbacks of several strawman solutions.

2.1 Scalability Issues of FullAud

2.1.1 Local Signaling Overheads

A centralized signaling unit is both the gathering point of signaling reports and originating source of signaling broadcasts, *e.g.*, so as to maintain a full picture of the conference room and direct the establishment of cascading links, respectively. To alleviate its pressure, we deploy local signaling units co-located with media servers, and aggregate signaling messages between centralized and local signaling units into $O(1)$ per broadcast. However, each local signaling unit generates $O(N)$ messages while signaling its connected participants.

As an example, Figure 2 illustrates the signaling round trip for audio subscription. FullAud amortizes the signaling delay to the moment a participant joins a conference, and thus, merges the broadcast of Join and Publish. It also triggers a consequent ingestion of $O(N)$ Subscribe messages. Additionally, signaling broadcasts also happen when participants (1) click the mute/unmute button and (2) leave the room.

If combining all participants in a room, the volume of signaling messages borne by local signaling units is as high as

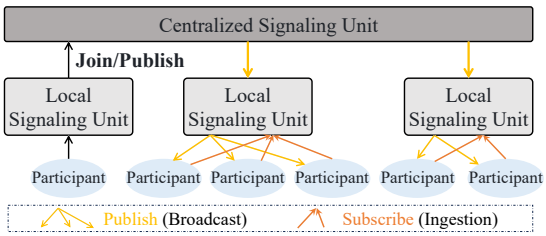


Figure 2: Signaling round trip for audio subscription in FullAud

Video Stream	Focus Window	Thumbnail
	700 Kbps	120 Kbps
Audio Stream	Unmuted	Muted (DTX only)
	64 Kbps	1 Kbps

Table 1: Approximate bandwidth requirements of video and audio streams (video based on Lark's mobile app)

$O(N^2)$. Although each message does not exceed several tens of bytes, massive-scale conferences may still suffer from signaling storm and degraded user experience, especially when:

- A large number of participants join within a short time period, which is common at the starting phase of a conference.
- Highly concurrent conferences during peak hours already leave a high CPU usage on the edge nodes.
- Participants repeatedly unmute/mute for different purposes, including playing with the button aimlessly.

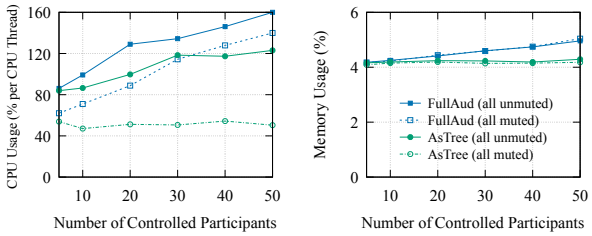
As will be demonstrated in §6.1, the signaling storm may consume even higher CPU than media processing. Regardless of the frequency of signaling storm, the possibility of its occurrence prevents us from scheduling many participants to the same media server. That limits the maximum number of participants a single media server can accommodate per room under FullAud, leading to inefficient utilization of server machine resources.

2.1.2 Client-Side Overheads

FullAud's client-side bandwidth consumption increases linearly with the room size. To intuitively compare it with video subscription, let us consider a simple scenario: a participant joins a conference using Lark Mobile on a smartphone. Assume speaker layout is used, with a 720p full-screen focus video and a 90p floating thumbnail. Referring to [14] and our internal configurations, bandwidth requirements of individual video and audio streams are approximated as in Table 1.

Each participant needs 0.82 Mbps to fetch the focus and thumbnail videos, regardless of the room size. When there are more than 12 unmuted participants at the same time, audio will consume more bandwidth than video. Even if all participants are muted, their DTX-based audio streams will require the same order of magnitude bandwidth as video with more than 100 participants. We should note that this is a conservative estimation, since we do not consider possible reduction in video resolution to resist weak networks, and needs for higher-quality audio for increased immersion.

In addition, we leverage the stress test system at Lark to demonstrate FullAud's hardware loads on user devices. We use a Redmi Note 13 smartphone to join setup conferences with different number of controlled participants, and monitor its CPU/memory usage (Figure 3). All controlled participants do not publish video streams to isolate the overheads of audio subscription. We test two extreme cases where controlled participants are either all unmuted or all muted.



(a) CPU Overheads **(b)** Memory Overheads

Figure 3: Client-side overheads induced by FullAud

As expected, FullAud's resource usage scales nearly linearly with the number of participants. Because the smartphone has to receive and process muted audio streams, when the number of muted controlled participants increases from 5 to 50, it occupies more than half an additional CPU core. It has almost the same memory usage in both cases. That means some low-end smartphones may not even have enough capabilities to join a conference with hundreds of participants. For comparison, we also depict AsTree in Figure 3. In AsTree, the smartphone subscribes to a limited number of audio streams, excluding any muted streams. Thus, its resource usage remains relatively stable with different number of muted participants. When there are 50 muted participants, AsTree's CPU and memory usage are 64% and 17% lower than FullAud, respectively. We note that all controlled participants use the same audio track, causing the selected dominant speakers to change frequently. That explains the increasing CPU usage of AsTree when there are more unmuted participants.

2.1.3 Server-Side Overheads

FullAud effectively creates an overlay mesh cascading every pair of media servers. As a result, a room that involves N participants and M media servers consumes $O(MN)$ and $O(N^2)$ bandwidth on inter-server WAN and egress links, respectively.

More importantly, a larger conference scale alone can raise server-side expenses, even with the same user population. Assume a media server instance has the CPU, memory and associated egress link bandwidth that are just enough to distribute 100 audio streams. Then, one such instance can hold 50 concurrent 2-party audio-only conferences, but we need 100 instances to have a single 100-party conference (*e.g.*, each instance serves a single participant). Moreover, the overall egress bandwidth consumption is 99×, and the same amount of WAN bandwidth is needed for cascading. Although ignoring the influence of many practical factors, such a comparison illustrates the possibly forbidding costs while scaling FullAud. Similar issue is mitigated by simulcast in video subscription, where the number of video streams distributed to an individual participant is capped by the user interface.

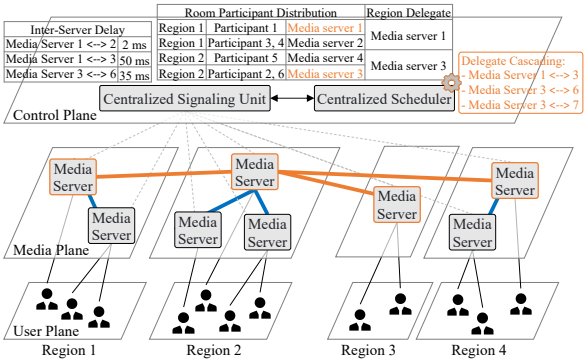


Figure 4: Overview of the AsTree architecture (local signaling units co-located with media servers are omitted)

2.2 Strawman Solution Analysis

Several strawman solutions can reduce the number of audio streams subscribed by each participant. Though seemingly simple, they come with various drawbacks.

(1) Restricted unmute privilege. The most straightforward way to enforce subscription to fewer audio streams is to set a hard limit on the maximum number of concurrent unmuted participants. Many online webinars adopt that solution. Therein, an audience member should request to unmute from a host, and be muted after finishing. That way, subscribing to all active speakers does not generate high overheads.

Drawbacks: Having to apply and compete for opportunities to speak induces inconvenient manual operations. One such interaction easily takes several seconds. Thus, the solution has a much narrower scope other than lecture-style conferences. It is not applicable in emerging scenarios requiring strong interactivity, such as online social and virtual karaoke.

(2) Signaling-based audio selection. While publishing to the local media server,[1] a participant also signals the attributes of their audio stream to the centralized signaling unit. Based on the collected audio attributes, the signaling unit leverages a controller to select a subset of audio streams to be forwarded. Each media server waits for instructions from the signaling unit to determine which ingested audio streams to cascade.

Drawbacks: This solution introduces new signaling messages for audio stream selection, which should be sent frequently to reflect dynamics in audio attributes (*e.g.*, fluctuating voice volume, *etc.*). That exacerbates the signaling storm issue. What is worse, user experience is more vulnerable due to the additional signaling round trip for a selected audio stream.

(3) Select before distribute A media server still pulls one audio stream from each participant, whether they are unmuted or not. Whereas in the downlink direction, a selection logic is added to each media server, such that only selective audio streams are distributed. Besides that, the publishing and cascading of audio streams are kept the same as in FullAud.

Drawbacks: Compared with the previous two solutions, this solution puts no limit to applicable use cases, and avoids complicated signaling interactions for audio selection. Yet it is still based on full audio subscription, and media servers are still cascaded by a mesh. An audio stream, whether selected or not, needs to be received and processed by all media servers involved in a conference. That provides limited help to control server-side costs on WAN bandwidth and media servers. In fact, that is an important reason why Jitsi had to temporarily disable mesh cascading during the COVID-19 pandemic [15].

[1] A local media server specific to a participant is the server to which the participant is directly connected. A local media server specific to an audio stream is the local media server of its publisher.

3 AsTree Design Overview

Figure 4 presents the high-level architecture of AsTree. Similar to [5], it is composed of user plane, media plane, and control plane. The user plane and media plane are divided into separate geographical regions, and the control plane is logically centralized.

AsTree is based on selective forwarding units (SFUs), which has been the dominating choice by modern multi-party conferencing providers [5, 6, 16, 17]. Participants connect to nearby local SFU media servers. Cascaded SFUs form overlay paths between participants, and forward media streams without decoding them. Furthermore, one or more SFUs involved in a conference room form an audio selection tree (hence the acronym AsTree[2], details in §4) as opposed to a mesh. Each region has a single media server as the region delegate. The other media servers are cascaded to their intra-region delegates. Then, all the delegate servers are connected by a spanning tree encompassing the corresponding regions. Such an AsTree topology is calculated by a centralized scheduler in the control plane. For that purpose, it synthesizes distribution of participants in the same room and latency between media servers, which are collected by the signaling unit.

Beyond that, the control plane does not participate in selection of audio streams. Instead, audio selection is conducted hop by hop at the media plane, *i.e.*, each media server only selects at most L loudest active speakers (details in §5). All participants, including those muted ones, still keep publishing audio streams to their local media servers, but muted audio streams will be excluded from audio selection. Thus, the number of audio streams forwarded on any cascading link and distributed to any participant is capped by a constant parameter. Moreover, no signaling broadcast is needed when a participant mutes/unmutes the microphone. The audio status of each participant displayed on user interface directly corresponds to the results of audio selection, *i.e.*, only those whose audio streams are selected are displayed as unmuted.

4 Calculation of Cascading Tree

In Lark's deployment, the division of regions is fairly static, depending on geo-planning of server clusters. The calculation of the AsTree topology focuses on the two hierarchies as outlined in §3, *i.e.*, intra-region and inter-region cascading.

[2] AsTree may stand for the audio subscription architecture or the cascading tree topology. The context where it is used can help avoid ambiguity.

4.1 Intra-Region Cascading

The core of intra-region cascading is the selection of region delegate. To lower the implementation complexity, we do not adopt complicated leader election algorithms (*e.g.*, [18, 19]) from the field of distributed systems. Instead, we take a "first comer elected" method. When a media server is the first in its region to join a conference room, it is configured to be the region delegate. Subsequent media servers joining the same room in the same region are cascaded to that delegate.

The above simple method is out of three considerations. First, the one-way latency between media servers in the same region is usually much lower than the industrial target for acceptable one-way latency (*e.g.*, above 200 ms between individual participants [20–22]). Changing the elected region delegate delivers similar audio QoE in most cases. Second, the interaction delay between intra-region participants is kept minimal, because their audio streams reach each other without leaving the region. Third, since each media server only forwards at most L audio streams, the region delegate will not face a high processing load. According to our experiences, the number of audio streams cascaded from non-delegate servers to a region delegate rarely exceeds two dozens, because many participants stay muted in a majority of time. That is far below the capacity of a single media server (§6.1).

A region delegate remains in the cascading tree, unless all participants in its region leave the room (details in §4.3).

4.2 Inter-Region Cascading

A spanning tree can connect a set of region delegates with the fewest cascading links. Each cascading link represents an overlay path, and the cascading links between a pair of delegates form a cascading path. The calculation of cascading spanning tree is formulated as follows.

$$
\begin{aligned}
\textit{Input}: \quad & \mathcal{R} = \{ R_1, R_2, \cdots, R_M \} \\
& \mathcal{N} = \{ N_1, N_2, \cdots, N_M \} \\
& \mathcal{D} = \{ d^L_{ij} \mid 1 \le i < j \le M \} \\
\textit{Output}: \quad & \mathcal{T} = \{ x_{ij} \mid 1 \le i < j \le M \} \\
\textit{Subject to}: \quad & x_{ij} \in \{ 0, 1 \}, \quad \forall\ 1 \le i < j \le M \\
& \sum_{j=i+1}^{M} x_{ij} + \sum_{k=1}^{i-1} x_{ki} > 0, \quad \forall\ 1 \le i \le M \\
& \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} x_{ij} = M - 1
\end{aligned}
$$

We use $\mathcal{R}$, $\mathcal{N}$, and $\mathcal{D}$ to denote the set of M ($M \ge 2$) region delegates, the number of participants in each region, and the cascading link RTT between each pair of delegates, respectively. They are updated by the signaling messages triggered when a participant joins or leaves. The cascading link RTTs are measured passively via recent media flows, or proactively with infrequent probes. To lower complexity, d^L_{ij} is regarded as undirected, and takes the larger value between the RTT measured at R_i and at R_j.

The inter-region spanning tree should aim at objectives of practical importance. Therefore, classic minimum spanning tree (MST) algorithms that minimize the total RTT of involved cascading links are not suitable in this case, because it may lead to cascading paths with arbitrarily long RTTs (Figure 5d). We provide several alternatives to generate the inter-region spanning tree. Meanwhile, we illustrate using an example with 4 regions in Figure 5. The RTT statistics in the figure refer to public Azure network latency [23]. Again, we defer the discussion on updating the spanning tree along with changes in $\mathcal{R}$ and $\mathcal{N}$ to §4.3.

4.2.1 Minimized Longest Path RTT

In a spanning tree, the cascading path with the longest RTT largely determines the worst-case end-to-end audio latency. Minimizing the longest path RTT guarantees the lower bound of QoE for all participants in a room. Given the above problem formulation, the objective can be expressed as:

$$
\textit{Objective}: \quad \min \left(\arg\max_{1 \le i < j \le M} w_{ij} \cdot d^P_{ij} \right)
$$

where d^P_{ij} denotes the RTT of cascading path between R_i and R_j. In addition, we add a path weight w_{ij} to enable priority control among different regions. For example, we can assign a smaller value to w_{ki} and w_{ij} ($k < i < j$) than the other weights, if R_i is favored as the root of the spanning tree (*e.g.*, when the conference host is in the same region as R_i and we need to minimize the delay from the host to other participants).

As shown in Figure 5a, when all path weights equal 1, the spanning tree rooted at West US minimizes RTT of the longest cascading path, which is between West Europe and East Asia (299 ms). However, if the host locates in East Asia and the three cascading paths originating from there have a smaller weight of 0.5, the root of the optimal spanning tree will change to delegate server in East Asia (Figure 5b).

4.2.2 Minimized Weighted Average Path RTT

Sometimes it is commercially advantageous to enhance experience for a majority of participants. That leads to the objective of minimizing weighted average of cascading path RTTs, based on participant distribution between regions:

$$
\textit{Objective}: \quad \min \sum_{1 \le i < j \le M} w_{ij} \cdot d^P_{ij} \cdot (N_i + N_j)
$$

Compared with §4.2.1, the optimal cascading spanning tree under unity value path weights is now rooted at East Asia (Figure 5c), because 5× participants in East Asia can benefit from shorter cascading paths. However, if the weights of three cascading paths from West US is decreased to 0.5, the optimal root delegate will still be in West US. We omit that case in Figure 5 due to limited space.

4.2.3 One Single Master Delegate

Both objectives on above increase the calculation complexity compared with classic MST algorithms. For cost-effectiveness, we propose a simple heuristic instead of trying

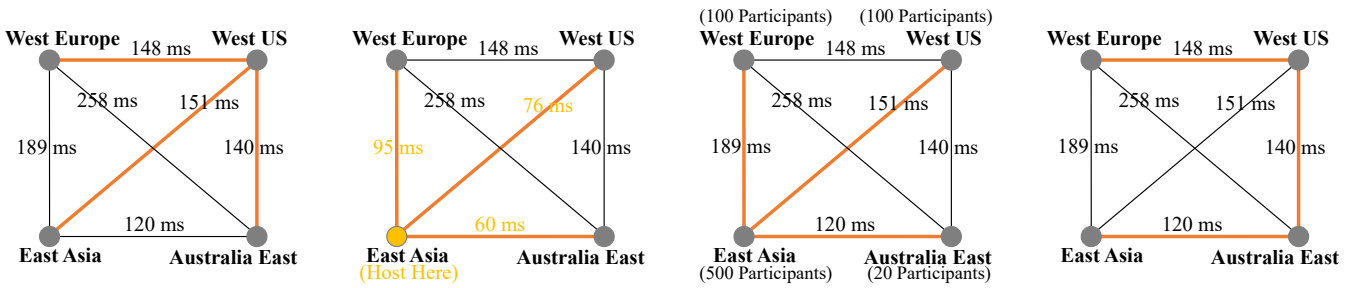


Figure 5: Illustrating example of inter-region cascading ((a), (c) and (d): all path weights equal 1; (b) path weights between East Asia and other regions equal 0.5, RTTs of three corresponding cascading paths are weight-adjusted and shown in yellow text)

to find the optimal solution to the optimization objectives. It enforces a cascading structure similar to that in the intra-region scenario: one of the region delegates acts as the master delegate, and directly cascades to other delegates. On that basis, the unweighted objective in §4.2.1 is adopted while selecting the master delegate. We postpone exploration on the necessity and effectiveness of tuning path weights to future work. By doing so, the needed computation is determined by the number of regions per room, which is usually a relatively small number in practice. We do not choose the objective in §4.2.2, because it is sensitive to fluctuations in $\mathcal{N}$.

In Figure 5, our heuristic actually generates the optimal spanning tree under the objective in §4.2.1 (Figure 5a&5b).

4.3 Reaction to Room Dynamics

To lower the workloads of engineering development and complexity of failover, our implementation tries to avoid disassembling established cascading links in reaction to dynamics in the involved regions. Next, we elaborate on how we handle participant joining and leaving, respectively.

4.3.1 Participant Joining

We focus on how to update inter-region cascading upon newly joined participants. The intra-region case has been covered in §4.1. The following is the process of constructing an AsTree from an initially empty room.

- There is no need for inter-region cascading, until a conference involves a second region. At that time, it suffices to cascade the only two delegates.
- When the third region is added, one of the existing two delegates is selected as the master delegate (§4.2.3). Then, the third region's delegate cascades to the master, without influencing the established cascading link.
- As more regions join, we stick to the same master delegate, unless the spanning tree cannot fulfill an empirical threshold of end-to-end audio latency.

We understand that starting from the third region, the selected master delegate may not be the optimal choice under the heuristic in §4.2.3 and the objective in §4.2.1 without changing established cascading links. Yet fortunately, in practical deployment, we find the need to change master delegate is rare, especially considering that most participants are from the same country as the first joiner in a majority cases [20], and the host tends to join in the beginning of a conference.

4.3.2 Participant Leaving

Ideally, if the last participant connected to a media server in a room goes offline, the media server can be removed from the corresponding AsTree. For brevity, we call such a media server an idle server specific to the AsTree topology.

It is safe to remove an idle server if it is a leaf node in the AsTree. That is the case if it is not the region delegate, or it is the only media server in its region but not the master delegate. Meanwhile, the intra-region or inter-region cascading connections on that idle server can as well be closed, if they do not carry other multiplexed traffic.

Otherwise, the idle server is a region delegate or the master delegate, and cascades other media server(s). In the former case, if we re-select the delegate, only participants under the new delegate will become closer to others by one intra-region cascading link. Provided the latency requirements of conferencing applications nowadays, we assess that such a benefit does not compensate for the complexity of tearing down and re-establishing cascading links. Therefore, we retain the role of the idle server as region or master delegate, since that does not worsen remaining participants' QoE anyway.

As more participants leave, those idle delegate servers can be removed from the AsTree topology when the media servers below them are all removed, making them the leaf nodes.

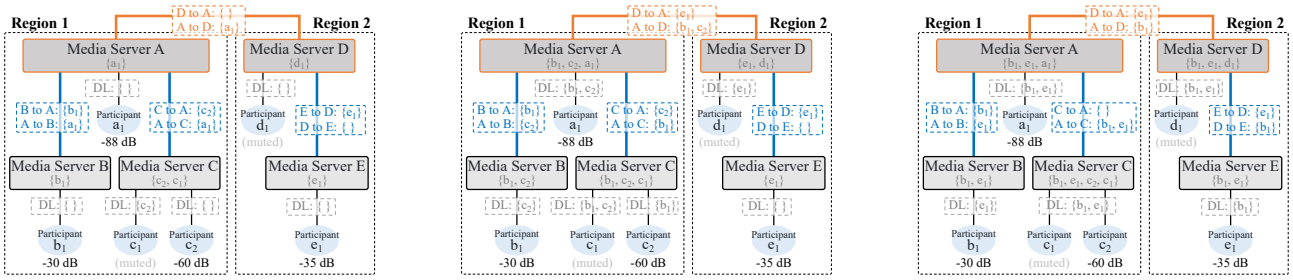
5 Select-Before-Forward Audio Selection

Different from strawman solution 3 in §2.2, AsTree expands audio selection to all cascading links. Intuitively, "select before distribute" is replaced by more generalized "select before forward". In this section, we illustrate the audio selection process, as well as how we implement dominant speaker identification and control signaling overheads.

5.1 Hop-by-Hop Audio Selection Process

All media servers adhere to a unified audio selection logic, irrespective of their roles in the cascading tree. They always select audio streams out of the following set:

- Those ingested from the connected participants.
- Those received from the cascading neighbors.



(a) Local Consensus **(b)** Intra-Region Consensus **(c)** Inter-Region Consensus

Figure 6: Illustrating the process of audio stream selection in AsTree

Then, when forwarding and distributing the selected audio streams, a media server does not send them back to where they are received. For example, a media server never sends an audio stream to its publisher again.

Another important characteristic of hop-by-hop audio selection is that it achieves asynchronous consensus on the selected active speakers among distributed media servers. Figure 6 exemplifies 3 possible states during the process (without being comprehensive). It depicts a scenario where participant a_1, b_1, c_2, and e_1 unmute together, and each media server only selects 2 audio streams for illustration.

- *Local consensus* (Figure 6a): At this phase, the unmuted audio streams are disseminated by their respective local media servers, but have not reached any cascading neighbors. So only participant c_1 hears its sibling speaker c_2.
- *Intra-region consensus* (Figure 6b): If the inter-region cascading link has much longer RTT, each region may agree on the selected intra-region active speakers, before receiving audio streams from other regions. Thus, participant c_2 is temporarily selected within region 1.
- *Inter-region consensus* (Figure 6c): In the end, every media server selects the same set of audio streams. Compared with Figure 6b, participant e_1 replaces c_2 as the second active speaker in region 1.

Figure 6 also shows that muted participants c_1 and d_1 also publish audio streams to their local media servers, which is the same as in FullAud. However, those muted audio streams are not considered in audio selection.

5.2 Dominant Speaker Identification

In Lark, RTP packets of an audio stream carry a header extension that indicates audio level [24]. It takes values from 0 to 127, representing $0 \sim -127$ dB. Media servers leverage that to select dominant active speakers, without processing raw audio. To avoid interference from non-vocal noise, we compare the weighted average audio level of each stream. It is computed from the most recent 15 audio packets (about 300 ms) of a stream, which are maintained in a per-stream ring buffer. To limit computation overheads, a media server computes a weighted average sample every 5 packets for each stream, and conducts audio selection every 50 ms (pseudo-code in Algorithm 1, with some engineering details omitted).

Instead of directly selecting L audio streams, we first pre-select at most L_i streams each time. If there are already L_i streams in the selected set S_A, a pre-selected audio stream is added to S_A only if it is louder than the least loud stream in S_A by an empirical margin (*extraCushion*). That limits the occurrence of frequent switching of selected streams. Once a stream is selected, it remains in S_A for at least a *smoothTime*. In our deployment, we set $L_i = 4$ and $L = 10$ by default.

Algorithm 1: Audio Selection

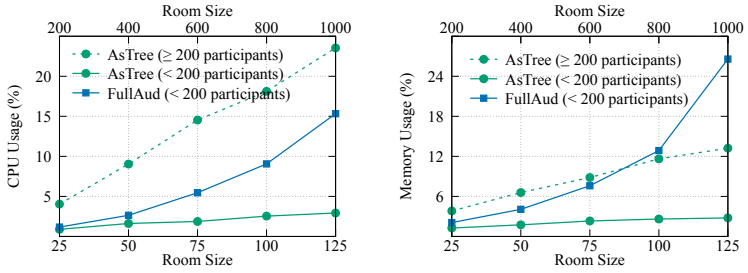
```
// S_A: set of selected audio streams
Pre-select at most L_i loudest audio streams as set S_t ;
foreach stream s ∈ S_A do
    if s ∈ S_t then
        s.lastSelected = now ;   S_t = S_t \ {s} ;
    else if now − s.lastSelected > smoothTime then
        S_A = S_A \ {s};
// louder stream has smaller audio level
foreach s ∈ S_t in order of increasing audio level do
    if |S_A| == L then break ;
    if |S_A| < L_i or s.audioLevel + extraCushion <
      GetMaxAudioLevel(S_A) then
        s.lastSelected = now ;   S_A = S_A ∪ {s} ;
```

As defined in [24], an audio level of 127 indicates a silent sample. Muted streams composed of silent samples are excluded from audio selection.

5.3 Signaling Overhead Mitigation

No Publish broadcast for audio. We do not broadcast Publish signaling messages specific to audio streams. The audio subscription process is initiated by media servers instead. Specifically, each media server caches the incoming SDP offers. Whenever a new active speaker is selected, it sends an updated SDP answer to cascaded media servers or connected participants. Immediately following that starts the transmission of audio RTP packets. Besides lowering signaling overheads, that avoids the subscription signaling round trip, and enables smooth change in the selected active speakers.

Aggregated Join. The Join message, unmerged from Publish, is solely used to notify the arrival of a new participant. When there are already a large number of participants in a room (*e.g.*, more than 50), it is both rare and hard for us to keep a close eye on every change to the participant list. Therefore, we do not necessarily need a Join broadcast specific to each newly joined participant. In that case, the local signaling unit aggregates



(a) CPU Usage **(b)** Memory Usage

Figure 7: Benchmark: AsTree uses less CPU and memory (Results of AsTree with ≥ 200 participants depicted in dashed line, refering to x2-axis on the top of the figures)

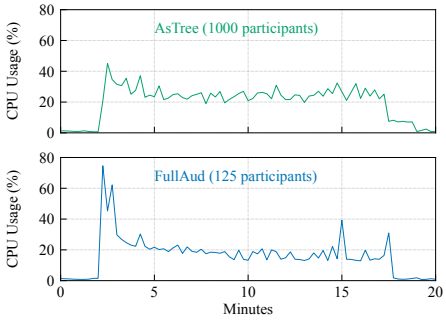


Figure 8: Example Benchmark Trials

Participant Type	Type Description
Type A	Active speakers that open both camera and microphone, and keep talking
Type B	Participants that leave their microphone unmuted but stay silent
Type C	Audience that close both the camera and the microphone

Table 2: Types of participants in benchmarking tests

newly joined participants in a short interval (*e.g.*, every 1 second) into a single signaling message. The cancellation of audio Publish and aggregation of Join dramatically mitigate signaling storm in the beginning of conferences.[3]

No Mute/Unmute signaling. In FullAud, the displayed audio status on user interface is updated strictly according to the Mute/Unmute broadcast, which has to be triggered each time a participant clicks the mute/unmute button. AsTree totally eliminates that kind of signaling broadcast. A participant is displayed as unmuted if and only if the corresponding audio stream is selected. That said, an unmuted silent participant will not be displayed as unmuted. Such a mismatch is not a very important factor in terms of user experience. Indeed, we do not receive any negative reviews on that front.

6 Benchmark

FullAud has been fully replaced by AsTree in Lark's current deployment. To compare them from a micro-level perspective, we conduct benchmarking stress tests in our test environment. Media servers there are co-located with but isolated from our production environment servers, *i.e.*, test traffic does not interfere with real-world user traffic. Three types of participants are introduced as detailed in Table 2. Through command line parameters, we can initiate a controlled number of virtual participants for each type, and specify audio/video files as their microphone/camera output. Each virtual participant runs an instance of our Linux SDK on specialized server clusters in each region. Because we only focus on evaluation of the audio subscription architecture, we need to eliminate the influence of video simulcast. Thus, we use a single-resolution (360p) video to emulate the camera video of type-A participants. Also, we limit the number of type-A participants so that all participants in a room subscribes to the same set of all published video streams. Besides, we feed two separate recorded audio tracks to type-A and type-B participants, respectively.

Unless otherwise specified, we average results of 10 runs in each test setup, to avoid interference of other test traffic.

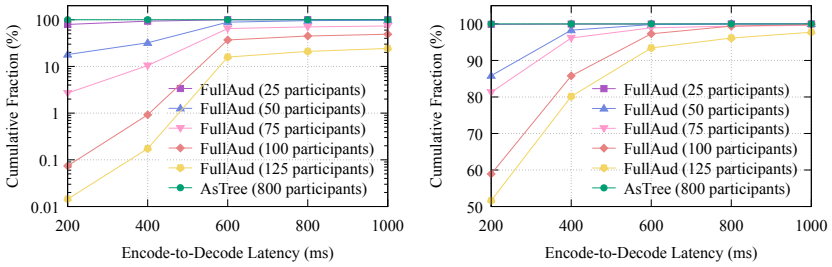
6.1 Media Server Capacity

AsTree forwards and distributes fewer audio streams and signaling messages than FullAud, and frees up considerable CPU and memory resources on media servers. To quantify that, we set up rooms with varying number of participants in a single region, all of which are forced to the same media server instance. Each conference lasts for 15 minutes, and involves 4 type-A participants, 4 type-B participants, and different number of type-C participants. In the beginning of each conference, we launch 50 participants per second, until reaching the target room size.

We first calculate average CPU and memory usage in the last 10 minutes of each trial (Figure 7). AsTree scales approximately linearly, and achieves increasingly significant gains over FullAud. That is because FullAud enforces quadratic subscriptions to all type-B and type-C participants, even though they mostly publish DTX frames. With 125 participants in a room, AsTree consumes 80.9% and 89.5% less CPU and memory than FullAud, corresponding to 5.2× conferences that can be held on a single server. Intuitively, the CPU busy time occupied by a 600-participant conference using AsTree is barely not enough for 125 participants under FullAud; the memory consumed by a 1000-participant AsTree-based conference can only support 100 participants under FullAud.

Figure 7 accounts for consumption of media processing in steady state. The capacity of a single media server also depends on the signaling overheads upon participants joining. We present the trend of CPU usage from a 125-participant conference under FullAud in Figure 8. Although it stabilizes at around 15% CPU usage, the initial peak is as high as 75% when many participants join together. In fact, given how we initiate a conference (*i.e.*, participants joining in bursts of 50), the media server instance crashes in the process of accommodating 150 participants using FullAud. That is why we only show FullAud's results with up to 125 participants in Figure 7. In comparison, the upper portion of Figure 8 exemplifies a 1000-participant room based on AsTree. The initial signaling peak is not obvious, despite having 8× participants.

[3] Leave/Unpublish in FullAud is replaced similarly.



(a) Audio **(b)** Video

Figure 9: Benchmark: AsTree reduces end-to-end latency

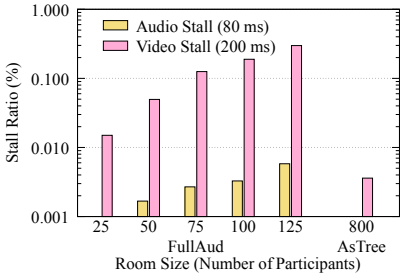


Figure 10: Benchmark: AsTree lowers stall ratio for both audio and video

Taking both media and signaling processing into consideration, AsTree can increase both the maximum conference scale and the number of concurrent rooms supported by a single server instance by as high as an order of magnitude, especially when it comes to massive-scale conferences.

6.2 User Experience

The decreased volume of traffic forwarded and distributed by AsTree mitigates cascading and last-mile congestion. That should eventually improve both audio and video QoE. To validate that, we test with a similar room setup as above, with 4 type-A participants, 4 type-B participants, and various type-C participants. This time, we load-balance participants to multiple media servers within the same region, instead of sticking to a designated media server.

We start with end-to-end latency (Figure 9), measured as the duration from when a media frame is encoded at publisher to when the same frame is decoded at subscriber. For AsTree, all audio and video frames are decoded within 200 ms. Yet due to interference from signaling storm and super-linear number of subscribed audio streams, the performance of FullAud deteriorates quickly as the room size grows. For instance, with 125 participants per room, merely 0.014% of audio frames are decoded within 200 ms, and the proportion is 51.6% for video frames. Even when we relax the latency target to 1000 ms, a 800-participant AsTree-based conference is 312% and 2.3% better than a 125-participant FullAud-based conference in terms of audio and video latency, respectively.

We note that the DTX frames from type-B and type-C participants, which have the lowest transmission and decoding priority, are also included in calculation. That leads to the extremely high audio latency by FullAud in Figure 9a. However, our test results still correctly reflect the performance trend of FullAud under different room sizes. That can be confirmed by the video latency in Figure 9b, as well.

Furthermore, lower end-to-end latency contributes to fewer stalls. In Figure 10, we plot the audio and video stall ratios, computed as the percentage of playback time in which the stall is longer than 80 ms and 200 ms, respectively. FullAud accomplishes zero audio stall only under a room size of 25 participants, while AsTree maintains zero audio stall with 800 participants. As for video stall, AsTree with 800 participants achieves a dramatic reduction of nearly two orders of magnitude, compared with FullAud with 125 participants.

Therefore, by optimizing the architecture of audio subscription, AsTree simultaneously delivers better audio and video experience at a larger conference scale.

7 Deployment

AsTree's initial deployment in Lark dates back to August 2021. It completely replaced FullAud in March 2022, and has been running at full scale since then for over two years. To demonstrate its effectiveness in improving user experience, we present the trend of three key indicators over 30 weeks ending in January 2022 in Figure 11. During that period, Lark served more than 100 million conferences. All data are normalized for confidentiality. We note that there were several parallel optimizations in Lark's audio codec starting late January 2022. We do not show results beyond that to isolate the gains by AsTree.

QoE Metrics. The audio and video stall ratios in Figure 11 account for stall duration longer than empirical thresholds of 80 ms and 200 ms, respectively. Over the presented period after AsTree was deployed, they were improved significantly at the same time: the median and 95th percentile audio stall ratios were reduced by more than 30% and 45%; corresponding reductions in video stall ratios both exceeded 50%. In practical deployment, an important factor contributing to such improvements is that AsTree alleviates congestion on participants' access links, which usually have more limited bandwidth than our test environment for stress tests (§6.2).

Additionally, we present the audio stall ratio partitioned in different range of room sizes in Figure 12. It is averaged from a recent month. We can see that a conference with more than 350 participants does not even double the audio stall ratio over a small-scale conference with only 3 to 5 participants. We regard that as acceptable for a global system like Lark, considering increasingly heterogeneous participant access bandwidth and geo-distributed regions under larger rooms.

User Satisfaction. Along with better QoE comes higher user satisfaction. We observed a reduction of around 40% in the ratio of negative reviews in Figure 11. The ratio of negative reviews complaining specifically about bad audio or video experience was also reduced by 40%, closely following the trend of overall negative reviews. That indicates the QoE gains by AsTree were the dominating reason that raised our

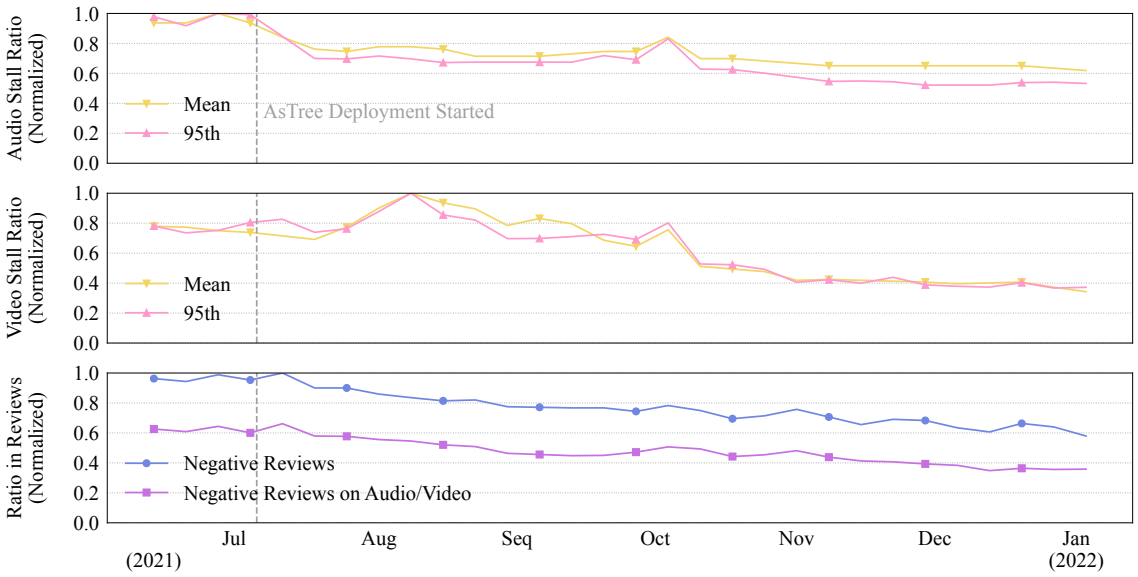


Figure 11: Trend of stall ratios, and negative review ratio after AsTree was deployed (median and 95th stall ratios are normalized independently)

user satisfaction. Moreover, if we look at the ratio of negative reviews under different room sizes (Figure 13, averaged from a month for both architectures), AsTree effectively narrows the gap of user experience under large and small conferences.

Cascading Complexity. A remaining concern on AsTree is whether it induces excessive cascading links and impacts user experience, *i.e.*, due to selection of region and master delegates. Enhanced user satisfaction shown above implies that does not occur to most participants. To confirm that more directly, we monitored the number of cascading links needed to forward and distribute a subscribed stream for two weeks earlier in 2024. On average, Lark only increases the number of cascading links per stream by 5.7% compared with other ByteDance RTC applications. Thus, the AsTree cascading topology has restricted adverse influence on QoE, indeed.

8 Discussion

In this section, we share some practical insights drawn from deployment experiences, and potential future work directions.

8.1 More Implementation Details

Participant aggregation. Because signaling storm is significantly mitigated, AsTree enables us to aggregate more participants on a single media server, so as to reduce the number of intra-region cascading links. Participant aggregation saves server-side costs on media servers and bandwidth, and lowers the complexity of participant migration upon failover. While achieving aggregation, we may need to redirect a joining participant to a different media server from the one retrieved through DNS. In that case, we notify the participant with a customized signaling message. Other approaches, such as DNS redirection as adopted by many CDN systems [12, 25], may also fulfill the same purpose. Meanwhile, to avoid overloading a single media server, we configure the upper limit of aggregation to 200 participants per room per server.

Cascading connection reuse. Before forwarding audio streams to and from a newly involved media server, we need to establish a cascading connection and accomplish SDP negotiation. To reduce latency and start media transmission as soon as possible, we try to reuse existing connections on the same cascading link. That means a cascading connection may carry audio streams of multiple conference rooms. Two corresponding configuration parameters are introduced. First, we restrict the maximum number of (RTP) streams allowed on a cascading connection, in case they impact the processing efficiency of the thread worker. Second, we set a timeout of several seconds before closing an idle cascading connection, during which it may be reused again.

Decoupled video and audio subscription. In the media plane, the AsTree cascading tree topology is independent of the routing and forwarding of video streams. In the control plane, the bandwidth reserved for audio streams before allocating to video streams of different bitrates is bounded and thus, more predictable. That ensures great flexibility for innovations in either video or audio subscription architecture. We do not need to worry about interfering video simulcast while implementing and deploying AsTree. As for the synchronization of video and audio streams of the same publisher, we rely on jitter buffer on client side. That fulfills the current latency requirements of multi-party conferencing.

Fault tolerance. Fault tolerance in AsTree is in essence the same as in video simulcast, *i.e.*, ensuring connectivity between each pair of participants. We mainly handle two types of failures, which are broken cadcading link and crashed server/cluster. The routing of each overlay cascading link is managed by our centralized SDN controller. When we observe abnormal RTT inflation and loss rate on a specific cascading link, we switch to a back-up path without awareness of conference participants and SFU media servers. If a

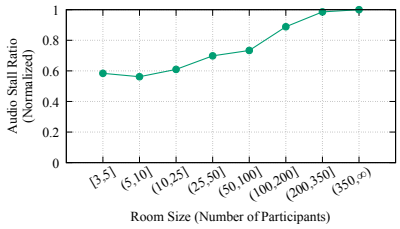


Figure 12: Audio stall ratio after AsTree is fully deployed

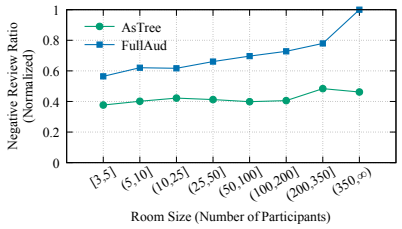


Figure 13: Negative review ratios before/after AsTree's deployment

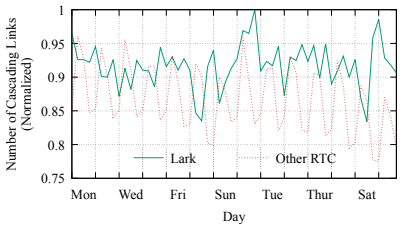


Figure 14: Number of cascading links per stream subscription

media server or a cluster crashes, participants already connected to a faulty server will go through a connection timeout, triggering a request to the control plane for a new SFU. The original faulty server is replaced by the new SFU in the AsTree topology, and overlay cascading paths are re-established. In that process, the faulty server(s) will be blacklisted by our scheduler until they recover from connection issues.

8.2 Other Implementation Options

Mesh *vs.* AsTree. The select-before-forward audio selection as detailed in §5 can also work when media servers are cascaded by a mesh. In that case, each media server selects audio streams from those injected from its connected participants, and forwards to all the other media servers. Compared with an AsTree, a mesh topology has shorter cascading paths on average, but induces more cascaded audio streams. For example, a conference room involving M media servers cascades at most $2L(M-1)$ audio streams with a cascading tree, which is no larger than $L \cdot M(M-1)$ when using a mesh. Considering that AsTree can satisfy the latency requirements of multi-party conferencing, we adopt it for lower deployment cost.

Audio stream SSRCs. Some existing service providers use a fixed set of negotiated RTP SSRCs [26] for the selected audio streams [27, 28]. Our web SDK also adopts similar solution, due to lower development workloads based on the WebRTC codebase. However, that requires the media server to make sure the participants receive consecutive sequence number and monotonous timestamp under each SSRC. To avoid such processing overheads, while implementing the native SDK of Lark application, we choose to let each media server generate a random SSRC each time an audio stream is selected, even if it is from a speaker that is previously selected. That said, different participants may observe different SSRCs for the same selected audio stream at the same moment; and the same participant may observe different SSRCs for the same speaker during a conference. The identity of each selected speaker is notified to participants through the updated SDP answer triggered by the media server (§5.3).

8.3 Effectiveness and Applicability

Benefits for all conference scales. Although designed to support massive-scale conferences, AsTree achieves significant gains in small-size conferences, as well. That is because a muted audio stream, although requiring lower bandwidth, consumes almost the same amount of memory and comparable CPU as an audio stream from an active speaker. Therefore, AsTree can serve as a general-purpose architecture that benefits a broad spectrum of cases, and we enable it by default.

Extended application scenarios. AsTree has been adapted to support more advanced use cases. An example is to provide simultaneous interpretation for foreign participants in international conferences. A channel attribute can be added to the original and interpretation audio streams. Participants can subscribe to the original channel, the interpretation channel, or both channels on demand from Lark user interface, and selection of audio streams at each media server can be conducted accordingly on a per-channel basis.

8.4 Future Directions

Joint video and audio optimization. Although audio and video subscription are decoupled in media plane, there may be QoE benefits to jointly optimize the cascading topology of video and audio streams in control plane. For example, in a typical conference where an invited speaker presents slides via screen sharing, the audio and screen video streams of the presenter may share a cascading tree rooted at the corresponding local media server, which also facilitate the synchronization of the speaker's audio and video signals.

Dynamic AsTree topology deformation. Given latency requirement of multi-party conferencing nowadays (*e.g.*, up to hundreds of milliseconds), we adopt a straightforward heuristic to select intra-region and master delegates, and rarely change the role of each media server (§4.3) during an ongoing conference. Yet doing so improves user experience in some cases. In the above example with an invited speaker, if we continue to the next speaker on the agenda who is in another region, it lowers the latency from the new speaker to the audience to switch the master delegate, as well. Such a deformation may even become inevitable as we keep pursuing more real-time interactivity. Therefore, we plan to work on a highly efficient deformation mechanism based on an established cascading topology. In the example of switching to a new speaker, it may be triggered by dynamically assigned cascading path weights. Besides, it can also be used for media server migration and failover.

Formulation of AsTree calculation. Similar cascading topology may also apply in other application scenarios such as live streaming. Our current objectives listed in §4.2 only optimize

the cascading path latency. When aiming at more latency-sensitive future-proof applications, we may need to refine the problem formulation and express the QoE target more accurately. For example, an additional term representing per-media server processing latency may be added, if that accounts for a fair proportion in the overall end-to-end latency. For compute-intensive applications (*e.g.*, cloud rendering [29, 30], artificial video generation [31]), the computing resource consumption may be explicitly captured, especially when leveraging limited computing power at the edge [32]. Furthermore, in extreme cases with multiple concurrent continuous publishers, a more complicated topology than a single tree may be needed.

9 Related Works

Dominant speaker identification (DSI). DSI is not a new problem in the context of multi-party conferencing. Volfin *et al.* identified the dominant speaker by computing per-participant immediate, medium-term, and long-term speech activity scores [11]. However, the computation was based on frequency representation of audio frames. That is not available on an SFU without decoding audio streams. Inspired by [11], Jitsi introduced LastN [10], and computed speech activity score directly from audio levels carried in RTP header [24]. Different from AsTree, LastN was used to make sure only dominant speakers' video streams are forwarded. It remained full subscription for audio streams. Similar DSI method is often utilized in video subscription to track and render the active speaker view [33, 34].

Audio selection. Full audio subscription was adopted by many service providers previously, some of which stick to that till now (*e.g.*, Twilio [8]). As the popular conference scale continues to grow, several service providers have started to support audio selection based on DSI in recent years. The audio mixer in early-stage WebRTC codebase mixed at most 3 audio sources [35], but it ran on client side. Google Meet and Google Duo forwarded the 3 loudest speakers with fixed SSRCs [27]. Jitsi introduced a "loudest" feature, which only forwards a configured number of loudest audio streams [36]. Agora allowed audio-strength stream selection based on audio volume [37]. Nevertheless, limited details are revealed about those schemes. Some providers such as Jitsi and Agora still disable audio selection in their default configuration. AsTree is the first to share the architecture and performance of audio selection from a large-scale practical deployment.

Conferencing architecture. For audio subscription, there are still a few services implementing Multipoint Control Unit (MCU) to get a single mixed audio streams [38, 39]. Yet for video subscription, SFU has been the choice by a majority of popular conferencing providers [6, 16, 17, 40], due to lower overheads on server side. Both simulcast and SVC can rely on the SFU architecture. In simulcast, each participant encodes multiple video streams in different bitrates, and the SFU determines which stream to forward to which participant. A representative example is GSO-Simulcast [5], where a centralized controller orchestrated the resolution and bitrate of video streams. SVC leverages scalable video codec [41] to include multiple layers in a single video stream, so that it can be adapted to multiple bitrates. Zoom is claimed to belong to this category [42]. AsTree is compatible with the SFU-based video subscription architecture.

In addition, focusing on routing in the media plane, existing works such as LiveNet [12] and VIA [43] constructed overlay cascading paths, enabling participants to access the service through the nearby edge. Aimed at resource provisioning for conferencing services, Switchboard [20] enforced peak-aware server allocation and capacity provisioning. Those works are complementary to AsTree.

Measurements of multi-party conferencing. Recognizing the importance of multi-party conferencing and the challenges it brings to the network, researchers conducted extensive measurements of popular applications on the market [6, 44–48]. On one hand, by analyzing their protocol encapsulation and session behaviors from collected traces, they confirmed the usage of simulcast by many conferencing applications, including Google Meet, Microsoft Teams, WebEx [44]. On the other hand, owing to lack of abundant resources of media servers, it is hard to capture the global picture under massive conference scales. For example, authors of [46] conducted small-scale experiments with only up to 6 participants.

10 Conclusion

In this paper, we present our design of AsTree, the architecture specifically for audio subscription aimed at massive-scale multi-party conferencing in Lark. AsTree evolved from our early-stage implementation of FullAud, which was based on full audio subscription. For each conference room, it constructs a two-hierarchy (intra-region and inter-region) cascading tree composed of all involved media servers, and enforces per-hop audio selection while forwarding audio streams. AsTree eliminates the signaling broadcasts triggered by participants' muting/unmuting operations.

After serving millions of conferences in several years of deployment, AsTree is proved to significantly improve the scalability of Lark. Thanks to dramatically mitigated signaling storm and fewer number of audio streams to forward, AsTree improves the capacity of media servers by nearly an order of magnitude compared with FullAud. Meanwhile, less congestion on last-mile and cascading links results in enhanced QoE. In our practical deployment, the median audio and video stall ratios are reduced by 30% and 50%, respectively. That comes with a 40% reduction in the ratio of negative reviews.

To fulfill even more stringent latency requirements, we envision more flexible and dynamic cascading topology calculated from more realistic QoE objectives in the future.

Acknowledgements

We thank our shepherd, Kurtis Heimerl, and the anonymous SIGCOMM reviewers for their valuable comments.

References

[1] Lark. https://www.larksuite.com/. Accessed: 2024.09.

[2] Zoom. https://zoom.us/. Accessed: 2024.09.

[3] Microsoft Teams. https://www.microsoft.com/en-us/microsoft-teams/group-chat-software. Accessed: 2024.09.

[4] Alibaba DingTalk. https://www.dingtalk.com/. Accessed: 2024.09.

[5] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. GSO-simulcast: global stream orchestration in simulcast video conferencing systems. In *ACM SIGCOMM*, 2022.

[6] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. Enabling passive measurement of zoom performance in production networks. In *ACM IMC*, 2022.

[7] Jean-Marc Valin, Koen Vos, and Timothy B. Terriberry. Definition of the Opus Audio Codec. RFC 6716, September 2012.

[8] Twilio Track Subscriptions. https://www.twilio.com/docs/video/api/track-subscriptions. Accessed: 2024.09.

[9] Alibaba RTC SDK. https://www.alibabacloud.com/help/en/live/user-guide/android-alirtcengine-class#0155f66448env.

[10] Boris Grozev, Lyubomir Marinov, Varun Singh, and Emil Ivov. Last N: relevance-based selectivity for forwarding video in multimedia conferences. In *ACM NOSSDAV*, 2015.

[11] Ilana Volfin and Israel Cohen. Dominant speaker identification for multipoint videoconferencing. *Computer Speech & Language*, 27(4):895–910, 2013.

[12] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, et al. Livenet: a low-latency video transport network for large-scale live streaming. In *ACM SIGCOMM*, 2022.

[13] Boris Grozev. Improving Scale and Media Quality with Cascading SFUs. https://webrtchacks.com/sfu-cascading/. Accessed: 2024.09.

[14] Bandwidth Requirements for Lark Video Meetings. https://www.larksuite.com/hc/en-US/articles/360048488053-what-are-the-bandwidth-requirements-for-video-meetings. Accessed: 2024.09.

[15] Boris Grozev. Bridge cascading with geo-location is back. https://jitsi.org/blog/bridge-cascading-is-back/, 2022.

[16] Jitsi Videobridge. https://jitsi.org/jitsi-videobridge/. Accessed: 2024.09.

[17] Amazon Chime Video Simulcast. https://aws.github.io/amazon-chime-sdk-js/modules/simulcast.html. Accessed: 2024.09.

[18] Sudarshan Vasudevan, Jim Kurose, and Don Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *IEEE ICNP*, 2004.

[19] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *ACM-SIAM SODA*, 2006.

[20] Rahul Bothra, Rohan Gandhi, Ranjita Bhagwan, Venkata N Padmanabhan, Rui Liang, Steve Carlson, Vinayaka Kamath, Sreangsu Acharyya, Ken Sueda, Somesh Chaturmohta, et al. Switchboard: Efficient Resource Management for Conferencing Services. In *ACM SIGCOMM*, 2023.

[21] Recommendation ITU-T G.114. One-way transmission time. Telecommunication standardization sector, International Telecommunication Union, May 2003.

[22] Twilio Glossary: What is Latency? https://www.twilio.com/docs/glossary/what-is-latency. Accessed: 2024.09.

[23] Azure Network Latency. https://learn.microsoft.com/en-us/azure/networking/azure-network-latency. Accessed: 2024.05.

[24] Jonathan Lennox, Enrico Marocco, and Emil Ivov. A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication. RFC 6464, December 2011.

[25] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM*, 2015.

[26] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003.

[27] Gustavo Garcia. Meet vs. Duo – 2 faces of Google's WebRTC. https://webrtchacks.com/meet-vs-duo-2-faces-of-googles-webrtc/, 2022.

[28] Jaya Allamsetty. Improving performance on very large calls: introducing SSRC rewriting. https://jitsi.org/blog/improving-performance-on-very-large-calls-introducing-ssrc-rewriting/, 2024.

[29] Serhan Gül, Dimitri Podborski, Jangwoo Son, Gurdeep Singh Bhullar, Thomas Buchholz, Thomas Schierl, and Cornelius Hellge. Cloud rendering-based volumetric video streaming system for mixed reality services. In *ACM MMSys*, 2020.

[30] Yongjie Guan, Xueyu Hou, Nan Wu, Bo Han, and Tao Han. MetaStream: Live Volumetric Content Capture, Creation, Delivery, and Rendering in Real Time. In *ACM MobiCom*, 2023.

[31] Sora: Creating video from text. https://openai.com/index/sora/. Accessed: 2024.09.

[32] Na Li and Yao Liu. EVASR: Edge-Based Video Delivery with Salience-Aware Super-Resolution. In *ACM MMSys*, 2023.

[33] Zoom SDK. https://zoom.github.io/zoom-sdk-android/us/zoom/sdk/InMeetingServiceListener.html#onActiveSpeakerVideoUserChanged-long-. Accessed: 2024.09.

[34] Azure Communication Services SDK: Get active speakers within a call. https://learn.microsoft.com/en-us/azure/communication-services/how-tos/calling-sdk/dominant-speaker. Accessed: 2024.09.

[35] WebRTC Audio Mixer. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/modules/audio_mixer/audio_mixer_impl.h;l=50;drc=0df0faefd5ea4df2364cccc0f8449431bf8200d0. Accessed: 2024.09.

[36] Jitsi Loudest: Only forward audio packets of the loudest speakers in a conference. https://github.com/jitsi/jitsi-videobridge/pull/1677/commits. Accessed: 2024.09.

[37] Agora: Audio Strength Stream Selection. https://docs.agora.io/en/video-calling/advanced-features/audio-strength-stream-selection?platform=android. Accessed: 2024.09.

[38] Amazon Chime Meeting Architecture. https://docs.aws.amazon.com/chime-sdk/latest/dg/meetings-sdk.html#mtg-arch. Accessed: 2024.09.

[39] Quobis: Hybrid architecture based on MCUs and SFUs. https://quobis.com/2021/05/03/sfus-vs-mcus-which-is-the-best-way-to-manage-multi-conferencing/. Accessed: 2024.09.

[40] An introduction to WebRTC Simulcast. https://blog.livekit.io/an-introduction-to-webrtc-simulcast-6c5f1f6402eb/, 2021.

[41] Scalable Video Coding (SVC) Extension for WebRTC. https://www.w3.org/TR/webrtc-svc/. Accessed: 2024.09.

[42] A Short on How Zoom Works. https://highscalability.com/a-short-on-how-zoom-works/. Accessed: 2024.09.

[43] Junchen Jiang, Rajdeep Das, Ganesh Ananthanarayanan, Philip A Chou, Venkata Padmanabhan, Vyas Sekar, Esbjorn Dominique, Marcin Goliszewski, Dalibor Kukoleca, Renat Vafin, et al. Via: Improving internet telephony call quality using predictive relay selection. In *ACM SIGCOMM*, 2016.

[44] Antonio Nistico, Dena Markudova, Martino Trevisan, Michela Meo, and Giovanna Carofiglio. A comparative study of rtc applications. In *IEEE International Symposium on Multimedia (ISM)*, 2020.

[45] Insoo Lee, Jinsung Lee, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Demystifying commercial video conferencing applications. In *ACM Multimedia*, 2021.

[46] Hyunseok Chang, Matteo Varvello, Fang Hao, and Sarit Mukherjee. Can you see me now? a measurement study of zoom, webex, and meet. In *ACM IMC*, 2021.

[47] Mehdi Karamollahi, Carey Williamson, and Martin Arlitt. Packet-level analysis of zoom performance anomalies. In *ACM/SPEC ICPE*, 2023.

[48] Albert Choi, Mehdi Karamollahi, Carey Williamson, and Martin Arlitt. Zoom session quality: A network-level view. In *PAM*, 2022.

AutoCCL: Automated Collective Communication Tuning for Accelerating Distributed and Parallel DNN Training

Guanbin Xu[†] Zhihao Le[†] Yinhe Chen[†] Zhiqi Lin[†] Zewen Jin[†] Youshan Miao[‡]
Cheng Li[†,◦]
[†]*University of Science and Technology of China,* [‡]*Microsoft Research*
[◦]*Anhui Province Key Laboratory of Biomedical Imaging and Intelligent Processing*
Institute of Artificial Intelligence, Hefei Comprehensive National Science Center

Abstract

The collective communication libraries are pivotal in optimizing the performance of distributed and parallel deep neural network (DNN) training. Most network optimizations are under the assumption that these libraries are well-tuned, ignoring their low-level parameter selection. In this paper, we present a novel automated tuning method AutoCCL that significantly improves communication performance without incurring additional costs. One of the primary challenges we tackle is the state explosion in searching for the optimal configuration. To overcome this, we decouple implementation-related parameters from those sensitive to the search space size and propose a divide-and-conquer algorithm, minimizing the requirement for exhaustive trials. We further propose an online tuning approach that accounts for communication-computation interference to enhance accuracy in finding optimal configurations, while hiding tuning overhead within early iterations of training jobs. We implement AutoCCL atop NCCL, a leading and widely-used communication library provided by NVIDIA. Our evaluation on both a 2-node cluster (16 A40 GPUs, intra-node NVLink, inter-node 2× 400Gbps InfiniBand) and a 4-node cluster (32 A40 GPUs, intra-node PCIe, inter-node 100Gbps InfiniBand) demonstrates that AutoCCL achieves 1.24-1.29× and 1.15-1.22× speedups on microbenchmarks compared to NCCL and another SOTA NCCL tuner, respectively, and up to 1.80× and 1.49× with concurrent computation. End-to-end evaluations on three large language models and one vision model show 1.07-1.32× improvements in per-iteration training time.

1 Introduction

Training contemporary DNN models often distributes computation over a cluster of GPUs and relies on collective communication primitives to frequently exchange model data in a group-wise manner. However, many studies have pointed out that communication is a well-known bottleneck that throttles training performance [14, 26, 27, 33, 34, 36].

Communication optimization has become a research hotspot, garnering growing interest including communication-computation overlapping [13, 19, 22, 23, 31, 43, 46, 50, 59, 60, 67], message compression [9, 53, 56, 62], and new collective algorithm design [10, 12, 15, 16, 27–29, 32, 35, 42, 47, 57]. These proposals assume the off-the-shelf communication libraries such as NCCL [5] to be already well-tuned. Some works [12, 47] have attempted to provide empirical guidance, but in Section 3.1, we have found that the guidance is not always correct in practice. Besides, our study highlights that collective communication tasks can be further improved, e.g., up to 35% higher bandwidth, by tuning low-level performance-sensitive parameters. Unfortunately, these libraries are often used as black boxes, and the potential for tuning them across different hardware and communication tasks is overlooked.

In this paper, we propose AutoCCL, a tool for automatically tuning collective communication with accelerated end-to-end training performance. Compared to existing communication optimizations, AutoCCL aims to provide *a free lunch* and transparently support upper-level training jobs, maintaining model accuracy without requiring additional hardware investment. However, AutoCCL's design faces the following challenges:

First, existing work lacks a comprehensive analysis of the mainstream collective communication libraries to identify key performance-sensitive parameters as tuning candidates, understand the joint impact of these parameters, and establish practical tuning guidelines. Second, computation and communication are executed concurrently during training. Therefore, tuning for communication also necessitates accounting for computational interference. This understanding needs to be further adapted to different hardware, as well as the dynamics in the concurrent combination of computation and communication due to runtime scheduling.

To address the above challenges, we propose the following innovations. First, we conduct a comprehensive tuning study (Section 2.3 and Section 3) to identify six low-level NCCL parameters that are performance-sensitive, forming a very large search space. Then, we experimentally analyze and theoretically model these parameters and the relationships among them and make some important and instructive

This research project began in June 2023.

observations. For example, the parameters can be classified into two categories, one defining the implementation method of communication primitives and the other determining the resource allocations. The former is a prerequisite for the latter, and the former has a small range of values while the latter has a large range of values. In addition, we analyze the resource parameters using the control variable method and find that their combination is characterized by a unimodal function on the communication bandwidth. These findings are far beyond a recent preliminary study [64].

Second, guided by the aforementioned analysis, we design an efficient tuning algorithm (Section 4) that avoids a brute-force search within the parameter space. This algorithm handles implementation parameters and resource parameters separately. Since the combinations of the former are relatively few and the analysis of resource parameters must begin with the determination of the communication task, the tuning algorithm first divides the implementation parameters into several subspaces, each representing an independent search task. Within any given subspace, we employ a coordinate descent method to search for the optimal combination of resource parameters. This approach is feasible because the characteristics of unimodal functions indicate that their combination to the communication bandwidth curve will initially increase, then reach a peak, and subsequently decrease gradually until reaching a steady state (at which point bandwidth becomes the bottleneck).

Finally, we propose an online tuning method (Section 5) that co-runs with training. We exploit the iterative nature of DNN training (where identical communication tasks are repeated throughout the training process) to integrate the tuning process into the initial training iterations. Once the optimal configuration is obtained, it is updated via atomic broadcast to all nodes participating in collective communication for consistent future usage. The advantage of this online approach is that we can directly capture the performance of communication primitives under the influence of computational interference and scheduling dynamics. After integrating this into the tuning algorithm, the performance analysis achieves higher accuracy, thus eliminating the need to model uncertain factors directly, such as hardware capabilities and the concurrency of tasks in the real training environment.

We implemented the tuning tool `AutoCCL` on top of NCCL, a leading and widely adopted collective communication library. `AutoCCL` maintains compatibility with the NCCL interface, allowing it to seamlessly integrate with training frameworks like PyTorch and transparently support any DNN model training task. We've made it open-source `AutoCCL` [2]. We conducted extensive testing on a 4-node cluster with 32 A40 GPUs and a 100Gbps network, as well as a 2-node cluster with 16 A40 GPUs and two 400Gbps links. Pure communication experiments with different communication primitives show that, compared to NCCL and AFNFA [64] (the most recent NCCL tuner), our communication bandwidth speedups are 1.24-1.29× and 1.15-1.22× higher, respectively. In microbenchmark experiments with computational interference, the results show that `AutoCCL` achieves even greater average gains, with improvements of 24.1% and 20.7%, respectively. Additionally, we conduct end-to-end training tests on three popular large-scale models and one computer vision model. Compared to NCCL and AFNFA, `AutoCCL` achieves average training speed improvements of 16.7% and 14.5%, respectively, with gains of up to 32% on some large models. Note that the training gains have taken tuning overhead into account, and since `AutoCCL` converges quickly, the tuning overhead is negligible.

In summary, we make the following contributions:

- A comprehensive study that uncovers the tuning guidelines of low-level performance parameters of collective communication primitives in NCCL, a leading communication library;
- An online tuning method that implements the parameter subspace division and intra-subspace coordinate descent search algorithms and leverages training iterativeness for accurately modeling the impacts of parameter assignment in the context of computational interference;
- The `AutoCCL` tuner that incorporates this tuning method and transparently supports training jobs;
- An in-depth evaluation of `AutoCCL` compared to NCCL and another state-of-the-art tuner on communication micro-benchmarks and representative training jobs.

2 Background

2.1 Distributed and Parallel DNN Training

It has been a common practice to distribute and parallelize DNN training jobs over a cluster of GPUs for fast model parameter updating [17, 20, 24, 25, 30, 37–39, 44, 45, 48, 49, 51, 55, 58, 66]. For instance, as shown in Table 1, when the model size does not exceed a single GPU's memory budget, *data parallelism* is often employed to let GPUs consume disjoint data partitions for collective model training. This requires GPUs to frequently synchronize gradients, with the transmission volume equal-sized with model parameters. When a model's memory consumption exceeds the GPU capacity, in addition to *data parallelism*, AI participants employ either *tensor parallelism* or *pipeline parallelism* to partition a model into smaller computational units, which are distributed across various GPUs. This results in more complex communication patterns among GPUs, which are often on the critical path of the training pipeline, compared to *data parallelism*. Recently, various studies have already proved that communication is the major bottleneck that limits the scalability of distributed and parallel DNN training [14, 26, 27, 33, 34, 36].

A wide range of optimizations aim to address communication bottlenecks, including scheduling [13, 19, 22, 23, 31,

Table 1: The parallelism and collective communication primitives used by training 4 DNN models on a cluster of 32 GPUs (four machines). *B* stands for billion model parameters. *TP*, *DP*, and *PP* stand for tensor parallelism [48], data parallelism [32], and pipeline parallelism [20], respectively. The numbers in parentheses following these parallelism approaches represent the corresponding number of GPU participants. For each collective communication primitive, its (*x_y_z*) value represents that an *x*-sized message is exchanged among a *y*-sized communication group for *z* times per training iteration.

Model	Phi-2-2B	Llama-3.1-8B	Yi-1.5-34B	VGG-19-0.14B
Parallelism[1]	TP (8) + DP (4)	TP (8) + DP (4)	TP (8) + PP (4)	DP (32)
AllGather	(80 MB_8_3,120)	(128 MB_8_6,240), (857 MB_4_1)	(56 MB_8_61,440)	—
ReduceScatter	(80 MB_8_2,064)	(128 MB_8_4,128), (1,710 MB_4_1)	(56 MB_8_93,184)	—
AllReduce	(632 MB_4_1)	—	—	(15-392 MB_32_6)

46, 50, 59, 60, 67], data compression [8, 9, 53, 56, 62, 63], sparsity exploration [10, 28], and topology-based algorithm design [12, 15, 16, 27, 29, 32, 35, 42, 42, 47, 57]. The vast majority of these approaches focus on improving communication performance based on the assumption that the underlying communication library is well-tuned and will effectively deliver their improvements. However, we have experimentally validated that the communication tuning matters a lot in Section 3 with significant bandwidth benefits.

2.2 Collective Communication

Distributed and parallel DNN training introduces a distinct class of communication patterns, known as *collective communication*, where data is aggregated or disseminated across multiple GPUs. The NVIDIA Collective Communication Library (NCCL) is designed to optimize multi-GPU and multi-node communication for NVIDIA GPUs and networking systems [5]. It offers high-performance primitives like *ReduceScatter*, *AllGather*, *AllReduce*, and so on. These APIs are optimized to provide high bandwidth and low latency communication over PCIe and NVLink interconnects within a single node, as well as over networks across multiple nodes. In addition, RCCL (ROCm Collective Communication Library) [1] is AMD's counterpart to NCCL. In this paper, we focus on NCCL since it is the most widely used and its design philosophy has influenced followers such as RCCL.

Table 1 summarizes the training parallelisms for the four deep neural network training tasks, including three large language models and a computer vision model, with typical message sizes and the collective communication primitives used. The detailed hardware and workload configurations can be found in Table 8. Of the four model training tasks, the VGG-19 model is the smallest, necessitating the use of only data parallelism. VGG-19 employs 32-way data parallelism, so every GPU needs to exchange its six bucketed gradients (15-392 MB in size) with any other GPU within the 32-GPU communication group via *AllReduce* in each iteration.

Unlike VGG-19, the other three large language models utilize a complex hybrid parallelism, combining 8-way tensor parallelism and 4-way data parallelism or 4-way pipeline parallelism. In such a configuration, for example, Yi-1.5-34B executes *AllGather* 61,440 times in a single training iteration, each time transferring 56 MB of activation data among eight GPUs within a single machine. It also executes *ReduceScatter* 93,184 times, each time passing 56MB of activation data between eight GPUs to each other within a single machine. Phi-2-2B and Llama-3.1-8B transfer more data each iteration, up to 1,710 MB, as it also requires transferring the model gradient along the data-parallel dimension between four GPU servers with *AllGather*, *ReduceScatter*, and *AllReduce*, respectively, due to their four-way data parallelism.

In summary, multiple types of collective communication primitives are frequently used in distributed DNN training, with message sizes ranging from tens of MB to a few GB, varying communication group sizes, and hardware setups. Therefore, it is crucial to efficiently execute communication tasks despite their complexity and variety.

Table 2: Primitive configuration parameters of NCCL

Parameter	Value Range
Algorithm (A)	Tree, Ring
Protocol (P)	LL, LL128, Simple
Transport (T)	peer-to-peer (P2P), shared memory (SHM)
Nchannel (NC)	$1 \le n \le 128, \quad n \in \mathbb{N}$
Nthread (NT)	$n = 32 \times i, \quad i \in \{1,2,3,\ldots,20\}$
Chunk size (C)	$n = 256 \times i, \quad i \in \{1,2,3,\ldots,8K\}$

2.3 Low-Level Primitive Configurations

We conduct a systematic and comprehensive study of NCCL's parameter space, analyzing all 158 parameters, including 93 undocumented ones. Our findings reveal that NCCL has 28 performance-sensitive parameters, categorized as follows: 1 for *algorithm* (*A*), 3 for *protocol* (*P*), 3 for *transport* (*T*), 11 for *nchannel* (*NC*), 3 for *nthread* (*NT*), and 7 for *chunk size* (*C*). Other parameters will be discussed in Section 8. When users call a communication primitive, NCCL runtime will create a corresponding collective communication task, and assign it a *configuration* in the form of $< A, P, T, NC, NT, C >$. As shown in Table 2, the parameter *A* determines how data is distributed, combined, or aggregated across GPUs and has several possible values like ring or tree. Specifically, when a *ring* algorithm is used for the *allreduce* primitive, data is transferred sequentially between nodes in a ring-like fash-

Table 3: Various configurations of *AllGather* (80MB) on the 8-GPU node with intra-node **PCIe** w.r.t. varied NCCL configurations. C_{P0} refers to the default NCCL configuration.

Config	A	P	T	NC	NT	C
$C_{P0}{}^{*}$	Ring	Simple	SHM	2	256	2 MB
C_{P1}	Ring	Simple	P2P	2	96	32 KB
C_{P2}	Ring	Simple	P2P	8	256	32 KB
C_{P3}	Ring	Simple	P2P	8	96	32 KB

ion for both reduction and broadcast phases. *P* defines how data is moved across GPU memory hierarchies and managed within the GPUs, such as loading data into the GPU cores (SMs) and performing necessary operations (like reductions). Among its options, *LL* stands for the Low-Latency protocol, optimized for small message sizes to minimize latency. *T* refers to the mechanism used for physically transferring data between GPUs, either within the same machine or across different machines. For instance, its *P2P* (Peer-to-Peer) option defines a direct connection between GPUs bypassing CPU, typically using NVLink or PCIe. *SHM* refers to enabling data transmission between certain GPUs via shared memory.

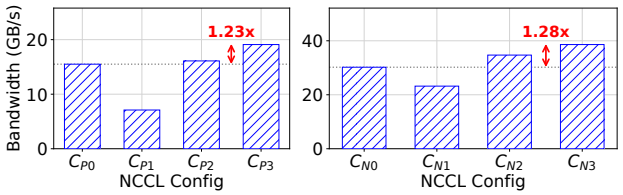
The remaining three parameters correspond to the allocation of resources or the degree of parallelism during communication. First, for a large message, NCCL will decompose data into *NC* partitions, with each partition bonded to a threadblock for independent transmission and processing. Each threadblock consists of a certain number of threads, defined by *NT*. In addition, each data partition can be further decomposed into several chunks, with the size defined by *C*. Therefore, *NC* threadblocks are transmitted chunks concurrently, and each threadblock processes its chunks sequentially.

3 NCCL Tuning Opportunities

However, we find that the default configuration of NCCL is often not optimal, and tuning the parameter values of the configuration can have a huge impact on communication performance during the DNN model training. Therefore, here, we will show the improvement of communication performance, as well as the positive impact during training performance through several examples.

3.1 Stand-alone Communication

Here, we consider communication tasks running alone without any interference. First, we take the $< AllGather, 80M, 8-A40-PCIe >$ communication task from the Phi-2-2B model in Table 1, i.e., *AllGather* aggregates 80MB data among 8 A40 GPUs in a single machine. Table 3 lists four primitive configurations that are used in the test. Figure 1 reports the communication bandwidth (higher is better). The best configuration C_{P3} achieves 2.69× the bandwidth of the worst configuration C_{P1}. The difference is that C_{P3} splits the message into more data partitions (larger *NC*), which are split into smaller chunks (smaller *C*), while adjusting the number of threads (*NT*) to align with the transmission granularity (*C*).



(a) PCIe (b) NVLink

Figure 1: Bandwidth comparison of using various NCCL configurations for <AllGather, 80M, 8-A40-*>

Table 4: Various configurations of *AllGather* (80MB) on the 8-GPU node with intra-node **NVLink** w.r.t. varied NCCL configurations. C_{N0} refers to the default NCCL configuration.

Config	A	P	T	NC	NT	C
$C_{N0}{}^{*}$	Ring	Simple	P2P	8	512	2 MB
C_{N1}	Ring	Simple	P2P	64	512	108 KB
C_{N2}	Ring	Simple	SHM	8	512	108 KB
C_{N3}	Ring	Simple	SHM	64	512	108 KB

Table 5: Various configurations of *AllReduce* on the 8-GPU node with intra-node **PCIe**. Each task is shown in the format x, y, where x indicates the message size and y represents the communication group size. *P* and *T* are omitted as they are set to Simple and SHM, respectively, across all configurations. C_0 refers to the default NCCL configuration.

Task	Config	A	NC	NT	C	Bwd (GB/s)
64 MB, 16	$C_0{}^{*}$	Ring	2	256	512KB	4.0
64 MB, 16	C_1	Tree	2	256	512KB	5.4
64 MB, 16	C_2	Tree	8	160	59 KB	8.9
15 MB, 8	C_3	Tree	8	160	59 KB	8.1
15 MB, 8	C_4	Ring	10	128	27 KB	8.8

All these parameters are set to utilize the parallel computing power of GPUs and the network transmission bandwidth. Compared to C_{P3}, C_{P1} uses a smaller concurrency count *NC*, leading to significant performance degradation. The performance of C_{P2} is lower than C_{P3}, even with an increase in the number of threads. It is worth noting that C_{P0}, the NCCL default configuration determined by NCCL's built-in cost model, is not optimal; its bandwidth is only 81.2% of that of C_{P3}.

Second, we run the aforementioned *AllGather* task in a new hardware environment with GPUs interconnected via NVLink instead of PCIe. As shown in Table 4, we test 4 configurations, namely, C_{N0} to C_{N3}. Figure 1 shows that the best configuration C_{N3} reaches 1.28× the bandwidth of the worst configuration C_{N0}. By comparing C_{N0} (the default configuration selected by NCCL) with C_{N3}, we observe that *T* plays a significant role, as C_{N3} employs share-memory (SHM) communication with better performance. By comparing C_{N3} with the second best performing C_{N2}, using more channels leads to better parallel performance, consistent with our observation in the PCIe experiment. Also, comparing C_{N1}, C_{N0} and C_{N3}, we can find that increasing *NC* plays the exact opposite role when setting the *T* parameter as P2P.

Table 6: Bandwidth of *AllGather* (80MB) on the 8-**NVLink**-GPU node w.r.t. varied computational interference levels

Interference level	NCCL Bwd	Tuned Bwd
Communication-only	30.08 GB/s	38.62 GB/s
Light computation	26.21 GB/s	35.14 GB/s
Heavy computation	18.26 GB/s	32.44 GB/s

Finally, we investigate the joint effect of different communication tasks, different hardware environments, and different configuration parameter choices. To this end, we take the *AllReduce-64MB* and *AllReduce-15MB* communication tasks from the VGG19 model, as listed in Table 1, running on a single machine with 8 A40 GPUs and two machines with 16 A40 GPUs, with PCIe for intra-node network and 100 Gbps InfiniBand for inter-node network. We test five configurations, with performance shown in Table 5. Comparing C_0 and C_1, we find that the algorithm parameter A becomes the key factor to performance as NCCL's default algorithm misjudging that the Ring algorithm better fits current hardware, but Tree is better in practice. Besides, if applying the best configuration C_2 for task $< AllReduce, 64MB, 16 >$ to the task $< AllReduce, 15MB, 8 >$ as C_3, it will be approximately 10% lower than the performance reached by C_4, indicating that configuration rediscovery is needed, even for very similar tasks. Comparing C_3 and C_4, we find that Ring algorithm is a better choice, and increasing the degree of parallelism (*NC*) can indeed enhance performance. In addition, we need to shrink both the minimum granularity of the transmission C and the number of threads *NT*.

According to all the experiments above, we observe that the default configurations determined by NCCL are often not optimal, leaving significant performance potential to NCCL parameter tuning. Our experiments in Section 6 prove that this problem also exists in multiple nodes. For various tasks on the same hardware or even the same task on different hardware, different optimal configurations are also needed. Also, it is not clear whether there is a certain pattern in the selection of parameters and the interactions between those parameters. Therefore, one needs to add an auto-tuning function so that the training task running above NCCL freely enjoys the optimized fast communication.

3.2 Concurrent Computational Interference

Communication and computation tasks without data dependencies are usually executed concurrently, and it is infrequent for communication to occupy resources exclusively. The competition between computation and communication for GPU resources, especially SM, cache, and global bandwidth, will cause the performance of each to degrade. Therefore, to better study the configuration tuning of NCCL communication tasks, especially in the presence of resource contention caused by computation tasks, we design the following NCCL tuning experiments with computational interference.

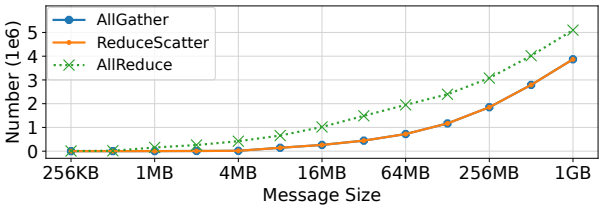


Figure 2: Combinations number of different communication

We introduce extra computational tasks over the *AllGather* communication task from Table 4, and take default configuration of NCCL as a baseline. The computational task we used is GEMM [4], which computes matrix multiplication A[m, k] $\times$ B[k, n] operation followed by a sigmoid [7] operation. Here, we choose different values of m, k, n in GEMM to represent the light and heavy computations. Table 6 shows that computational interference brings a significant communication performance degradation, even with light computation. For example, the *AllGather* bandwidth drops by 12.8% with interference by light computation and up to 39.3% by heavy computation. However, with carefully primitive configuration tuning, *AllGather*'s bandwidth has been improved by 34-78% for all settings. It's worth noting that, after configuration tuning, the cases with computational interference can be 7.8-16.8% faster than default NCCL without interference.

3.3 Challenges in NCCL Tuning

Tuning NCCL primitive configurations presents challenges that require addressing the following two questions.

How to quickly find the performant configuration in a large space? For specific communication types, message sizes, and communication group sizes, the configuration tuning space is huge. As shown in Figure 2, the number of possible combinations can reach up to millions. Traversing such a space and comparing the latencies and bandwidths of candidates is time-consuming. For example, testing all parameter combinations for *AllGather* with 80MB takes several hours on a single machine with 8 A40 GPUs connected by PCIe. With the increase of message size and communication group size, the time overhead expands significantly.

How to model computational interference w.r.t the training runtime dynamics? For parallel training tasks of a model, there are usually highly variable operators, dimensions of operator partitioning, and inputs to the operators, and consequently a diverse range of computational tasks are executed on the GPU. Modeling computational tasks and their impact on communication further increases the space of NCCL configuration. In addition, training frameworks add scheduling optimizations to the runtime, and introduce dynamics that can make the concurrent execution of computation and communication tasks unpredictable in advance. Exploring all potential combinations of these factors simultaneously results in an impractical search cost.

4 Communication Tuning Method

Next, we will design a unified, simple, and efficient configuration parameter tuning tool, `AutoCCL`, for collective communication primitives. We first consider communication modeling and tuning in an ideal state without computational interference. Recall that Table 2 presents six parameters that make up each configuration, and the selection of these parameter values forms a large search space. To address this challenge, we analyze NCCL's parameters and make several observations. First, we categorize the six parameters and apply different approaches to different types of parameters. We divide the parameters into two categories: those related to the implementation of primitives and those related to resource allocation, with a detailed analysis provided in Section 4.1.

Second, we find that resource allocation parameters are the primary cause of the large search space, and there are some patterns in the interactions between these parameters. By examining these interactions and their joint effects on performance, we propose a theoretical model to predict the performance of parameter combinations, thus avoiding exhaustive search in the vast parameter space (see Section 4.2). Finally, we integrate all these observations into a tuning algorithm, which is detailed in Section 4.3.

4.1 Parameter Division

First, algorithm, protocol, and transport are implementation-related and together determine the topology of the algorithm execution logic, the method of data transmission, and the software stack calls required to complete the primitive semantics. These parameters are crucial, but the search space is relatively small. For example, there are only two options for *algorithm*, three for *protocol*, and two for *transport*.

Some prior works have provided useful guidelines for selecting these implementation parameters [12,16,47]. However, in practice, we find that these guidelines are not always reliable. For instance, some works assume that bandwidth and latency between GPUs are fixed, allowing the time cost for Tree and Ring topology to be estimated as $log_2 N$ and $2 \times (N-1)$, respectively. However, in reality, bandwidth is affected by many factors, such as message size, cluster topology, congestion, and concurrency, which makes the cost model inaccurate. As a direct consequence, in the experiment corresponding to Table 5, NCCL incorrectly selected the algorithm.

Second, the parameters *NC*, *NT*, and *C* correspond to resource allocation. It is well-known that communication tasks during model training require both data transmission over the network and GPU cores for operations like accumulation and averaging. These three parameters jointly determine how to utilize the network bandwidth and GPU computational power. The search space for these parameters is relatively large, with options such as 8,192 for *Chunk Size* and 128 for *Nchannel*. We further observe that analyzing their impact on performance must be based on a prior selection of the above three implementation parameters. In other words, the choice of the first three parameters provides the foundation for modeling the analysis of the latter three. Although the first three parameters can be explored exhaustively, it is practically infeasible to search through the combinations of the latter three exhaustively. Therefore, we must build a performance model for these resource-related parameters.

Table 7: Notation table

Notation	Description
M	Message size
α_i	Initialization cost of phase i
$\overline{\beta i}$	Peak bandwidth of phase i
β_i	Bandwidth of phase i
$\gamma(NC,NT,C)$	Congestion of all phases

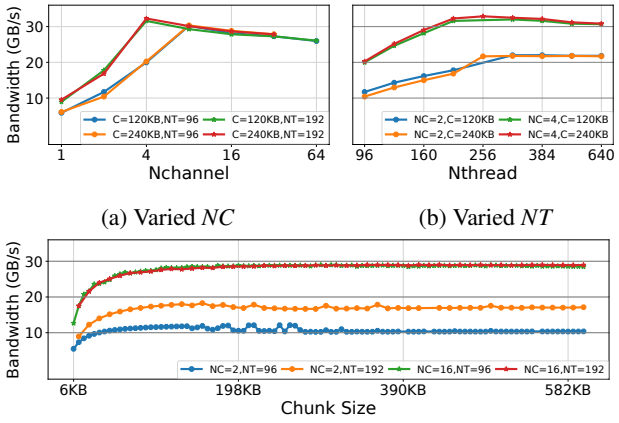
In summary, based on the classification of these parameters, we propose a subspace-based tuning method. First, we divide the entire search space into different subspaces $< A,P,T >$, each of which corresponds to a specific combination of *algorithm*, *protocol*, and *transport*. Then, within each subspace, we use a unified performance model for *NC*, *NT*, and *C* to determine the optimal combination of these parameters for that subspace. Finally, we compare the optimal combinations from all subspaces to identify the global optimal configuration. This approach mitigates the search space explosion for resource parameters while eliminating reliance on heuristic rules for the implementation parameters.

4.2 Modeling Resource Parameters

Given a subspace, we need to model three additional resource parameters. The notations required for modeling are provided in the table 7. Here, we evaluate the impact of resource parameter choices on communication bandwidth, denoted by $\beta(NC,NT,C)$. We provide qualitative analysis while avoiding difficult-to-achieve quantitative modeling. The execution of a collective primitive can be divided into two phases. In phase 0, the transport step is responsible for reading data from other GPUs and transferring it to the local buffer. Then, in phase 1, the protocol loads the data from the buffer to the SM, performs the reduction, and stores it back to the buffer. The bandwidths of these two phases are denoted as β_0 and β_1, respectively. Since the execution of the transport and protocol stages has data dependencies and is serial, as shown in Equation 1, $\beta(NC,NT,C)$ is equal to the minimum of the transport and protocol bandwidths.

$$\beta(NC,NT,C) = \min(\beta_0(NC,C), \beta_1(NC,NT)) \tag{1}$$

Next, we introduce the derivation process for the transport bandwidth formula in phase 0. The performance of transport depends on chunk size *C* and the number of concurrent messages *NC*, but not on *NT* since the transport does not include computation. We first compute the time t_0 required to transfer a message of size *M* across GPUs, as shown in Equation 2. $\frac{M}{NC \times C}$ represents how many serial steps the message will be



(a) Varied NC (b) Varied NT

(c) Varied C

Figure 3: The trend of <AllGather, 80M, 8> bandwidth on a node with 8 A40 GPUs connected via NVLink

divided into, where each step consists of NC parallel transmission tasks, each of size C. The term in parentheses on the right side of the equation represents the time cost of one step. This cost consists of the initial latency α_0 and the transmission time for a step of size $NC \times C$, where $\overline{\beta_0} \times \gamma$ represents the bandwidth under the current congestion. γ is the congestion coefficient, which increases with the increase of NC, NT, and C. We only analyze it qualitatively here. Equation 3 defines bandwidth as the message size divided by the time. By solving Equations 2 and 3, we obtain the bandwidth formula for phase 0 as Equation 4.

Similar to the bandwidth derivation for phase 0, we can also obtain the estimated bandwidth formula for phase 1, as shown in Equation 5. The protocol already determines the data granularity and cache pattern, so processing performance is related to NC and NT, not to C. Due to space limitations, we omit the derivation process. According to the model, when any two parameters among NC, NT, and C are fixed and the third parameter is gradually increased, β_i will monotonically increase, approach its physical bandwidth upper limit $\overline{\beta_i}$, and then stabilize, or even fall, thereby affecting the overall bandwidth $\beta(NC,NT,C)$.

$$t_0(NC,C) = \frac{M}{NC \times C} \times (\alpha_0 + \frac{NC \times C}{\overline{\beta_0} \times \gamma}) \quad (2)$$

$$\beta_0(NC,C) = \frac{M}{t_0(NC,C)} \quad (3)$$

$$\beta_0(NC,C) = \frac{NC \times C}{\alpha_0 + \frac{NC \times C}{\overline{\beta_0} \times \gamma}} \quad (4)$$

$$\beta_1(NC,NT) = \frac{NC \times NT}{\alpha_1 + \frac{NC \times NT}{\overline{\beta_1} \times \gamma}} \quad (5)$$

To validate the aforementioned model and its characteristics, we designed a set of communication primitive experiments. By using a controlled variable method, we varied the values of different resource parameters and observed

Algorithm 1: Subspace-Directed Tuning

Input: Task (w).
1 optimum ← nil
2 **for** subspace s ∈ $[A \times P \times T]$ **do**
3 config ← CoordinateDescentSearch(s)
4 **if** config.BwDelta(optimum) > 0 **then**
5 optimum ← config
6 **return** optimum

the changes in bandwidth of $< AllGather, 80M, 8-A40-NVLink >$. As expected, as shown in Figure 3, when any two parameters are fixed, the bandwidth increases first, then decreases or stabilizes as the third parameter increases. Furthermore, when the first two parameters change, the bandwidth peak (sweet point) corresponding to the third parameter also shifts accordingly. Figure 3a depicts that the optimal values of NC are 4 and 16 when $C = 80KB, NT = 96$ and $C = 20KB, NT = 96$, respectively, since a larger C results in a higher congestion factor γ, reducing the number of NC values required to reach the optimal point. Similarly, as shown in Figure 3b, the optimal values of NT are 320 and 196 when $NC = 2, C = 120KB$ and $NC = 4, C = 120KB$, respectively, since increasing either NC or NT increases γ, and with a larger NC, the NT curve reaches the sweet point earlier. Similar patterns are observed again in Figure 3c.

In summary, we find that the joint impact of NC, NT, and C on performance is not monotonic but exhibits the characteristics of a unimodal function with a sweet point. Based on this feature, we can naturally use the coordinate descent method [3] to search the resource parameter combination space. By abstracting NC, NT, and C as the three dimensions of the coordinate descent method, we can find the maximum value by continuously moving in the ascending direction along each dimension (see Section 4.3 for details).

4.3 Tuning Algorithm

Based on the parameter classification, resource-related parameter modeling, and subspace search method described above, we design an NCCL parameter tuning approach, which consists of Algorithm 1 and Algorithm 2.

Subspace-directed Tuning. The algorithm 1 uses a divide-and-conquer strategy to separately search each subspace for every communication task w. Specifically, we iterate through all the subspaces (lines 2-5). In line 3, we invoke the coordinate descent search (Algorithm 2) for subspace s to obtain the optimal configuration within that subspace. If the optimal configuration of subspace s outperforms the current best configuration, we update the best configuration (lines 4-5).

Coordinate descent search. For the input subspace s, Algorithm 2 sets M as the total number of resource parameters, representing the dimensions of tunable parameters within the subspace s (line 1). In our case, M is 3. Next, a configuration p is randomly generated, where the implementation parameters of p are determined by subspace s, and the values of the

Algorithm 2: Coordinate Descent Search

Input: Subspace (s).

```
1  M ← Number of resource parameters.
2  Randomly generate a config p from subspace s.
3  optimum ← p
4  dim, tuned_dim, lr ← 0, 0, 0.01
5  while tuned_dim ≤ M do
6      p.ProfileBw()
7      if p.BwDelta(optimum) > 0 then
8          lr ← p.BwDelta(optimum) / p.Bw()
9          optimum, tuned_dim ← p, 0
10     else
11         tuned_dim ← tuned_dim + 1
12         dim, lr ← dim + 1, 0.01
13     p ← optimum
14     p[dim] ← p[dim] + lr
15 return optimum
```

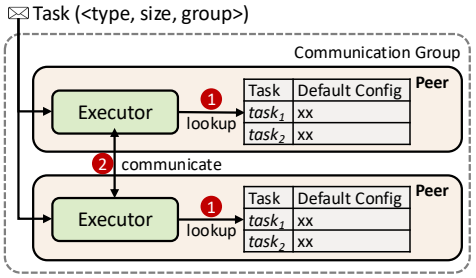
other resource parameters are randomly selected from their respective ranges (line 2). Line 4 initializes the index of the dimension to be tuned, the number of tuned dimensions, and the learning rate. Lines 5-14 execute a gradient descent search loop until all dimensions have been tuned (i.e., the number of tuned dimensions equals the total number of tunable dimensions). In line 6, configuration p is executed to profile its bandwidth. In line 7, the bandwidth of p is compared to the current *optimum*. Following this, two scenarios must be considered:

If p achieves higher bandwidth, then the configuration should be updated along the current tuning dimension. Line 8 calculates the percentage improvement in the bandwidth of p compared to the current *optimum*, using this percentage as the learning rate lr. Line 9 updates *optimum* to p and resets the tuned dimension counter, *tuned_dim*. If p does not improve bandwidth, then a new tuning dimension should be selected. Line 11 increments *tuned_dim*, and line 12 selects the next dimension to tune and resets the learning rate lr.

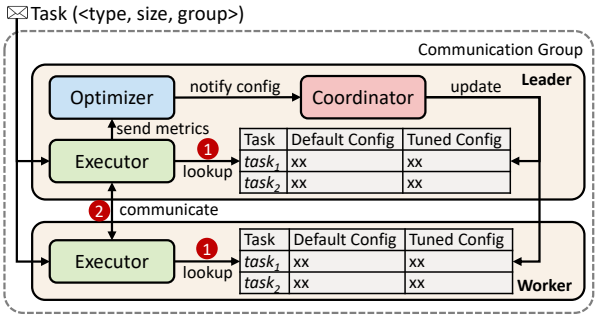
After analyzing p, lines 13 and 14 update the tuning dimension *dim* in the optimal configuration based on lr. If all dimensions have been tuned, line 15 returns the optimal configuration found within the subspace s.

5 AutoCCL: Design and Implementation

Next, we present the design and implementation details of the automated tuner, AutoCCL, focusing on how AutoCCL gathers the performance profiling data required for tuning from the training tasks and selects new configurations. To avoid the complexity of modeling computational interference and dynamic scheduling, we propose an online tuning approach. This approach leverages repetitive patterns of computation and communication cycles during the training process, using the tuning algorithm introduced in Section 4.3 for the same communication task until tuning is completed. It enables the collection of communication performance data affected by interference during the training process. This data is then incorporated into the tuning algorithm to capture accurately the impact of co-running computational tasks on communication. It could also amortise the cost of tuning over the early iterations.



(a) NCCL



(b) AutoCCL

Figure 4: The architecture of NCCL and AutoCCL

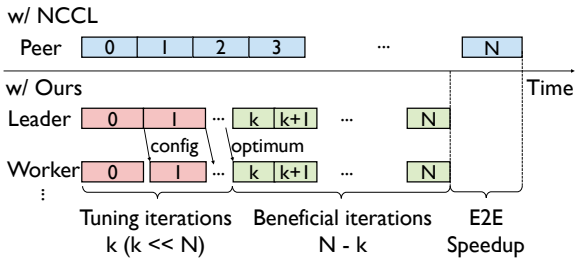


Figure 5: Online tuning workflow with AutoCCL (bottom). Rectangles with numbers represent training iterations.

5.1 Overall Architecture

Figure 4 illustrates the architecture of AutoCCL and its comparison with NCCL. Although AutoCCL evolved from NCCL, it exhibits significant differences from NCCL in several aspects. First, AutoCCL does not follow NCCL's peer-to-peer design. In NCCL, within each communication group executing a collective communication task, all GPUs are identical *Peers*, and each *Peer* independently generates a default configuration for the same communication task based on a deterministic cost model. Unlike this, in AutoCCL, one GPU is designated as the *Leader*, responsible for running the tuning algorithm, identifying the performant configuration, and updating the results to

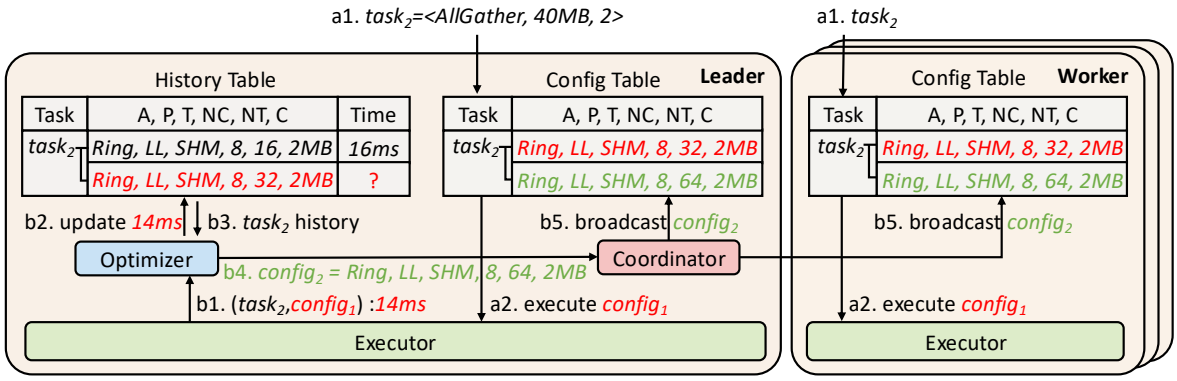


Figure 6: The online tuning workflow of AutoCCL. Colors distinguish different configurations.

the other GPUs, referred as *Workers*. Each AutoCCL *Worker* acts similarly as a *Peer* in NCCL, maintaining a configuration table. For any task, this table stores a default configuration (generated by NCCL's cost model) as well as a tuned configuration generated by the *Leader*. When a communication task is received, the *Worker*'s *Executor* checks the configuration table and selects the tuned configuration if available; otherwise, it uses the default configuration.

In contrast to the simple design of *Workers*, the logic of the *Leader* is more complex. The *Leader* introduces two additional system components: the *Optimizer* and the *Coordinator*. *Optimizer* collects performance profiling data from the *Executor*, determines whether tuning is necessary, and initiates the tuning process. Without requiring coordination with other nodes, it independently searches the performant configuration and decides whether to replace the current one. Notified by *Optimizer*, *Coordinator* then broadcasts the updated configuration to all nodes in the communication group, updating their tuned configuration. Upon success, all subsequent identical communication tasks will use the updated configuration.

5.2 Iterative Online Tuning

As shown in Figure 5, by placing the communication tuning process in the iterations of the pre-training period, we could cope with the diversity of workloads and hide tuning time cost.

We adapt the tuning algorithm for each subspace to the same communication task across the iterations. As shown in Table 1, due to the iterative nature of training and the use of microbatches, the same communication task is executed many times in each iteration. When performing a communication task, we run one step of the coordinate descent algorithm for the corresponding subspace. Upon completion, if the subspace has not reached its optimum, we save the corresponding state to facilitate the next search. Otherwise, we stop and switch to the next subspace. When the performant solution is found, the *Coordinator* broadcasts it to each node within the communication group, which guarantees the atomicity of the update and avoids the inconsistent state of the same communication task using different versions of the configuration.

Note that tuning introduces a slight overhead, which may extend the duration of each iteration. However, since the tuning process is highly efficient and the tuned iteration significantly improves performance, the end-to-end training acceleration remains substantial.

5.3 New Workflow

Here, we provide a walking-through example of the aforementioned online tuning method, illustrating the new workflow of AutoCCL, as shown in Figure 6. On the *Leader*, an additional *History Table* records the execution times of various tasks under different configurations, which the *Optimizer* uses to generate new configurations for tuning. Initially, the *Config Table* contains only the red *config_1* (<Ring, LL, SHM, 8, 32, 2MB>) for *task_2*, without green *config_2*, which will be generated during the tuning process of this example. *a*1 and *a*2 show the process of fetching and executing the current optimal configuration for the communication task, while *b*1 to *b*5 illustrate the online tuning and configuration update process.

Next, supposing both the *Leader* and *Worker* receive a new communication task with the same communication primitive, message size, and communication group members as the previously recorded *task_2*, the task is an *AllGather* operation with a message size of 40MB and a communication group size of 2. Both the *Leader* and *Worker* first need to query the *Config Table* for the target configuration (step *a*1). Since the *Config Table* is fully replicated between the *Leader* and *Worker*, both will retrieve the red *config_1* and submit *task_2* to the runtime for execution using *config_1* (step *a*2).

After a certain period, *task_2* completes execution, and the *Executor* on the *Leader* returns the execution time (14ms) to the *Optimizer* (step *b*1). The *Optimizer* updates the execution time in the *History Table* (step *b*2). The *Optimizer* then retrieves the historical execution times for *task_2* and initiates a new round of tuning (step *b*3). The *Optimizer* observes that increasing *NT* from 16 to 32 reduces the communication time from 16 ms to 14 ms. According to the coordinate

Table 8: DNN model statistics

Model	MBS	GBS	TP	PP	DP
Phi-2-2B	8	512	8	1	1-4
Llama-3.1-8B	2	256	8	1	1-4
Yi-1.5-34B	1	1,024	8	4	1
VGG-19-0.14B	32	32×[8,16,32]	1	1	8-32

descent method, the next attempt will also increase NT. Therefore, it selects $NT = 64$, generating *config_2*, and notifies the co-existed *Coordinator* (step *b*4). The *Coordinator* uses a broadcast operation with synchronous semantics to update *config_2* in both the *Leader* and *Worker*'s *Config Table*, replacing the original *config_1* (step *b*5). Subsequent *task_2* executions will use *config_2*, and the online tuning process will continue until the optimal configuration is found.

5.4 Implementation Details

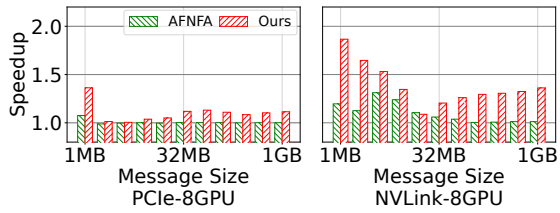
AutoCCL is implemented based on NCCL 2.18.3, with 9,176 lines of C++ code, supporting collective communication primitives such as *AllGather*, *Allreduce*, and *ReduceScatter*. We modified NCCL's configuration generation module and transport initialization module to allow more flexible use of various configurations. For any communication group, AutoCCL automatically launches an additional thread as the *Leader* on one node to perform communication task tuning. Due to the asynchronous execution characteristics of CUDA kernels, we start an additional thread to measure execution time.

AutoCCL is implemented based on the C++ standard library, the *Coordinator* is implemented based on Linux sockets, and no additional libraries are required. Thereby, users can seamlessly migrate from NCCL to AutoCCL by preloading the dynamic library without requiring any code modification. Since AutoCCL and NCCL share identical interfaces, training frameworks such as PyTorch [41] and MegatronLM [51] can directly integrate AutoCCL without modification, and training tasks running on these frameworks are unaware of the underlying switch. Our code is open-sourced at [2].

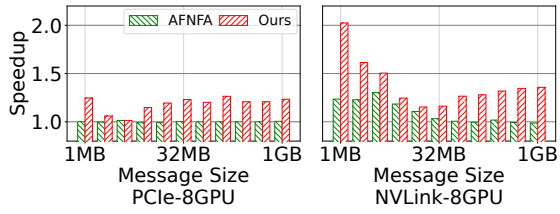
6 Evaluation

6.1 Experimental Setup

Cluster setup. We use two clusters to demonstrate the generality of our tuner. Cluster A has 2 nodes, interconnected via 2 pairs of 400 Gbps InfiniBand-PCIe 5.0 NIC. Each node has 8 NVIDIA Ampere A40 GPUs (48 GB GDDR6, PCIe 4.0), an AMD EPYC 9654 CPU, and 64GB of host memory. The 8 GPUs within a single node are divided into 4 pairs, with each pair of GPUs connected via 400 Gbps NVLink interconnects. Cluster B contains 4 machines in a single rack, connected via a 100 Gbps InfiniBand network. Each machine has 8 NVIDIA Ampere A40 GPUs (the intra-node GPUs are connected via PCIe 4.0), an Intel Xeon Gold 5320 CPU, and 504 GB of host memory. In Cluster B, due to hardware limitations, peer-to-peer communication is not available between some GPUs. Both clusters operate under the same environment with CUDA



(a) *AllGather*



(b) *ReduceScatter*

Figure 7: Bandwidth speedups for the *AllGather* and *ReduceScatter* communication of different A40 clusters

v12.1 and NVIDIA driver v470.63.01.

DNN training jobs. We evaluate four models from different application scenarios, including Phi-2-2B [21], Llama-3.1-8B [54], Yi-1.5-34B [65], and VGG-19 [52] shown in Table 8, where three are large language models with up to 32 billion parameters and one computer vision model. Model training is conducted using PyTorch version 2.1.0. Various parallelisms are managed by MegatronLM. Due to limited GPU memory, Llama-3.1-8B combines *distributed optimizer* [45] along with data parallelism. We set the micro-batch size as large as possible under the GPU memory limitation for better hardware utilization. The global batch size follows the setting of the GPT series with a similar model scale [11].

Baseline systems. We compare against NCCL v2.18.3-1, almost the latest version provided by NVIDIA, and internally performing tuning to some extent. We always enable its tuning function. We also include an academic collective communication tuning system, AFNFA [64]. We use the random forest algorithm to train 1% of the offline sampled data and use environment variables to set the configuration for different communication to reproduce the results of AFNFA. All data represent the average results from multiple experiments.

6.2 Communication Micro-Benchmarks

We explore how AutoCCL improves different collective communication primitives, considering various hardware setups and message transmission sizes. We first evaluate the communication performance without computational interference, followed by experiments with interference.

Communication without computational interference. We first consider *AllGather* and *ReduceScatter* primitives, which are heavily used in *Tensor Parallelism*. We test them with 8 GPUs within a single machine, connected via PCIe 4.0 or NVLink. Figure 7 summarizes the bandwidth speedups of AutoCCL and AFNFA over the native NCCL.

For A40-PCIe, AFNFA's results are almost identical

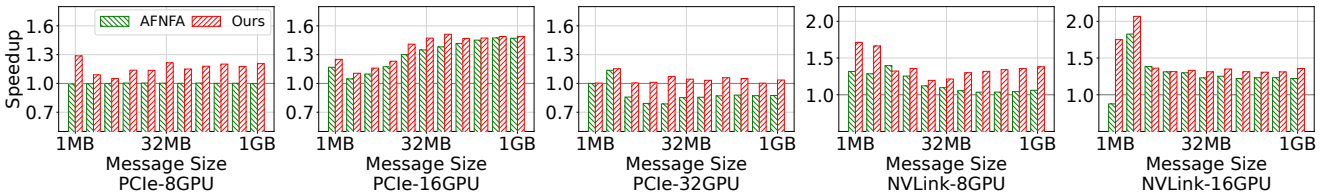


Figure 8: Bandwidth speedups for the *AllReduce* communication of A40 clusters with different setups

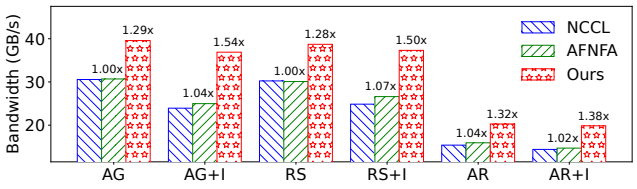


Figure 9: Comparison of bandwidth with and without computation **interference** for different communication. The message size is 128 MB. '+I' refers to the communication that interfered with computation. The speedup numbers are annotated in the figure. *AG*, *RS* and *AR* stand for *AllGather*, *ReduceScatter* and *AllReduce* respectively.

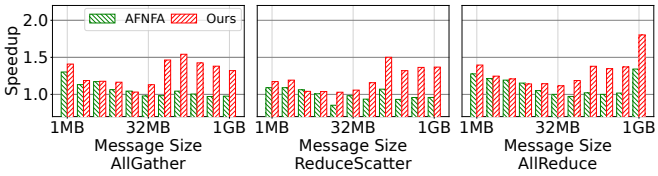


Figure 10: Bandwidth speedups with increasing message sizes and computation **interference** for different communication types of a cluster with 8 A40 connected with NVLink

to NCCL, offering little optimization. However, AutoCCL achieved an average bandwidth improvement of 22.66% and 27.52% for *AllGather* and *ReduceScatter*, respectively, compared to NCCL. This is especially evident for larger message sizes, where AutoCCL outperforms both baselines.

It is widely acknowledged that NVLink is highly efficient, and NCCL has been heavily optimized for it. Despite this, AutoCCL demonstrates even greater advantages on NVLink compared to PCIe. For instance, for various message sizes, the average bandwidth speedups are 1.38× and 1.39×, respectively. In contrast, AFNFA shows far fewer benefits, particularly for large message sizes, where its tuning results still align with those of NCCL since it uses global configuration and cannot adapt to different message sizes.

We observe that in a few cases, AutoCCL's tuning results align with those of NCCL. This occurs because, for these particular message sizes, NCCL's configuration is already optimal. In other words, its standard α-β network model can accurately predict performance at these points, although this model does not generalize to all cases.

Next, we analyze *AllReduce*, which is commonly used in data parallelism to exchange gradient information. As shown in Figure 8, similar to the results above, AutoCCL achieved average bandwidth speedups of 1.28× and 1.15× compared to NCCL and AFNFA, respectively. In the PCIe-32GPU cluster, where AFNFA even results in negative optimization, AutoCCL slightly outperforms NCCL since the default configuration of NCCL is effective. However, NCCL becomes less suitable under computational interference.

Communication with computational interference. We test the behavior of AutoCCL when communication and computation are co-executed. We evaluate the above three primitives with a GEMM operation running concurrently alongside each primitive. The matrices of GEMM are A[m, k] and B[k, n], where m, k, and n are 3,456, 128, and 3,456. Figure 9 compares the bandwidth of these primitives with and without computational interference for a fixed message size.

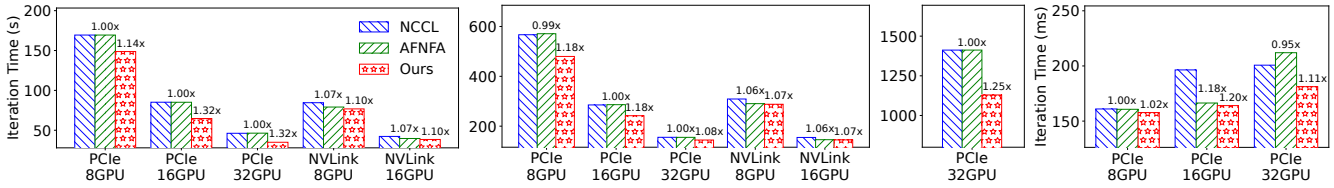
Similar to previous studies, the bandwidth of the three primitives decreases significantly under computational interference. With AutoCCL tuning, the bandwidth speedups compared to NCCL are 1.29×, 1.50×, and 1.38×, respectively. In contrast, AFNFA demonstrates poor performance, showing no improvements for *AllGather* and *ReduceScatter* and merely a 1.02× improvement for *AllReduce*. Interestingly, after AutoCCL tuning, the optimal bandwidth under interference closely approximates the optimal bandwidth without interference. This indicates that AutoCCL can accurately predict performance even with interference.

Figure 10 differs from Figure 9 by increasing the message size for primitives while simultaneously scaling up the corresponding co-executed GEMM to simulate varying degrees of resource sharing and interference.

For *AllGather* and *AllReduce*, AFNFA only shows tuning benefits for message sizes below 16 MB, compared to NCCL. In contrast, AutoCCL not only matches or slightly outperforms AFNFA for small message sizes, but also achieves significant bandwidth improvements of 1.11-1.76× and 1.16-1.39× for larger message sizes (e.g., those exceeding 32 MB) compared to NCCL, respectively. For *ReduceScatter*, AFNFA either performs on par with NCCL or results in negative optimization. This indicates that in more competitive scenarios, both NCCL and AFNFA fail to deliver satisfactory performance, as their modeling approaches are insufficient to handle complex environments and dynamic changes. In contrast, AutoCCL shows a distinct performance advantage in these situations.

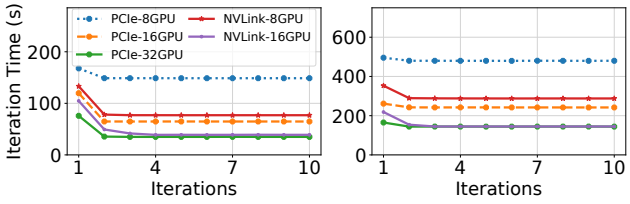
6.3 End-to-End Performance

To show how AutoCCL accelerates the training progress for real-world models, we conduct experiments to train models



(a) Phi-2-2B (b) Llama-3.1-8B (c) Yi-1.5-34B (d) VGG-19-0.14B

Figure 11: Training iteration time for different models between NCCL, AFNFA, and AutoCCL, with speedup annotated



(a) Phi-2-2B (b) Llama-3.1-8B

Figure 12: Rapid convergence of AutoCCL tuning at early iterations of end-to-end LLM training

on different communication systems under various hardware setups. Figure 11 shows the training iteration time of different models. Among all the hardware setups, AutoCCL outperforms both NCCL and AFNFA in training these models, while AFNFA is slower than NCCL in some cases as its offline-tuned configurations degrade performance under computational interference. On the PCIe machine, we observed a greater performance improvement compared to microbenchmark results because our online tuner exhibits stronger resilience to such interference than NCCL and AFNFA, whose performances are significantly degraded. The end-to-end gains on NVLink machines are modest because in scenarios where communication overlaps with computation and computation is dominant, excessive optimization of communication can slow down computation and reduce overall performance. For best cases, AutoCCL can improve the training iteration time by more than 32% compared to both NCCL and AFNFA.

6.4 Efficiency of Iterative Online Tuning

To show the efficiency of the AutoCCL to find the performant configuration with iterative online tuning, we measure how the iteration time changes when training the models, as shown in Figure 12. As noted in Table 1, because of the repetition of layers in the model and the large number of repeated micro batch sizes in each global batch size, the Transformer models like Llama-3.1-8B, Phi-2-2B, and Yi-1.5-34B can generate thousands to tens of thousands of repetitive communication operations per iteration. For these models, only several iterations are required for AutoCCL to find the performant configurations. For smaller models like VGG-19, although less repetitive communication operations are involved in each iteration, the time consumed per iteration is relatively short, e.g., 150-200 ms per iteration for VGG-19 in Figure 11d, which is around 1,000 times shorter than that of Llama-3.1-8B. In our evaluation, it only takes no more than 10 minutes for AutoCCL to find the performant configuration in VGG-19. In summary, for these four models of different areas, AutoCCL can rapidly identify a sufficiently performant configuration within just a few iterations or minutes through online tuning, highlighting its ability to adapt quickly to new environments, which is infeasible for offline tuning strategies.

7 Related Work

Computation-communication scheduling. At the application level, many studies [13, 19, 22, 23, 31, 46, 50, 59, 60, 67] aim to reduce communication overhead by overlapping computation and communication, thereby enhancing training efficiency. For example, Horovod [46] overlaps all-reduce communication with backward computation to improve performance. AutoCCL complements these methods by additionally considering the interference between computation and communication during the tuning process, making it more effective for tasks with specific scheduling requirements.

Communication compression. Other approaches [8, 9, 53, 56, 62, 63] improve communication efficiency by exploiting data sparsity. HiPress [9] introduces a framework to efficiently compress gradient data in *AllReduce*, reducing latency. ZeRO++ [56] combines quantization with collectives of *AllReduce* and *AllGather* in the ZeRO optimization to improve efficiency. AutoCCL is complementary to them, as it can be used to further enhance the performance of underlying communication collectives for these compression algorithms.

Collective algorithm generation. At the algorithm level, several studies focus on developing topology-aware [12, 15, 16, 27, 29, 32, 35, 42, 42, 47, 57] or sparse-aware [10, 28] communication algorithms. For instance, Swing [16] proposes a new *AllReduce* algorithm to minimize hops in torus networks, while MCCLang [15] offers a domain-specific programming interface and efficient compiler for easier algorithm customization. Unlike these works, AutoCCL does not introduce new algorithms but treats the algorithm choice as a tunable parameter. In the future, AutoCCL can be integrated with these approaches to further enhance runtime performance.

Network tuning. At the hardware level, to achieve higher network communication bandwidth, NCCL [5] uses the Alpha-Beta model to predict communication efficiency based on hardware topology and bandwidth. AFNFA [64] improves this prediction using machine learning with offline profiling.

However, all these approaches share a global configuration (i.e., through NCCL environment variables [6]) and do not consider computational interference in real-world workloads. In contrast, AutoCCL employs online profiling to capture the interference effects and dynamically selects a performant configuration for each collective communication operation, improving performance across diverse workloads.

8 Discussion

8.1 Parameter Selection and Extensibility

Once the implementation-wise parameter T is determined, NCCL allows further optimization of transport-specific parameters, including *Ethernet*, *IB*, *COLL_SHARP*, *NVLS*, *SHM*, and *P2P*. AFNFA considers *NCCL_SOCKET_NTHREADS* for Ethernet transport and *NCCL_NET_GDR_LEVEL* for IB transport. However, our study does not include these parameters, as they introduce a risk of failures while providing limited performance gains (Section 8.2). AFNFA attempts to optimize NCCL including *NCCL_ALGO* for *A*, *NCCL_SHM_DISABLE* and *NCCL_P2P_LEVEL* for *T*, NCCL_MAX/MIN_NCHANNEL for *NC*, and *NCCL_BUFFSIZE* for *C*. These tunable combinations are incorporated into our search space. Our search space also includes P and NT, whose importance is demonstrated in Section 3.1.

The subspace coordinate descent method is inherently scalable, allowing users to optimize transport-specific parameters. Introducing transport-specific parameters only increases the number of implementation-wise parameter combinations, effectively subdividing the existing subspaces into finer-grained subspaces. The scalability of the subspace coordinate descent method enables users to further optimize transport-specific parameters as needed.

Our findings suggest that using default or fixed values for transport-specific parameters already yields robust performance. For example, we attempted to optimize IB-specific parameters and found that the default values of *NCCL_IB_SPLIT_DATA_ON_QPS* and *NCCL_IB_QPS_PER_CONNECTION* are already optimal. Similarly, the optimal value for *NCCL_NET_GDR_LEVEL* is typically determined based on the NIC and GPU topology, making it effective across different communication workloads. Since the optimal values for transport-specific parameters are often hardware-dependent, further exploration of their optimization is left for future work.

8.2 Task Failures and Valid Configurations

Tuning NCCL parameters can potentially lead to training task failures. This issue has been reported in related studies [40, 64] and confirmed by our experiments. As previously mentioned, once an implement-wise parameter such as T is determined, collective communication will employ a specific transport mechanism, such as P2P transport. NCCL allows further optimizations for specific transport mechanisms—for instance, enabling *P2P_USE_CUDA_MEMCPY* for P2P transport can improve bandwidth. However, in end-to-end training tasks, this setting may result in deadlocks. Additionally, we observed another type of failure: for resource-allocation parameters such as *NC*, *NT*, and *C*, excessively large values can lead to resource saturation, ultimately causing program crashes.

Nevertheless, failures induced by tuning have minimal impact on our system for two reasons. First, transport-specific optimizations are beyond our scope; therefore, we inherently avoid failures associated with them. In the future, we may explore further optimizations under failure recovery support. Second, for *NC*, *NT*, and *C*, due to the unimodal nature of the performance function, excessively large values do not yield performance gains. Therefore, we can impose upper bounds on resource usage without sacrificing performance, effectively preventing failures.

Besides, if a failure occurs, online tuning introduces additional overhead for loading checkpoints compared to offline tuning. However, given the numerous recent optimizations in checkpointing [18, 26, 61], this overhead has been significantly reduced and is negligible. The dominant cost of failure recovery remains container rebooting, which is required for both offline and online tuning.

Finally, failure handling and recovery are orthogonal to tuning. We can explore this topic in future work.

9 Conclusion

AutoCCL is an automated tuning tool that optimizes low-level NCCL parameter selection. By decoupling implementation-related parameters from those that inflate the search space, our divide-and-conquer approach reduces the necessity for extensive trial runs. Furthermore, its online tuning strategy efficiently handles communication-computation interference. Evaluation across multiple clusters and models demonstrates that AutoCCL significantly enhances both communication and end-to-end training performance compared to NCCL and another SOTA tuner.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rui Miao, for their insightful comments. This work is supported in part by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No.: XDB0660101, XDB0660000, and XDB0660100, the National Key R&D Program of China under Grant No. 2024YFB4505701, and the University Synergy Innovation Program of Anhui Province under Grant No. 2024YFB4505701. We thank the computing resources and technical support from Institute of Artificial Intelligence, Hefei Comprehensive National Science Center. We thank the computing resources and technical support from China Telecom Co., Ltd. Hefei Branch. Cheng Li is the corresponding author.

References

[1] AMD. ROCm communication collectives library (RCCL). https://github.com/ROCm/rccl, 2024. [Accessed 19-09-2024].

[2] Automated tunable collective communication library (AutoCCL). https://github.com/gbxu/autoccl, 2024. [Accessed 19-09-2024].

[3] Coordinate descent. https://en.wikipedia.org/wiki/Coordinate_descent, 2024. [Accessed 19-09-2024].

[4] General Matrix Multiplication. https://en.wikipedia.org/wiki/Matrix_multiplication, 2024. [Accessed 19-09-2024].

[5] NVIDIA Collective Communication Library (NCCL). https://github.com/nvidia/nccl, 2024. [Accessed 19-09-2024].

[6] NVIDIA Collective Communication Library (NCCL) Documentation. https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2183/user-guide/docs/env.html, 2024. [Accessed 19-09-2024].

[7] Sigmoid Function. https://en.wikipedia.org/wiki/Sigmoid_function, 2024. [Accessed 19-09-2024].

[8] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

[9] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 359–375, 2021.

[10] Charles Block, Gerasimos Gerogiannis, Charith Mendis, Ariful Azad, and Josep Torrellas. Two-face: Combining collective and one-sided communication for efficient distributed spmm. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1200–1217, 2024.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[12] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 62–75, 2021.

[13] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 178–191, 2024.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[15] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 502–514, 2023.

[16] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. Swing: Short-cutting rings for higher bandwidth allreduce. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1445–1462, 2024.

[17] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[18] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1110–1125, 2024.

[19] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems*, 1:418–430, 2019.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[21] Alyssa Hughes. Phi-2: The surprising power of small language models — microsoft.com. https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/. [Accessed 19-09-2024].

[22] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 402–416, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *Proceedings of Machine Learning and Systems*, 1:132–145, 2019.

[24] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Yong Li, Zhen Zheng, Xiaoyong Liu, Wei Lin, et al. Whale: Scaling deep learning model training to the trillions. *arXiv preprint arXiv:2011.09208*, 2020.

[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[26] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. Megascale: Scaling large language model training to more than 10,000 gpus. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[27] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. Tccl: Discovering better communication paths for pcie gpu clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 999–1015, 2024.

[28] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[29] Sabuj Laskar, Pranati Majhi, Sungkeun Kim, Farabi Mahmud, Abdullah Muzahid, and Eun Jung Kim. Enhancing collective communication in mcm accelerators for deep learning training. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–16. IEEE, 2024.

[30] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[31] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed moe training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 945–959, 2023.

[32] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[33] Shengwei Li, Kai Lu, Zhiquan Lai, Weijie Liu, Keshi Ge, and Dongsheng Li. A multidimensional communication scheduling method for hybrid parallel dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[34] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. nnscaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, 2024.

[35] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems*, 2:82–97, 2020.

[36] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954, 2020.

[37] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[38] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[39] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[40] Lichen Pan, Juncheng Liu, Jinhui Yuan, Rongkai Zhang, Pengze Li, and Zhen Xiao. Occl: a deadlock-free library for gpu collective communication. *arXiv preprint arXiv:2303.06324*, 2023.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[42] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[43] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[44] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

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

[47] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. Taccl: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, 2023.

[48] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[49] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[50] Shaohuai Shi, Xinglin Pan, Qiang Wang, Chengjian Liu, Xiaozhe Ren, Zhongzhe Hu, Yu Yang, Bo Li, and Xiaowen Chu. Schemoe: An extensible mixture-of-experts distributed training system with tasks scheduling. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 236–249, 2024.

[51] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[52] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[53] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. Optimus-cc: Efficient large nlp model training with 3d parallelism aware communication compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 560–573, 2023.

[54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[55] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating dnn training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[56] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, et al. Zero++: Extremely efficient collective communication for large model training. In *The Twelfth International Conference on Learning Representations*, 2023.

[57] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems*, 2:172–186, 2020.

[58] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[59] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 93–106, 2022.

[60] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. Topoopt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, 2023.

[61] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[62] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Ng. Espresso: Revisiting gradient compression from the system perspective. *arXiv preprint arXiv:2205.14465*, 2022.

[63] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Eugene Ng. Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 867–882, 2023.

[64] Zibo Wang, Yuhang Zhou, Chen Tian, Xiaoliang Wang, and Xianping Chen. Afnfa: An approach to automate nccl configuration exploration. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, pages 204–205, 2023.

[65] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, et al. Yi: Open foundation models by 01.ai, 2024.

[66] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[67] Yonghao Zhuang, Lianmin Zheng, Zhuohan Li, Eric Xing, Qirong Ho, Joseph Gonzalez, Ion Stoica, Hao Zhang, and Hexu Zhao. On optimizing the communication of model parallelism. *Proceedings of Machine Learning and Systems*, 5, 2023.

OPTIREDUCE: Resilient and Tail-Optimal AllReduce for Distributed Deep Learning in the Cloud

Ertza Warraich, Omer Shabtai[†], Khalid Manaa[†], Shay Vargaftik[‡],
Yonatan Piasetzky[†], Matty Kadosh[†], Lalith Suresh[§], Muhammad Shahbaz[*]
Purdue University [†]*Nvidia* [‡]*VMware Research* [§]*Feldera* [*]*University of Michigan*

Abstract

We present OPTIREDUCE, a new collective-communication system for the cloud with bounded, predictable completion times for deep-learning jobs in the presence of varying computation (stragglers) and communication (congestion and gradient drops) variabilities. OPTIREDUCE exploits the inherent resiliency and the stochastic nature of distributed deep-learning (DDL) training and fine-tuning to work with approximated (or lost) gradients—providing an efficient balance between tail performance and the resulting accuracy of the trained models.

Exploiting this domain-specific characteristic of DDL, OPTIREDUCE introduces (1) mechanisms (e.g., unreliable bounded transport with adaptive timeout) to improve the DDL jobs' tail execution time, and (2) strategies (e.g., Transpose AllReduce and Hadamard Transform) to mitigate the impact of dropped gradient entries on model accuracy. Our evaluation shows that OPTIREDUCE achieves 70% and 30% faster time-to-accuracy (TTA), on average, when operating in shared, cloud environments (e.g., CloudLab) compared to Gloo and NCCL, respectively.

1 Introduction

Synchronous distributed data-parallel training [175] is now the de-facto standard for training and fine-tuning large-scale deep-learning models (comprising billions or even trillions of parameters) and datasets (comprising terabytes of data) that form the backbone of many mainstream enterprise applications, including computer vision [64, 74, 96, 161], natural-language processing and large-language models [53, 63, 110, 167], recommendation and prediction systems [70, 71, 84, 86, 130], and healthcare [93, 112, 125, 169]. Under this scheme, the training occurs in rounds (or epochs). Workers locally train a copy of the model on a fragment of data and then share the model updates (i.e., gradients) among themselves over the network to compute an aggregated result. The aggregate is then used to update the model locally for the next round of training. Distributed deep-learning (DDL) is, therefore, inherently a computation- and communication-intensive workload and is becoming even more so with the growing model sizes (e.g., Bart [101], GPT-2/3 [53, 126], LLaMA [65, 146]), and datasets [38, 135, 143].

To train and fine-tune such large models, extensive efforts are underway in reducing both the computation and communication time of DDL jobs, albeit in isolation. On the one hand, we have GPUs [141] and emerging hardware accelerators, like Tensor Processing Units (TPUs) [90], that are drastically bringing down the computation time—reducing it by 62× over the last decade [136]. While, on the other hand, we have recent proposals based on programmable switches [157] that aim at reducing the communication time by 2–5× (via in-network aggregation) [136]. Yet, when seen together, both these efforts mainly help in improving the average completion time of a deep-learning job (either by accelerating computation or communication). The vast array of system-level variabilities (e.g., device failures, OS and hypervisor scheduling, and resource contention) and network-level delays (e.g., congestion and retransmissions) still lead to long tails; hence, resulting in poor overall performance for these training jobs—with tail reaching as high as 4× the mean latency in shared environments (e.g., public cloud providers)[1] [72, 77, 99, 106, 118, 153, 166].

In this paper, we make the case for OPTIREDUCE, a collective-communication system for the cloud tenants that ensures bounded, predictable completion times for deep-learning jobs in the presence of myriad computation and communication variabilities. Public clouds are becoming increasingly appealing for training, and more specifically fine-tuning, large foundation models [50], for enterprises and individuals lacking resources to set their own in-house distributed training clusters [14, 16, 17, 20, 21, 28, 33, 36, 37]. OPTIREDUCE exploits the inherent resiliency and the stochastic nature of deep-learning systems to work with approximated or lost gradients and provides an efficient balance between tail performance and the resulting accuracy of the trained models. Others are already utilizing this characteristic of deep learning to optimize DDL hardware design (e.g., chip area [134, 171]), minimize traffic overhead [68, 103, 109, 154], or offload certain DDL tasks to the network switches [99, 136, 153, 157, 160]. For instance, to reduce communication time, ATP [99] and SwitchML [136] leverage fixed-point arithmetic for gradient aggregation in programmable switches with acceptable approximation loss, whereas MLT [153] prioritizes and drops packets inside switches based on model layers and gradient magnitudes to limit loss in accuracy. Various gradient-compression schemes [68, 103, 109, 154] employ lossy compression to reduce network traffic over-

[1] Cloud providers typically do not offer preferential treatment to small tenants, but even large tenants with dedicated racks face long tails when communicating across racks in the provider's network. Private communication with a hyperscaler.

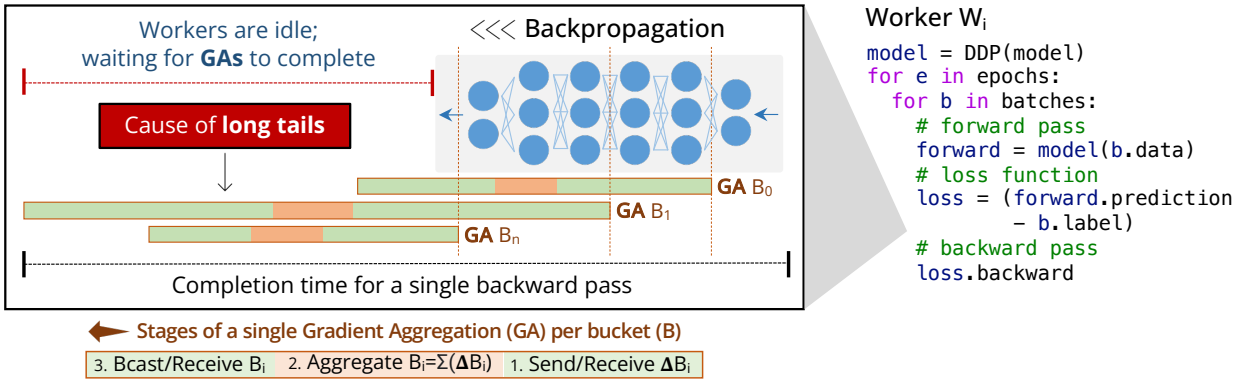


Figure 1: A backpropagation pass in distributed data-parallel (DDP) training. Multiple gradient aggregation (GA) runs share a bucket (B_i) worth of gradient entries among worker nodes (W_n), in parallel. The two send(bcast)/receive stages (1, 3) in GA incur the most time—contributing to the tail latency and stalling workers.

head (e.g., total bytes transferred) while limiting deviation from the achievable model accuracy. Similarly, hardware designers are incorporating approximate operations (e.g., approx. multipliers [134, 171]) in their architectures to minimize resource and energy usage—to scale to ever-increasing DDL models. However, these solutions are still susceptible to tail effects (e.g., slow workers and network variabilities) [47, 61, 78, 108, 127, 140, 153, 170, 173], and are not optimized for cloud environments, often times requiring direct access to the provider's network infrastructure.

In OPTIREDUCE, we exploit this resiliency and replace the (tail-prone) deterministic, run-to-completion stages of an AllReduce collective in DDL, with best-effort, *time-bounded* implementations.

- OPTIREDUCE introduces a *Transpose-Allreduce Collective (TAR)* to reduce the impact of lost gradient entries by establishing direct peer-to-peer communication among nodes in each round, rather than propagating the entries through all nodes as in Ring [121].
- It also implements an *Unreliable Bounded Transport (UBT)* to maximize the number of gradient entries received during each window. UBT introduces the notion of *adaptive timeout* to restrict the time a deep-learning job spends doing computation (aggregation) and communication (gradient sharing). Furthermore, it adds support for *dynamic incast* to dynamically adjust the number of concurrent senders per receiver in each round, thus optimizing communication by reducing the total rounds required for gradient aggregation.
- Lastly, to minimize the impact of missed or dropped gradient entries, OPTIREDUCE employs the *Hadamard Transform (HT)* [124] to ensure, for any drop pattern (e.g., tail drops), a receiver still obtains an unbiased estimate of the model's gradients resulting in a minimal loss in accuracy.

We implement OPTIREDUCE as a new AllReduce scheme inside Gloo [13],[2] a popular collective-communication library. Doing so makes OPTIREDUCE immediately compatible with existing DDL frameworks (e.g., PyTorch) without modifications. We run our experiments on various popular large deep-learning models (including BART [101], OpenAI's GPT-2 [126], and Meta's Llama 3.2) and evaluate OPTIREDUCE on CloudLab [66]—a public cloud facility for researchers—as well as under different shared environment settings using a local virtualized cluster, with varying tail-to-median latency ratios. We have made our complete OPTIREDUCE prototype [29] publicly available at https://optireduce.github.io.

Our evaluation demonstrates that OPTIREDUCE achieves, on average, 57% and 25% faster time-to-accuracy (TTA) on CloudLab compared to Gloo [13] and NCCL [87], respectively. These performance gains are even more pronounced in environments with larger tail-to-median latency differences, where OPTIREDUCE outperforms Gloo by 91% and NCCL by 35% (§5). We observe that it is the latency of the gradient aggregation (GA) step (Figure 1) that inflates three folds when operating under tail-heavy environments (e.g., public clouds), hence strengthening the need for a new collective like OPTIREDUCE. We also perform a deeper analysis of the various components of OPTIREDUCE. For example, in our evaluation, enabling Hadamard Transform mitigates the impact of tail-drops and improves TTA by 1.8× even when up to 10% of gradient entries are lost.

We begin with a background on distributed deep-learning (DDL) training—more specifically, distributed data-parallel (DDP) training—and the impact of stragglers on performance (§2). We then make a case for and present a design (§3) and implementation (§4) of a new communication-collective system, OPTIREDUCE, and evaluate how it exploits DDL's resiliency against gradient loss to improve performance (§5).

2 Background & Motivation

2.1 Distributed Deep Learning & Stragglers

Distributed deep learning (DDL) helps scale (and speedup) model training by utilizing an increasing number of hardware

[2]We pick Gloo for its ability to use both GPUs and CPUs (more suitable for a cloud environment), but OPTIREDUCE can operate with other libraries as well (e.g., NCCL [87] or MSCCL [24]).

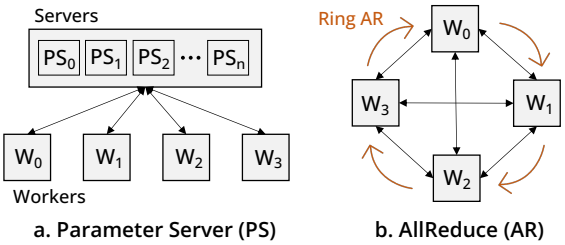


Figure 2: Architectures for gradient aggregation: Parameter Server (PS) and AllReduce (AR).

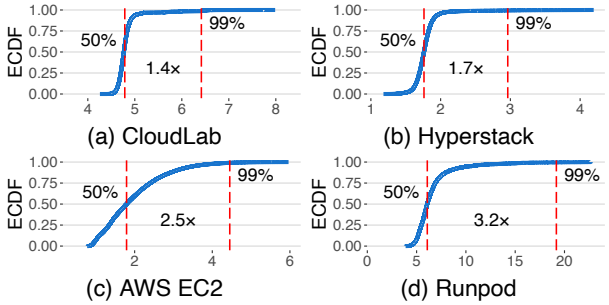


Figure 3: The latency ECDF (in milliseconds) showing tail-to-median ratio ($P_{99/50}$) observed across leading AI cloud platforms.

accelerators, e.g., GPUs [141] and TPUs [90], across server nodes [39, 55, 116]. To do so, it employs two approaches: *(i)* distributed data parallelism (DDP) [175] to run batches[3] on multiple accelerators in parallel, and *(ii)* distributed model parallelism (DMP) [91] to deploy larger models that fail to fit on a single accelerator. These approaches are orthogonal and can be used in conjunction. We focus on DDP in this paper to highlight our contributions (§3).

In distributed data parallelism (DDP), multiple worker nodes run the same deep-learning model on their portion of a training dataset, distributed evenly across nodes. Each portion is further subdivided into batches, which are processed sequentially (i.e., forward pass, loss function, and backward pass) in each epoch, Figure 1. During the forward pass, the model (e.g., a neural network) operates on a batch and generates a prediction, which is then compared with the ground truth (e.g., label) to calculate the model's loss. Next, the backward pass computes gradients using a loss function, which is used by an optimization algorithm (e.g., Stochastic Gradient Descent, SGD [52, 94]) to update the model parameters. Finally, to ensure all workers learn from what others have learned from their portion of the dataset, these gradients are averaged (reduced) and shared across all nodes, after each backward pass.

The process of accumulating, reducing, and sharing gradients back with worker nodes is referred to as *gradient aggregation* (or reduction) [62]. Originally, reduction used to happen strictly after the backward pass; however, more recently, to hide communication latency, modern frameworks (like PyTorch [106]) overlap it with the backward pass (Figure 1). As soon as a bucket (B) worth of gradient entries becomes available on a node, it is sent for reduction.[4]

Two common architectures for gradient aggregation are: Parameter Server (PS) [105] and AllReduce (AR) [59]. In the PS architecture (Figure 2a), a central server (or group of servers) receives (gathers) gradients from participating workers, aggregates (reduces) them, and broadcasts them back to all nodes. In AR (Figure 2b), instead of having separate servers, we distribute the aggregation task across workers, each reducing a subset of gradients and distributing them among themselves (e.g., Ring [121]). Both these architectures have their pros and cons. PS operates well in environments with less powerful worker machines but is bandwidth hungry—increasing linearly with the number of worker nodes. AR, especially Ring, is bandwidth-optimal but leads to longer execution delays that inflate with the number of worker nodes.

Impact of Stragglers on Performance. As shown in Figure 1, during each backward pass, all DDP worker nodes wait for the gradient aggregation (GA) operations to complete before processing the next batch of data. The forward and backward passes computation mostly takes place on a machine-learning (ML) accelerator (e.g., GPU or TPU)—a highly parallel and pipelined architecture with predictable and bounded execution time [69, 156]. Therefore, it is typically the GA operations that lead to long tails and GPU stalls (taking as much as 50% of the overall DDP processing time) [136, 137]. Our measurements across major AI cloud platforms—including AWS EC2 [3], Hyperstack [17], CloudLab [8], and RunPod AI [34]—quantify network tail latencies in a distributed training environment. Using the Gloo benchmark [12] with 2K gradients on eight nodes, we observe tail-to-median ($P_{99/50}$) latency ratios reaching up to 3.2× (Figure 3).

Various factors can contribute to this slowdown in gradient aggregation, including slow workers, transmission delays, incast effects, packet loss and retransmissions, network congestion, and more. For example, even in the PS architecture, each node sends a complete set of gradients to the parameter server, which can result in excessive drops and retransmissions, due to high incast at the ToR switch [104, 172]—hence, increasing the time to process gradients. Similarly, in Ring, a single slow worker (or a buggy link) can cause significant delays, because all nodes participate in the aggregation operation in the form of a ring.

2.2 Straggler Mitigation & Gradient Loss

Mitigating stragglers in distributed systems (as well as distributed deep learning) is an active area of research [48, 54, 60, 61, 81, 92, 132, 145, 158, 164]. One direction focuses on treating stragglers as black boxes and employs schemes, such as redundant task execution [73, 159] or skipping slow workers [57, 67, 164], to mitigate the delays due to network congestion or heterogeneous hardware, for example. They either employ backup workers and select the output of the fastest ones,

[3] A batch is a set of training data used in a given forward/backward pass.
[4] PyTorch limits these simultaneous reduction operations to two [106].

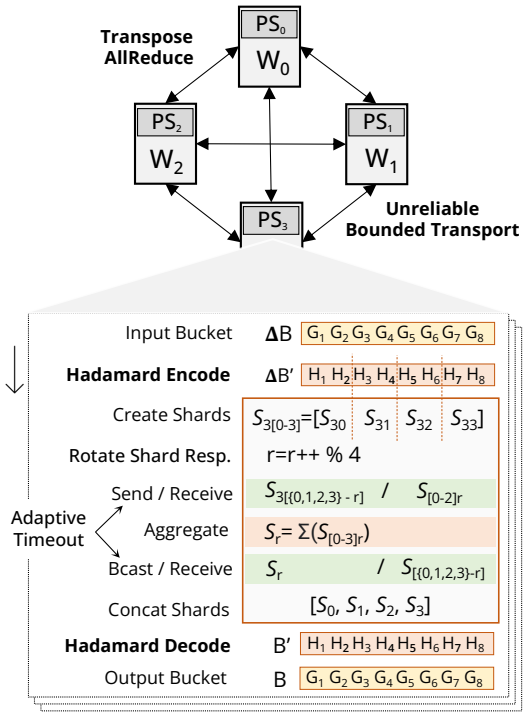


Figure 4: The OPTIREDUCE design: Transpose AllReduce with colocated parameter servers, Unreliable Bounded Transport, and Hadamard Transform.

or simply skip the slow workers altogether. However, the former can significantly increase the operational expense (in dollars). For example, training a GPT-3 model consisting of 175 billion parameters over 355 GPU-years (on a V100) [5, 53] can cost an additional $1 Million ($5.6 Million total) on the AWS instance, `p3.16xlarge`, [27] with only 16% backup nodes (i.e., 2 backups for every 10 worker nodes). This cost can further inflate by about 10× when using more powerful GPUs (e.g., A100 and H100) [6, 7], higher link speeds (e.g., 40/100 Gbps), and RDMA-enabled NICs [4]. Whereas ignoring worker nodes entirely, in the latter case, can lead to slower convergence rates and poor accuracies [57, 164]. The other direction is to replace commodity servers with specialized hardware (e.g., powerful machines with predictable performance [163, 174]) and dedicated (lossless) communication fabric [47, 95, 123]. Despite their success in HPC-like environments [117, 120, 122, 162], these solutions are not applicable in a cloud environment with myriads of tenants, all sharing the resources of the underlying data centers.

Instead of treating them as black boxes or specialized devices, we argue to replace the (tail-prone) deterministic, run-to-completion workers with their best-effort, *time-bounded* implementations. The idea is to restrict the processing time of a slow worker and utilize its partial output (gradients) in the next training phase, rather than skipping it entirely.

Resilience to Gradient Loss vs. Performance and Accuracy. Unlike traditional distributed systems (e.g., file sharing and web serving), the stochastic nature of distributed deep-learning systems provides an interesting trade-off between gradient loss (approximation or drops), performance, and accuracy. These systems (based on SGD-based optimization) [52] are shown to be resilient against estimation inaccuracies in stochastic gradients under different settings [79, 153]. For example, various gradient sparsification [68, 131, 154] and quantization [44, 109, 155] schemes employ this fact to reduce network traffic overhead. ATP [99] and SwitchML [136] utilize fixed-point arithmetic to execute gradient aggregation in programmable switches with acceptable approximation loss. Hardware designers incorporate approximate operations (e.g., approx. multipliers [134, 171]) to minimize chip area and energy usage. More recently, MLT [153] demonstrated that these deep-learning models (e.g., CNNs and LMs) are also resilient to a certain degree of gradient drops—sustaining high accuracy up to 1% of gradient loss. Additionally, the impact of gradient loss further diminishes as the number of worker nodes increases [168].

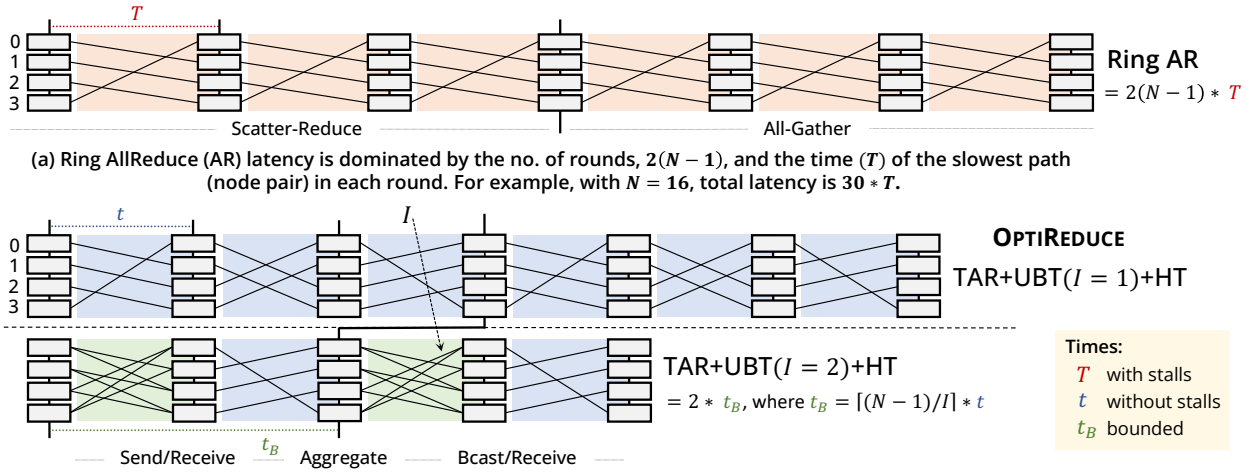
3 Design of OPTIREDUCE

We present OPTIREDUCE, a robust AllReduce communication collective system optimized to mitigate tail-latency by exploiting the unique characteristics of distributed-deep learning (DDL)—i.e., resiliency to gradient loss—to quickly reach the convergence accuracies of traditional architectures (e.g., PS and Ring) while mitigating the impact of stragglers and network variabilities.

Overview. Figure 4 shows the various components of OPTIREDUCE. *Transpose AllReduce (TAR)* (§3.1) implements a peer-to-peer collective-communication fabric, where each node also serves as a parameter server (PS)—a colocated PS architecture [68, 88]. *Unreliable Bounded Transport (UBT)* (§3.2) allows these nodes to connect with each other in a best-effort but controlled manner, and bounds the time spent by the two send(bcast)/receive stages during the AllReduce phase. The PS nodes encode (and decode) gradients in the input bucket (B), using *Hadamard Transform (HT)* (§3.3) to disperse the effect of gradient loss, before creating shards (S_{ij}) of gradients to be sent to other nodes for reduction. They iteratively *rotate shard responsibility* among themselves by maintaining a global index (r). Finally, OPTIREDUCE employs mechanisms (i.e., snapshots and selective skipping) to protect against excessive gradient loss due to transient errors or failures (§3.4); these safeguards ensure robustness and help maintain accuracy under unstable conditions. All these components operate in tandem to optimize for three competing objectives in OPTIREDUCE: maximizing performance, minimizing gradient drops, and sustaining accuracy.

3.1 Transpose AllReduce (TAR)

We begin with Transpose AllReduce (TAR), which implements a hierarchical peer-to-peer gradient-sharing strategy to limit the impact of lost gradient entries when applying the other design optimizations, discussed later. TAR operates by



(a) Ring AllReduce (AR) latency is dominated by the no. of rounds, $2(N-1)$, and the time (T) of the slowest path (node pair) in each round. For example, with $N = 16$, total latency is $30 * T$.

(b) OPTIREDUCE improves latency by reducing the no. of rounds to $2\left\lceil\frac{N-1}{I}\right\rceil$ with incast parameter (I) and bounds the path delay to t_B using Transpose AllReduce (TAR), Unreliable Bounded Transport (UBT), and Hadamard Transform (HT). For example, with $N = 16, I = 2$, the total latency is within $2 * t_B = 16 * t$, where $t \ll T$.

Figure 5: A comparison of Ring versus OPTIREDUCE.

having each node send its gradient entries directly to all other nodes during an AllReduce phase for aggregation; hence, a lost entry would only impact the aggregated results of a given node-pair in that phase. Whereas, in Ring [121], the impact is accumulated and propagated through a ring until it reaches the intended destination nodes. For instance, in our micro-benchmarks (§5.3), the Mean Squared Error (MSE) between the expected gradients and those of Ring in the presence of loss is 6× that of TAR.

3.1.1 TAR Algorithm: A Colocated PS-inspired Collective. TAR combines the key features of traditional AllReduce (i.e., P2P communication) [59, 102] and Ring (i.e., minimizing bandwidth using shards and avoiding incast via rounds) [121].

In TAR, each node acts as a worker as well as a parameter server (PS), connected together over a P2P collective-communication fabric (Figure 4). The i^{th} PS node (PS_i) receives a bucket of gradient entries (ΔB) as input from the worker process (W_i) and divides it into N shards ($S_{i[0...N-1]}$), equal to the number of nodes. Keeping the r^{th} shard it is responsible for aggregating (S_{ir}), the node PS_i sends the remaining shards ($S_{i[\{0...N-1\}-r]}$) to the neighboring nodes. At the same time, PS_i waits for its shards ($S_{[\{0...N-1\}-i]r}$) and aggregates (i.e., averages) them with S_{ir} into a single shard S_r. Next, PS_i broadcasts S_r to all nodes and receives the aggregated shards from them, $S_{[\{0...N-1\}-r]}$. Finally, these shards are concatenated into a bucket (B) and forwarded to the worker (W_i) to process the next batch of data. When $r = i$, the whole operation appears like a row-wise sum of the transpose of the shard matrix S, as shown in Figure 6; hence, the name *Transpose* AllReduce.

In P2P, each PS node communicates directly with the other nodes (instead of forming a ring). When sharing gradients, they all interact with each other twice: once sending/receiving shards to the other node and then broadcasting/receiving aggregated shards, hence limiting the impact of accumulated

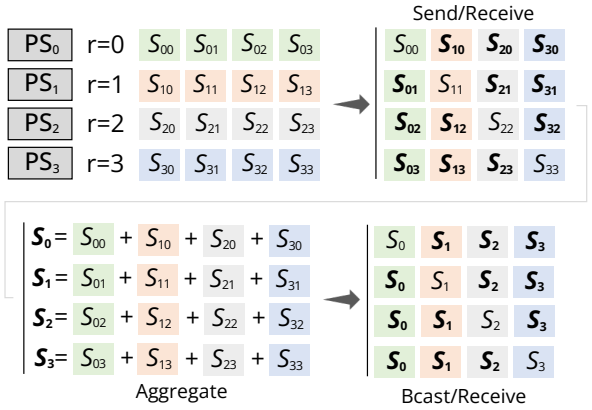


Figure 6: Transpose AllReduce algorithm: PS nodes send/receive shards S_{ij}, aggregate them, and bcast/receive to other nodes (all acquiring the same copy).

gradient loss. Sharding also alleviates the load on the PS nodes, where each node only aggregates a bucket (B) worth of gradients (rather than $N * B$).[5] Moreover, TAR utilizes the same bandwidth as Ring by sending $B * (N-1)$ bytes over the network during the two send(bcast)/receive stages. Lastly, to mitigate incast, TAR splits the communication between PS nodes over multiple rounds, where—unlike Ring with fixed node-pairs (Figure 5a)—nodes communicate with each other using a round-robin strategy, ensuring a given node-pair never repeats across rounds (Figure 5b).

3.1.2 Hierarchical 2D TAR: Scaling to Larger Node Clusters. TAR scales efficiently using a hierarchical design similar to the 2D Ring [22, 147]. By grouping nodes, it first performs intra-group communication, in parallel, to locally aggregate gradients, followed by inter-group communication for global aggregation. Doing so reduces the number of rounds from $2(N-1)$ in traditional TAR to $2(N/G-1)+(G-1)$, where

[5] PyTorch and TesnorFlow typically use a bucket size of 25 MB [106, 133].

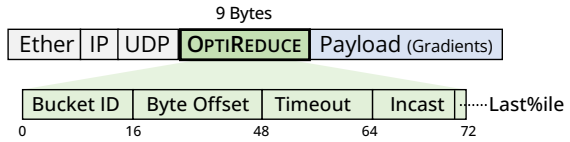


Figure 7: OptiReduce's header format

N is the total number of nodes and G is the number of groups. For example, with $N = 64$ and $G = 16$, traditional TAR requires an order of magnitude more rounds than 2D TAR—126 compared to just 21. We provide more details in Appendix A.

Summary: TAR functions similarly to Ring, yet it *reduces the impact of lost gradients* by establishing P2P communication among all nodes in each round, avoiding the propagation of losses via aggregation through intermediate nodes.

3.2 Unreliable Bounded Transport (UBT)

One of the primary causes affecting tail latency in DDL is the variability in the network behavior due to congestion [61, 153]. Current transport protocols (like TCP) further exacerbate these effects by demanding reliable, in-order delivery of packets (gradients) between training nodes—if packets are dropped or received out-of-order, TCP will stall until all gradients are received over the affected path.

However, simply replacing TCP with message-based protocols (like UDP) would not work. While UDP is faster, as it avoids packet retransmission and reordering, it lacks congestion control, which can lead to network congestion collapse. Moreover, UDP sends data at full link speed (e.g., 100 Gbps), causing excessive drops (loss of gradients) beyond what DDL models can tolerate.

To address this, we enhance UDP with adaptive timeouts, dynamic incast, and minimal rate control to create a new Unreliable Bounded Transport (UBT) protocol, which limits its computation and communication time while maximizing gradient delivery in each round. It adds a new 9-byte header, `OptiReduce` (Figure 7), to commit arriving packets (with gradients) to the right bucket and offset using the header fields, `Bucket ID` and `Byte Offset`, respectively. These fields ensure that gradients reach the correct bucket, irrespective of the ordering of the incoming packets when multiple GA operations are running in parallel (Figure 1).

3.2.1 Adaptive Timeout. UBT implements adaptive timeouts to bound the tail communication time of the send(bcast)/receive stages of the GA operations to t_B (Figure 5b). By restricting the time to t_B, we can control the worst-case execution of these stages—allowing GA operations to finish within a bounded time.

However, there are a couple of challenges with this approach. (1) How to select the value of t_B? Too small will lead to undue loss, and too high will cause unnecessary delays. Moreover, the value will vary with environmental settings (e.g., GPU type, CPU clock, vCPUs, and interface speed) and parameters (e.g., no. of nodes, bucket sizes, and incast). (2) A single lost packet, which is likely in UBT, would cause the

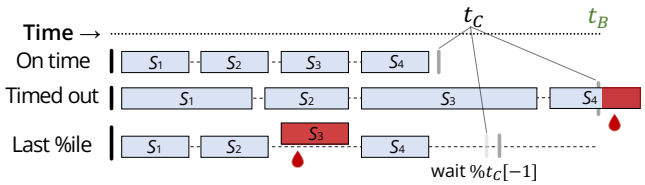


Figure 8: Different timeout strategies in OptiReduce.

GA operation to always take t_B (worst-case) time to finish. TCP, on the other hand, can perform better in some instances where communication may finish faster than waiting for the full timeout (t_B), even with retransmissions.

Selecting the Timeout Value (t_B). As shown in Figure 1, during backpropagation, multiple GA operations execute in parallel on buckets of varying sizes. For selecting t_B, during the initialization phase, we run GA with TAR and TCP, using the largest bucket, for a couple of iterations to collect completion times for both send(bcast)/receive stages. PS nodes share these values with each other using the `Timeout` field in the OptiReduce header (Figure 7).

We then form a list of these times and set t_B to the 95th %ile of that list. In §5, we show that using 20 iterations and the 95th %ile value allows OptiReduce to sustain full model accuracies while finishing up to 2× faster.

Progressing Quickly via Early Timeout. To avoid approaching t_B every time a loss happens, we introduce an early timeout scheme, which causes GA's receive stages to expire whenever there are no remaining gradient entries to read (i.e., the buffer is empty). For each bucket, we track a moving average (t_C) of completion times; we keep separate averages for both the receive stages in GA (Figure 5). The sender PS node tags the last 99th %ile packets by setting the `Last%ile` field in the header. When the buffer is empty, the receiver node checks if some of the last %ile packets have been received from all nodes. If so, it waits for an x% of t_C time before expiring (Figure 8).

The value of x% is dynamically adjusted based on the percentage of gradient entries dropped from the previous round. Starting at 10%, the goal is to maintain gradient losses between 0.01% and 0.1%. If losses exceed this range, x% is doubled until they return within the limit. If losses drop below 0.01%, x% is decreased by 1 until the desired range is reached. (The maximum x% is capped at 50%.) If gradient losses exceed 2% at any point, we activate Hadamard Transform (§3.3) to mitigate the effects of dropped gradients on convergence accuracy. [6]

We calculate t_C in the following steps. First, we compute the (expected) completion time of a given receive stage: (1) if on time, then we set t_C to the current time spent, (2) if timed out, then $t_C = t_B$, and (3) if last %ile received, then t_C is set to the expected time needed to receive all data (e.g., t_C = current time spent × total / received data). Next, we pick the median

[6] The 2% threshold is set based on prior work [153] and our evaluations §5.3.

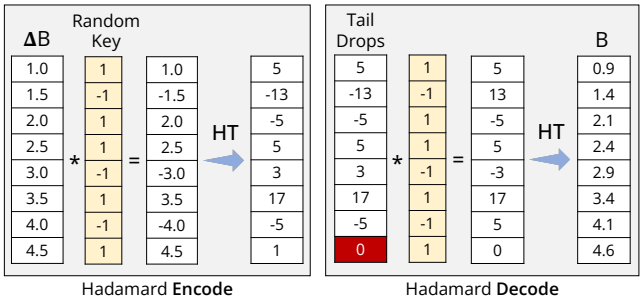


Figure 9: Dispersing the effect of lost gradients (e.g., due to tail drops) using Hadamard Transform (HT).

t_C from the values computed by the N PS nodes (shared over the `Timeout` field in the header). Finally, we calculate the moving average: $t_C = \alpha * t_C + (1-\alpha) * t_C[-1]$.

3.2.2 Dynamic Incast. UBT further introduces a notion of dynamic incast (Figure 5b). The TAR's P2P communication model lets OptiReduce alter the number of senders (I) a PS node can receive gradients from in a given round. For example, setting $I = 1$ (a single sender) would cause TAR to take the same number of rounds as Ring, $2(N-1)$; however, increasing $I = 2$ would quickly reduce these rounds by about half, $2\lceil (N-1)/2 \rceil$; and so on.

The incast parameter (I) can be configured either statically at boot time, based on the available network and node capacity (e.g., modern datacenters can handle hundreds of thousands of incast packets without degrading performance [80, 115]), or dynamically adjusted based on runtime metrics (like throughput, latency, or loss rate). In UBT, receivers dynamically modify the incast factor in response to current loss and timeout events. If the loss rate increases, the factor is reduced to alleviate congestion; conversely, if the loss rate remains low (indicating timely packet reception with no timeouts), the incast factor is increased. Receivers communicate their incast factor, I, by updating the `Incast` field in the OptiReduce header (Figure 7), and the sender then selects the smallest reported value of I for that round.

3.2.3 Minimal Rate Control. Since OptiReduce is resilient to loss, we only require a minimal scheme for rate control to prevent congestion collapse. For that, UBT employs a basic TIMELY-like rate-control mechanism [114], where the sender adjusts flow rates based on RTT feedback derived from timestamps returned by the receiver at regular intervals (every 10th packet) over a separate control channel. If the RTT (or its gradient) is below T_{low}, the sender increases the rate by α, and if the RTT exceeds T_{high}, the rate is reduced by $(1-\beta \cdot (1-T_{high}/\text{RTT}))$. In our experiments, we set $T_{low} = 25\mu$s, $T_{high} = 250\mu$s, $\alpha = 50$ Mbps, and $\beta = 0.5$, when running in a shared environment [114, 153].

Summary: UBT, in conjunction with TAR, *improves tail latency* by minimizing the impact of network congestion; adaptive timeouts bound the latency of the send(bcast)/receive stages, while dynamic incast reduces the number of communication rounds.

3.3 Dispersing Gradient Loss

Finally, to make OptiReduce resilient against drop patterns (e.g., tail drop) in the network, we employ *randomized* Hadamard Transform (HT) [144, 148, 149], which spreads the effect of a dropped gradient over the entire bucket. For example, in Figure 9, HT encodes a bucket (ΔB) and sends it over the network. Upon reception, the last gradient (in red) was lost; however, HT preserves the lost information by slightly perturbing the values of other gradients in the decoded bucket (B). The Mean Squared Error (MSE) between the decoded and received (without HT) bucket, compared to the original one, is 0.01 and 2.53, respectively. That is why, when combined with rotating shard responsibility between nodes (Figure 4), HT lets OptiReduce be more aggressive with the timeout value (t_B) while still reaching high model convergence accuracies (§5).

Summary: HT, together with TAR+UBT, *limits the effect of dropped gradients* by spreading it across the entire bucket, thus preserving the lost information. Additionally, it allows OptiReduce to operate faster, with stringent t_B values, without affecting convergence accuracies (§5).

3.4 Safeguards against Excessive Loss

OptiReduce continuously monitors gradient loss during each AllReduce phase, and if the loss exceeds a predefined threshold, it can either skip the gradient update for that round or automatically halt the training, prompting user intervention. Skipping an update helps minimize potential harm to the overall training process by discarding transient high-loss updates without impacting long-term model accuracy or completion time. This mechanism helps prevent major disruptions in the training process, ensuring users are notified of any accuracy concerns and can make necessary adjustments. Similar techniques are routinely integrated into modern deep-learning pipelines to monitor, track, and recover model accuracy [1, 2, 32, 119].

4 Implementation

We develop OptiReduce as a new collective-communication scheme inside the Gloo library (v0.5.0) [13] and integrate it with PyTorch Distributed (v1.12) [106], a widely used deep-learning framework—allowing OptiReduce to work without modification with a large body of deep-learning models (e.g., CNNs [82, 98, 138], RNNs [56, 85, 113], and Transformers [53, 63, 150]). We pick Gloo due to its simpler design and our familiarity with the codebase; however, we expect OptiReduce will yield similar benefits when operating with other popular libraries (e.g., NCCL [87] or MSCCL [24]).

We extend the C++ implementation of Gloo to support our Transpose AllReduce (TAR) collective and provide support for both reliable transport (over TCP) and our best-effort transport (over UBT). We prototype UBT as a userspace transport

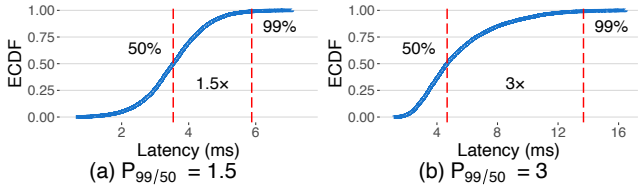


Figure 10: Our local cluster with tail-to-median latency ratio ($P_{99/50}$) of 1.5 (a) and 3 (b).

layered on UDP, including rate control, using Nvidia DPDK API (v20.11) [11].

We further add support for communication hiding in OptiReduce, i.e., running two AllReduce operations in parallel with backpropagation.[7] The sender maintains separate layer-3 port numbers to tag gradients for the two parallel AllReduce operations. On the receive side, two PMD threads poll incoming traffic (gradients) in their local receive queues. An Nvidia Connectx-6 NIC routes traffic to the respective queues based on the port numbers; we install rules in the NIC using DPDK's `rte_flow` API [10]. We also install rules to route non-OptiReduce traffic to the kernel using DPDK's Flow Bifurcation mechanism [9].[8] Doing so ensures that Gloo's kernel stack remains unaffected and other network operations, e.g., rendezvous in PyTorch DDP [106], continue uninterrupted.

To include support for adaptive timeouts, we use C++ STL library's `wait_for()` function [89], which is a blocking call that returns either when a given condition is met (such as received all gradients) or a timeout occurs. For the timeout, we pair the `wait_for()` function with Chrono library's `high_resolution_clock()` [89] to operate at nanosecond clock granularity. For Hadamard Transform, we apply a widely-used C++/CUDA implementation by HazyResearch [15], which uses GPUs to perform this operation. We use PyTorch DDP's communication hook [31] to register Hadamard's encode/decode callbacks for processing gradient buckets before and after reduction, respectively.

5 Evaluation

In this section, we provide an end-to-end comparison of OptiReduce with state-of-the-art solutions (§5.2), and microbenchmark the utility of its design components (§5.3).

5.1 Experimental Setup

5.1.1 Test Environments. We evaluate OptiReduce in both our local virtualized cluster and a real-world environment using CloudLab [66].

a) Local Virtualized Cluster. Our local testbed is a collection of four servers configured as a virtualized cluster [30]. Each machine has a 32-core AMD EPYC 7542 CPU @ 2.90 GHz, 512 GB RAM, two Nvidia Tesla V100 GPUs, and a ConnectX-5 dual-port NIC. In total, there are eight V100 GPUs, one per VM in the cluster. The VMs communicate over the network using a dedicated NIC port with Nvidia's OFED device drivers (v24.04). Both GPU and NIC interfaces are exposed to the VMs via Intel's VT-d PCIe passthrough technology [41]—allowing direct (dedicated) access to the physical functions. A programmable switch (Tofino1 [18]) connects the servers and VMs over a 25 Gbps network. Additionally, it facilitates in-network aggregation for SwitchML benchmarks (§5.3).

Recent studies from Microsoft [137], Amazon [91], and Google [61, 62] show that the tail-to-median ratio ($P_{99/50}$) for distributed workloads, including deep-learning training, ranges from 1.5× to 4× in large cloud data centers [40, 61, 106].[9] To emulate these environments and their tail characteristics in our testbed, we follow the approach of previous studies [42,45,46,151,165] by running background workloads on random nodes and links. Varying the number of concurrent workloads allows us to adjust the tail-to-median latency ratio within the network. We validate the fidelity of our scheme using the Gloo benchmark utility [12] with 2K gradients. As shown in Figure 10, our method accurately preserves the expected latency distributions, maintaining the $P_{99/50} = 1.5, 3$ ratios.

b) Public Cloud: CloudLab. We configured our real environment on CloudLab [66], a public research cloud widely shared by researchers and academics for computing and distributed systems experimentation. We provisioned eight `d7525` instances [8], each equipped with an Nvidia Ampere A30 GPU and a ConnectX-6 DX dual-port NIC, all connected via a 10 Gbps network.

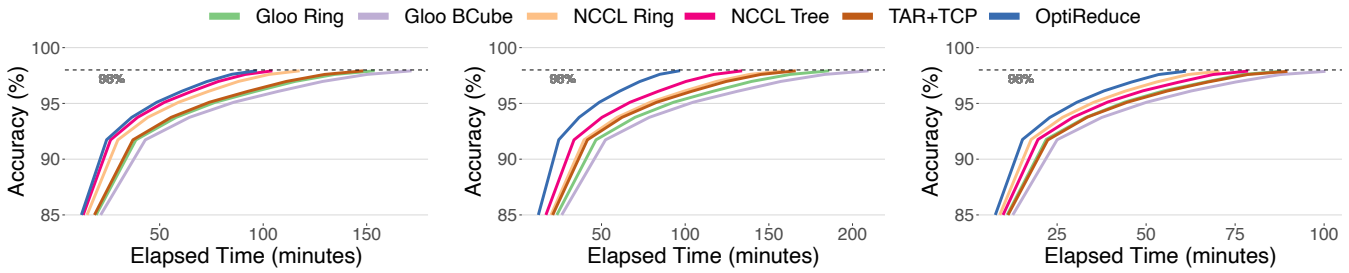
5.1.2 Baselines, Workloads, and Parameter Settings. We evaluate OptiReduce against the following baseline systems: Gloo (Ring [121] and BCube [76]), NCCL (Ring [121] and Tree [22]) with TCP, as well as a reliable version of our Transpose AllReduce (TAR) with TCP (TAR+TCP). Additionally, we evaluate BytePS and three popular compression algorithms: Top-K [142], TernGrad [155], and THC [103]. To provide further insights, we also microbenchmark OptiReduce against in-network systems such as SwitchML [136], despite their reliance on switch-level access within the provider's network, which makes them inapplicable for cloud environments.

We train a variety of language models (LMs), including BERT-base/large [63] and RoBERTa-base/large [111] on the SQuAD 2.0 dataset [129], as well as BART-base/large [101] and OpenAI GPT-2-base/large [126] on the GLUE benchmark [152] for the SST2 (Stanford Sentiment Treebank) task [139]. We further evaluate OptiReduce on additional models and tasks, which we discuss in Appendix B and C. Specifically, we train the Llama-3.2 1B model [65] on three

[7] This is consistent with existing parallelism approaches that allow for two concurrent AllReduce operations (e.g., PyTorch [106]).

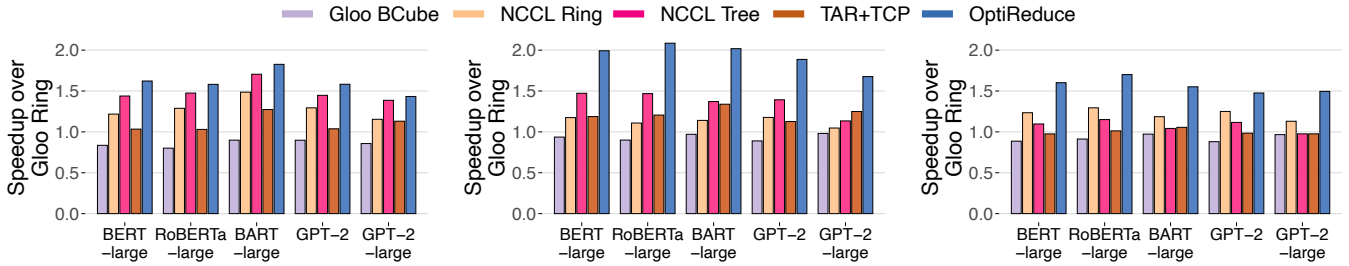
[8] Flow Bifurcation is a mechanism that lets hardware-capable NICs forward traffic directly to the userspace (DPDK thread) or the Linux kernel.

[9] Even CloudLab, a relatively small-scale cloud compared to commercial ones, exhibits a $P_{99/50}$ ratio of around 1.45.



(a) Local Cluster: $P_{99/50} = 1.5$ **(b)** Local Cluster: $P_{99/50} = 3$ **(c)** CloudLab

Figure 11: Time-to-accuracy (TTA) comparison for the OpenAI GPT-2 model with eight worker nodes.



(a) Local Cluster: $P_{99/50} = 1.5$ **(b)** Local Cluster: $P_{99/50} = 3$ **(c)** CloudLab

Figure 12: Training throughput comparison for large language models (LLMs) with eight worker nodes.

standard downstream tasks: SQuAD (extractive question answering) [129], ARC (science reasoning) [58], and MATH (symbolic mathematics) [83] (Appendix B). Additionally, we evaluate and microbenchmark OptiReduce on network-intensive models (VGG-16/19) [138] using the CIFAR-100 dataset [97] and compute-intensive models (ResNet-50/101/152) [82] with the ImageNet dataset [135] (Appendix C).

We compute the OptiReduce's timeout value (t_B) for each model using 20 iterations; we set $\alpha = 0.95$ when calculating the moving average (t_C). We use the incast parameter of $I = 1$, unless stated otherwise.

5.2 End-to-End Evaluation

We conduct end-to-end evaluations in two environments: (1) our local virtualized cluster, with tail-to-median ratios $P_{99/50} = 1.5$ (low variability) and 3 (high variability), and (2) a real public cloud, CloudLab. We compare OptiReduce against the baseline systems Gloo (Ring and BCube), NCCL (Ring and Tree), and TAR+TCP; and measure time-to-accuracy (TTA), throughput, gradient drop percentage (in bytes), and the achieved training accuracy.

Our results show that OptiReduce consistently outperforms the baselines. On our local cluster, we observe time-to-accuracy (TTA) reductions of up to (82%, 98%) compared to Gloo (Ring, BCube), and (44%, 25%) compared to NCCL (Ring, Tree), respectively. These improvements extend to CloudLab, where we see average TTA reductions of up to (47%, 67%) over Gloo (Ring, BCube), and (18%, 32%) over NCCL (Ring, Tree). Furthermore, OptiReduce achieves the same convergence accuracy as the baselines while limiting gradient entry losses to less than 0.1% of the total traffic.

• **TTA and Throughput.** Figure 11 illustrates how TTA for the five baselines and OptiReduce varies under different environments—Local cluster ($P_{99/50}$ = 1.5 and 3) and CloudLab—for the OpenAI GPT-2 model. Across all runs, OptiReduce maintains a lower TTA from the onset.[10] For example, on our local cluster with $P_{99/50} = 1.5$ (Figure 11a), OptiReduce converges in 96 minutes, while NCCL Tree takes 105 minutes, and the next best, NCCL Ring, taking 118 minutes. With $P_{99/50} = 3$, the TTA differences become more pronounced (Figure 11b). OptiReduce remains unaffected by the increased variability, maintaining its lead in TTA with a 98% accuracy and finishing in about 97 minutes. In contrast, the baselines experience significant slowdowns, with their TTA inflating by 1.41–2.18× compared to OptiReduce.

We see the same trend on CloudLab (Figure 11c), OptiReduce reaches the convergence accuracy in 60 minutes, whereas it is 71 minutes for NCCL Ring. Other baselines continue to trail behind OptiReduce, with NCCL Tree having the next-best TTA of 79 minutes.

We observe similar speedups for OptiReduce when training other models, including BERT-large, RoBERTa-large, BART-large, and GPT-2-large (Figure 12).

• **Gradient Drops and Convergence Time.** We further evaluate the drops in gradient entries and their impact on convergence time (Table 1). In our local cluster with $P_{99/50} = 1.5$, a small percentage of gradient entries is lost (i.e., 0.07%) due to OptiReduce's adaptive timeouts in UBT, causing the system to progress without waiting on stragglers. These timeouts manifest as dropped gradients in OptiReduce; whereas

[10] We observe that under ideal conditions, with $P_{99/50} = 1$ (no variability), all systems perform similarly (not shown).

Test Environment	Gloo		NCCL		TAR+TCP	OptiReduce:	*Dropped Gradients (%Entries)*
	Ring	*BCube*	*Ring*	*Tree*			
Local Cluster: $P_{99/50} = 1.5$	154	172	118	105	148	96	0.07
Local Cluster: $P_{99/50} = 3.0$	186	210	159	135	166	97	0.18
CloudLab	88	100	71	79	90	60	0.05

Table 1: Comparing the end-to-end convergence time (in minutes) of baseline systems vs. OptiReduce for OpenAI GPT-2 (total gradients, 40 TB). TAR+UDP suffers excessive drops, losing up to 30% of gradients, and fails to converge.

baseline systems stall on these stragglers. Still, OptiReduce achieves the same convergence accuracy (98% for GPT-2) as the baselines but in under 96 minutes, compared to 105 minutes for the next best, NCCL Tree. When $P_{99/50}$ increases to 3, increased congestion in the network and stragglers cause more gradient entries to be lost, but only slightly (0.18%), and does not impact OptiReduce's training accuracy and convergence time, whereas it inflates NCCL Tree's time to 135 minutes.

Similarly, in CloudLab, OptiReduce sees a 0.05% drop in gradient entries, which allows it to reach the convergence accuracy in 60 minutes, compared to its next best, NCCL Ring, taking 18% longer.

5.3 Microbenchmarks

We now evaluate the effectiveness of the individual design components in OptiReduce. We run the VGG-19 model on the CIFAR-100 dataset for these measurements using our local cluster.

• **OptiReduce's TAR topology leads to minimum dropped gradients when using a best-effort transport.** We compare the number of gradients lost across different AllReduce topologies using our Unreliable Bounded Transport (UBT). We measure Mean Squared Error (MSE) to gauge the difference between the original gradients and those received over these topologies for three different schemes on our local cluster with $P_{99/50} = 1.5$: Ring topology in Ring-AllReduce, P2P in PS, and P2P with rounds in TAR, using a 500 M tensor. Ring-AllReduce has the worst MSE (14.55)—an order of magnitude greater than TAR (2.47). The presence of fixed node pairs in Ring-AllReduce (§3.1) propagates losses, resulting in a higher deviation from the original gradients. PS also has a high MSE (9.92) due to excessive incast when all nodes send gradients to the parameter server (PS) simultaneously. In contrast, TAR avoids this by distributing P2P communication over multiple rounds.

• **UBT's dynamic incast improves OptiReduce's latency without overloading the receiver nodes.** We measure the effects of UBT's dynamic incast feature on OptiReduce's training latency. Figure 13 compares two configurations: one where we fix $I = 1$, and the other with dynamically managed incast. The results show that OptiReduce's senders can leverage buffer occupancy at receivers to increase I, thus reducing average latency by about 21% compared to always

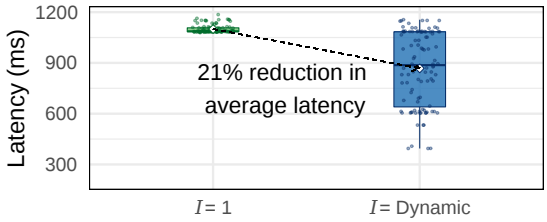
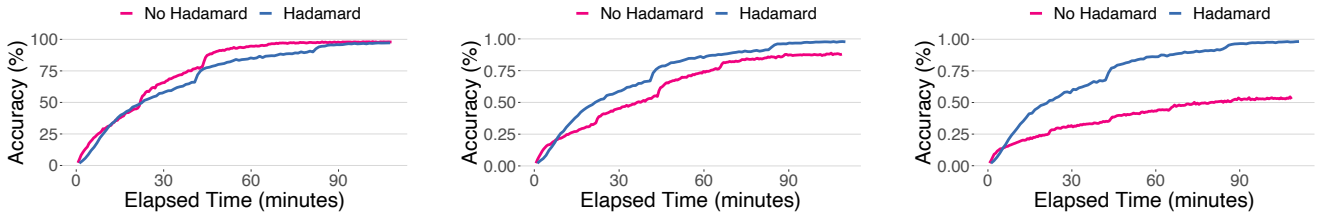


Figure 13: Latency distribution of OptiReduce with static ($I = 1$) vs. dynamic incast feature in UBT, using a synthetic 500 M-gradient AllReduce workload.

sending to a single receiver. This ability to dynamically control the incast parameter (I) allows OptiReduce to adapt itself based on the capacity of the receivers' resources, which is not the case with PS (all workers send to parameter server) or Ring-AllReduce (a receiver interacts with a single sender).

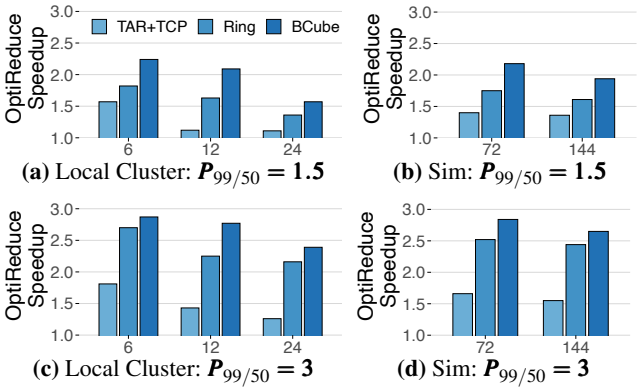
• **OptiReduce's early timeout strategy enables faster progress towards TTA.** We evaluate the effectiveness of the early timeout strategy (t_C) in OptiReduce. We disable t_C and only keep the timeout value t_B and measure its effect on training accuracy, time, and dropped gradients. We find that when training VGG-19 with $P_{99/50} = 1.5$, OptiReduce takes 130 minutes to reach convergence accuracy in 200 epochs while incurring 0.02% of gradient drops. Enabling early timeout brings this training time down by about 16% (to 112 minutes) with a similar drop rate (0.02%). By adapting t_C, OptiReduce sustains the same drop rate and finishes quickly, rather than waiting for the higher t_B value each time. We notice that with early timeout enabled, OptiReduce triggers t_C 95% more often than t_B; hence, resulting in faster TTAs.

• **OptiReduce's Hadamard Transform (HT) allows it to reach convergence accuracies even under higher percentages of dropped gradients.** Figure 14 shows the training accuracy of VGG-19 model with and without Hadamard enabled. When considering TTA, we see that Hadamard introduces some computational overhead when operating with only 1% of dropped gradient entries (Figure 14a). It takes Hadamard more time to reach convergence accuracies (around 97 minutes) compared to when it is disabled (90 minutes). However, as drops increase (5% or more), it starts to outperform the non-Hadamard instance with much faster TTAs (Figure 14b,c). Looking closely, we notice that across all dropped percentages, Hadamard is able to sustain the same TTA ($\approx$ 97 minutes)—showing its resilience to drops. In contrast, the non-Hadamard case quickly degrades and fails to achieve



(a) 1% gradient drops **(b)** 5% gradient drops **(c)** 10% gradient drops

Figure 14: Training accuracy of VGG-19 with/without Hadamard Transform (HT) in OPTIREDUCE.



(a) Local Cluster: $P_{99/50} = 1.5$ **(b)** Sim: $P_{99/50} = 1.5$

(c) Local Cluster: $P_{99/50} = 3$ **(d)** Sim: $P_{99/50} = 3$

Figure 15: OPTIREDUCE speedup over baseline systems (TAR+TCP, Ring, BCube) with increasing #workers using a synthetic 500 M-gradient AllReduce workload.

convergence accuracy even under 10% drops. The percentage drops include both drops incurred due to network variabilities (e.g., congestion and retransmissions) and gradients that a slow worker could not send due to timeouts.

• **OPTIREDUCE scales with increasing number of worker nodes, consistently maintaining high speedups.** To demonstrate OPTIREDUCE's performance at scale, we first run tests using CPU-based worker nodes on our local cluster. We compare OPTIREDUCE with TAR+TCP and Gloo (Ring and BCube) on a synthetic AllReduce workload, aggregating 500 M gradient entries across 6–24 nodes (Figure 15a, c).[11] Next, we conduct simulations with larger clusters (72 and 144 nodes), similar in sizes to prior works [43, 100, 107, 153]—using latencies sampled from the local cluster and scaled for higher node counts (Figure 15b, d). Across all tests, OPTIREDUCE consistently delivers high speedups, achieving 2× improvements over Ring and BCube in high-tail environments, $P_{99/50} = 3$ (Figure 15c, d).

• **Unlike OPTIREDUCE, lossy/compression schemes are vulnerable to tail effects in shared environments.** Though OPTIREDUCE is orthogonal to sparsification and quantization techniques, our comparison (Figure 16) shows that lossy/compression schemes (e.g., Top-K, TernGrad, and THC) fail to effectively address tail effects. While these schemes reduce the volume of gradient entries shared, they rely on a static evaluation of how much data to compress (or drop) a priori before transmission. In contrast, OPTIREDUCE handles loss

[11] We exclude NCCL from this comparison as it relies on GPUs.

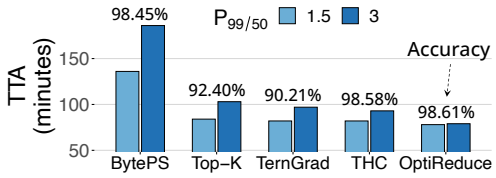


Figure 16: OPTIREDUCE comparison with lossy/compression schemes (BytePS, Top-K, TernGrad, & THC), showing TTA and their convergence accuracy.

in real-time, dynamically adapting to network conditions and minimizing tail latency. For instance, THC matches OPTIREDUCE in convergence accuracy but takes 4% and 18% longer to complete under $P_{99/50} = 1.5$ and 3, respectively. Other schemes perform even worse, either requiring 2× more time to converge or stalling at lower accuracies due to their lossy compression, failing to improve end-to-end TTA even with additional training epochs [103, 153].

• **In-network aggregation (INA) approaches struggle with tail effects, while OPTIREDUCE remains unaffected.** In-network aggregation (INA) methods, such as SwitchML [136], reduce network latency through hardware-accelerated aggregation within the network. However, they remain vulnerable to tail effects—significantly inflating their completion times as the tail-to-median ratio increases. For instance, in a low-tail environment ($P_{99/50} = 1.5$), SwitchML performs 52% faster than OPTIREDUCE. However, as the tail-to-median ratio increases from $P_{99/50} = 1.5$ to 3, its completion time rises by about 2.1×, even surpassing OPTIREDUCE by 28%. In contrast, OPTIREDUCE remains unaffected by this change while reaching convergence. By bypassing stragglers and proceeding without waiting for all gradients, OPTIREDUCE is better suited for shared and high-tail environments. Moreover, OPTIREDUCE's design can be extended to incorporate INA, potentially achieving similar speedups in low-tail environments—an avenue we plan to explore in future work.

6 Limitations & Future Work

In the current AllReduce design, there are two primary bottlenecks: (a) in the computation (or reduction) phase and (b) in the communication phase. We explore potential solutions for both bottlenecks in the subsequent sections.

a) Accelerators for Reduction. In OPTIREDUCE, we primarily focus on bounding the execution time of the two send/receive stages in AllReduce (Figure 1). The reduction stage, i.e.,

the process of averaging gradients together, still happens on CPUs. However, as models grow and gradient sizes increase, the reduction stage can become a bottleneck. Rather than opting for the most extreme case of offloading all of AllReduce to network switches [99,136], we can instead consider offloading the reduction operation on the end-host server (similar to how we accelerate GEMMs using GPUs) [141]. Modern SmartNICs [19,25,26], with onboard FPGAs and ML accelerators, can present a promising opportunity for accelerating reduction. But, doing so requires rethinking and redesigning the application interface (API) between OptiReduce and SmartNICs. We hope OptiReduce to serve as a stepping stone for research in this direction.

b) Accelerators for Network Transport. As with reduction, network transport can also become a bottleneck with link/interface speeds reaching 400 Gbps+. Existing offloads, like RDMA [77], provide high-bandwidth communication between nodes by moving data to/from the main memory and the network without engaging the host CPU. However, these implementations are still susceptible to tail effects in the network (e.g., packet drops and retransmissions). We hope OptiReduce's transport design can offer guidance in building new offloads for network transport, particularly with support for unreliable bounded protocols. As a next step, we could explore offloading OptiReduce's transport onto hardware using RDMA's Unreliable Connected (UC) or Unreliable Datagram (UD) queue pairs [49]. However, these implementations currently suffer from excessive packet drops when packets arrive out of order [23]. We plan to address these challenges in future work.

7 Related Work

Lossy Architectures for Accelerating Allreduce Collectives. THC [103] presents compression-aware gradient synchronization architectures for DNN training, introducing homomorphic compression to reduce bandwidth through quantization. OmniReduce [68] introduces the concept of a streaming aggregation, which exploits parameter sparsity to maximize effective bandwidth use by sending only non-zero data blocks. MLT [153] configures network switches to prioritize and drop packets based on model layers and gradient magnitudes, leveraging inter-packet order-independency to balance load. In contrast, OptiReduce exploits DDL's resiliency to gradient drops in mitigating tail effects while sustaining convergence accuracies in the cloud without requiring access to the provider's network. It could apply techniques like OmniReduce to reduce network usage for models with sparse gradients or use quantization methods similar to THC.

Accelerating Deep Learning using In-Network Computing. SHArP [75] is a hierarchical aggregation protocol and architecture in Nvidia Switches (e.g., IB-2 [35]), which builds an overlay reduction tree for aggregating data flowing through the switch. SwitchML [136] accelerates distributed training by using a programmable data-plane device (e.g., Intel Tofino [18,51]) to aggregate the model updates from multiple workers in the network. To overcome the switch memory and computational constraints, they co-design the in-switch processing with end-host protocols (e.g., sliding window of parameters) for handling drops. ATP [99] is an in-network aggregation solution similar to SwitchML, but for deep learning, and is designed to provide a dynamic, best-effort in-network aggregation service for multi-tenant multi-switch clusters. Unlike these solutions, OptiReduce does not require specialized hardware or access to the provider's network.

Optimizing Deep Learning for Enterprise and HPC Environments. Cassini [128] is a network-aware job scheduler for ML clusters in HPC environments that optimizes network resource usage by interleaving communication patterns of ML jobs, reducing congestion and improving cluster performance. Meta's recent paper [72] presents a custom backend for distributed deep-learning training targeting enterprise data centers. It optimizes network topology, job scheduling, placement, and data transport to improve training performance, efficiency, and scalability. On the other hand, OptiReduce offers a resilient and tail-optimal solution for deep-learning training in the cloud.

8 Conclusion

OptiReduce leverages distributed-deep learning's (DDL) resiliency to lost gradients and achieves speedups of up to (70%, 30%), on average, over existing frameworks (Gloo, NCCL), in shared environments (e.g., public clouds). OptiReduce implements a domain-specific Transpose Allreduce collective algorithm with unreliable bounded transport (UBT) featuring adaptive timeouts, while mitigating the impact of gradient loss using Hadamard Transform. It delivers higher tail performance (e.g., TTA and training throughput) while preserving DDL models' convergence accuracy and limiting gradient drops to under 0.1%. Looking forward, we hope OptiReduce inspires further exploration of the tradeoff between tail performance and training accuracy in processing contemporary deep-learning models.

Acknowledgements

We sincerely appreciate the guidance of our shepherd, Changhoon Kim, as well as Ashwin Murthy, Roop Mukherjee, Leo Liu, Minlan Yu, Tushar Krishna, Ajay Brahmakshatriya, and the anonymous reviewers for their valuable feedback in strengthening this paper. We also thank Roop Mukherjee, Leo Liu, Ali Imran, and Ali Aqdas for helping with the artifact and initial background studies on tail behavior in distributed training environments, conducted on Nvidia's internal shared clusters and AWS EC2 instances. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by NSF awards CAREER-2338034 and CNS-2211381; and through a Google Research Scholar Award. Support also came in part via a generous gift from Nvidia.

References

[1] Collaborative AI Platform. https://www.evidentlyai.com. Last accessed: 01/2025.

[2] neptune.ai AI Experiment Tracker. https://neptune.ai. Last accessed: 01/2025.

[3] Amazon AWS EC2. https://aws.amazon.com/ec2. Last accessed: 01/2025.

[4] Amazon AWS EC2 Elastic Fabric Adapter (EFA). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa.html. Last accessed: 01/2025.

[5] Amazon EC2 P3 Instance. https://aws.amazon.com/ec2/instance-types/p3. Last accessed: 01/2025.

[6] Amazon EC2 P4 Instance. https://aws.amazon.com/ec2/instance-types/p4. Last accessed: 01/2025.

[7] Amazon EC2 P5 Instance. https://aws.amazon.com/ec2/instance-types/p5. Last accessed: 01/2025.

[8] CloudLab Hardware. https://docs.cloudlab.us/hardware.html. Last accessed: 08/2024.

[9] DPDK Flow Bifurcation Guide. https://doc.dpdk.org/guides-20.11/howto/flow_bifurcation.html. Last accessed: 01/2025.

[10] DPDK Generic Flow API Docs. https://doc.dpdk.org/guides-20.11/prog_guide/rte_flow.html. Last accessed: 01/2025.

[11] DPDK v20.11. https://doc.dpdk.org/guides-20.11. Last accessed: 01/2025.

[12] Gloo Benchmarking Utility. https://github.com/facebookincubator/gloo?tab=readme-ov-file#benchmark. Last accessed: 01/2025.

[13] Gloo Collective-Communication Library. https://github.com/facebookincubator/gloo. Last accessed: 01/2025.

[14] Grand View Research: Cloud AI Market Report. https://www.grandviewresearch.com/industry-analysis/cloud-ai-market-report. Last accessed: 01/2025.

[15] HazyResearch's Open Source Hadamard CUDA Implementation. https://github.com/HazyResearch/structured-nets. Last accessed: 01/2025.

[16] How Cloud Computing Revolutionized Business Operations. https://www.forbes.com/sites/emilsayegh/2023/11/28/how-cloud-computing-revolutionized-business-operations-and-what-lies-ahead. Last accessed: 01/2025.

[17] Hyperstack: Complete Guide to Machine Learning for Cloud 2024. https://www.hyperstack.cloud/blog/case-study/complete-guide-to-machine-learning-for-cloud-2024. Last accessed: 01/2025.

[18] Intel Tofino P4 Switch. https://www.intel.com/content/www/us/en/products/details/network-io/programmable-ethernet-switch/tofino-series.html. Last accessed: 01/2025.

[19] Intel® FPGA SmartNIC. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic.html. Last accessed: 01/2025.

[20] Machine Learning in the Cloud. https://www.run.ai/guides/machine-learning-in-the-cloud. Last accessed: 01/2025.

[21] Machine Learning in the Cloud: What Are the Benefits. https://symphony-solutions.com/insights/machine-learning-in-the-cloud-what-are-the-benefits. Last accessed: 01/2025.

[22] Massively Scale Your Deep Learning Training with NCCL 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4. Last accessed: 01/2025.

[23] Mellanox Support for Out-of-order Packets Data Placement. https://docs.nvidia.com/networking/display/mlnxofedv543100/out-of-order+(ooo)+data+placement. Last accessed: 01/2025.

[24] Microsoft Collective-Communication Library. https://github.com/microsoft/msccl. Last accessed: 01/2025.

[25] napa:tech; SmartNICs. https://www.napatech.com/products. Last accessed: 01/2025.

[26] NVIDIA ConnectX Series SmartNIC. https://www.nvidia.com/en-us/networking/ethernet-adapters. Last accessed: 01/2025.

[27] OpenAI's GPT-3 Language Model: A Technical Overview. https://lambdalabs.com/blog/demystifying-gpt-3. Last accessed: 01/2025.

[28] Optimizing Artificial Intelligence and Machine Learning with Cloud Computing. https://www.orange-business.com/en/blogs/optimizing-artificial-intelligence-machine-learning-cloud-computing. Last accessed: 01/2025.

[29] OptiReduce Public Artifact. https://github.com/OptiReduce.

[30] Proxmox VM Environment. https://pve.proxmox.com/wiki/Main_Page. Last accessed: 01/2025.

[31] PyTorch DDP Communication Hook. https://pytorch.org/docs/stable/ddp_comm_hooks.html. Last accessed: 01/2025.

[32] PyTorch TorchFT: Per-step Fault Tolerance. https://github.com/pytorch/torchft. Last accessed: 01/2025.

[33] Redress Compliance: Exploring Cloud-Based Machine Learning Platforms. https://redresscompliance.com/exploring-cloud-based-machine-learning-platforms. Last accessed: 01/2025.

[34] RunPod - The Cloud Built for AI. https://www.runpod.io. Last accessed: 01/2025.

[35] Switch-IB™ 2 EDR. https://network.nvidia.com/files/doc-2020/pb-switchib2-edr-switch-silicon.pdf. Last accessed: 01/2025.

[36] The Power of Machine Learning in the Cloud: Transforming Business Operations. https://mentormate.com/blog/the-power-of-machine-learning-in-the-cloud-transforming-business-operations. Last accessed: 01/2025.

[37] The Rise of Cloud Machine Learning. https://encapture.com/the-rise-of-cloud-machine-learning. Last accessed: 01/2025.

[38] Criteo: Terabyte Click Logs Dataset. https://labs.criteo.com/2013/12/download-terabyte-click-logs, 2013.

[39] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467*, 2016.

[40] Sepehr Abbasi, Shiva Ketabi, Ali Munir, Mahmoud Bahnasy, and Yashar Ganjali. DWTCP: Ultra Low Latency Congestion Control Protocol for Data Centers. *arXiv:2207.05624*, 2022.

[41] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3), 2006.

[42] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *USENIX NSDI*, 2022.

[43] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. Harmony: A Congestion-free Datacenter Architecture. In *USENIX NSDI*, 2024.

[44] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via Gradient Quantization and Encoding. *NeurIPS*, 2017.

[45] Mohammad Alizadeh and Tom Edsall. On the Data Path Performance of Leaf-spine Datacenter Fabrics. In *High-Performance Interconnects*. IEEE, 2013.

[46] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM*, 2014.

[47] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[48] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[49] Dotan Barak. RDMA Aware Networks Programming User Manual. *NVIDIA Docs*, 2015.

[50] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the Opportunities and Risks of Foundation Models. *arXiv:2108.07258*, 2021.

[51] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *ACM SIGCOMM CCR*, 2013.

[52] Léon Bottou. Stochastic gradient descent tricks. *Neural Networks: Tricks of the Trade: Second Edition*, 2012.

[53] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models Are Few-shot Learners. *NeurIPS*, 2020.

[54] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv:1604.00981*, 2016.

[55] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv:1512.01274*, 2015.

[56] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv:1406.1078*, 2014.

[57] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric P Xing. Solving the straggler problem with bounded staleness. *White paper, Carnegie Mellon University*, 2013.

[58] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think You Have Solved Question Answering? Try ARC, The AI2 Reasoning Challenge. *arXiv:1803.05457*, 2018.

[59] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In *Programming Environments for Massively Parallel Distributed Systems, Springer*, 1994.

[60] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *USENIX NSDI*, 2017.

[61] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), 2013.

[62] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large Scale Distributed Deep Networks. *NeurIPS*, 2012.

[63] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805*, 2018.

[64] Xianzhi Du, Mostafa El-Khamy, Jungwon Lee, and Larry Davis. Fused DNN: A Deep Neural Network Fusion Approach to Fast and Robust Pedestrian Detection. In *IEEE winter conference on applications of computer vision (WACV)*, 2017.

[65] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 Herd of Models. *arXiv:2407.21783*, 2024.

[66] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *USENIX ATC*, 2019.

[67] Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. Slow and Stale Gradients Can Win The Race: Error-runtime Trade-offs in Distributed SGD. In *International conference on artificial intelligence and statistics*, 2018.

[68] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient Sparse Collective Communication and its Application to Accelerate Distributed Deep Learning. In *ACM SIGCOMM*, 2021.

[69] Xu Fei, Xu Jianian, Chen Jiabin, Chen Li, Shang Ruitao, Zhou Zhi, and Liu Fangming. iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud. *arXiv:2211.01713*, 2022.

[70] Shuo Feng, Huiyu Zhou, and Hongbiao Dong. Using Deep Neural Network with Small Dataset to Predict Material Defects. *Materials & Design*, 162, 2019.

[71] Mingsheng Fu, Hong Qu, Zhang Yi, Li Lu, and Yongsheng Liu. A Novel Deep Learning-based Collaborative Filtering Model for Recommendation System. *IEEE transactions on cybernetics*, 49(3), 2018.

[72] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over Ethernet for Distributed Training at Meta Scale. In *ACM SIGCOMM*, 2024.

[73] James Salamy Ayush Sharma Manya Ghobadi and Muriel Médard. FlexEnt: Entropy Coding to Curb Stragglers in Large-Scale Distributed Machine Learning. *Workshop on AI Systems at SOSP*, 2019.

[74] Abhinav Goel, Caleb Tung, Yung-Hsiang Lu, and George K Thiruvathukal. A Survey of Methods for Low-power Deep Learning and Computer Vision. In *IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020.

[75] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *IEEE International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016.

[76] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *ACM SIGCOMM*, 2009.

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

[78] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*, 2015.

[79] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *ICML*, 2015.

[80] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *ACM SIGCOMM*, 2017.

[81] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Addressing the Straggler Problem for Iterative Xonvergent Parallel ML. In *ACM Symposium on Cloud Computing*, 2016.

[82] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE CCVPR*, 2016.

[83] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring Mathematical Problem Solving with the MATH Dataset. *NeurIPS*, 2021.

[84] Brian Heredia, Joseph D Prusa, and Taghi M Khoshgoftaar. Social Media for Polling and Predicting United States Election Outcome. *Social Network Analysis and Mining*, 8(1), 2018.

[85] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8), 1997.

[86] Zhenhua Huang, Guangxu Shan, Jiujun Cheng, and Jian Sun. TRec: An Efficient Recommendation System for Hunting Passengers with Deep Neural Networks. *Neural Computing and Applications*, 31(1), 2019.

[87] Sylvain Jeaugey. NCCL 2.0. In *GPU Technology Conference (GTC)*, 2017.

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

[89] Nicolai M Josuttis. The C++ Standard Library: A Tutorial and Reference. *Addison-Wesley*, 2012.

[90] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[91] Can Karakus, Rahul Huilgol, Fei Wu, Anirudh Subramanian, Cade Daniel, Derya Cavdar, Teng Xu, Haohan Chen, Arash Rahnama, and Luis Quintela. Amazon Sagemaker Model Parallelism: A General and Flexible Framework for Large Model Training. *arXiv:2111.05972*, 2021.

[92] Can Karakus, Yifan Sun, Suhas Diggavi, and Wotao Yin. Straggler Mitigation in Distributed Optimization through Data Encoding. *NeurIPS*, 2017.

[93] Asif Iqbal Khan, Junaid Latief Shah, and Mohammad Mudasir Bhat. CoroNet: A Deep Neural Network for Detection and Diagnosis of COVID-19 from Chest X-Ray Images. *Computer methods and programs in biomedicine*, 196, 2020.

[94] Jack Kiefer and Jacob Wolfowitz. Stochastic Estimation of The Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 1952.

[95] Michael Ko, Daniel Eisenhauer, and Renato Recio. A Case for Convergence Enhanced Ethernet: Requirements and Applications. In *IEEE International Conference on Communications*, 2008.

[96] Michał Koziarski and Bogusław Cyganek. Image Recognition with Deep Neural Networks in Presence of Noise–Dealing with and Taking Advantage of Distortions. *Integrated Computer-Aided Engineering*, 24(4), 2017.

[97] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *White paper, University of Toronto*, 2009.

[98] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Communications of the ACM*, 60(6), 2017.

[99] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *USENIX NSDI*, 2021.

[100] Jason Lei and Vishal Shrivastav. Seer: Enabling Future-Aware Online Caching in Networked Systems. In *USENIX NSDI*, 2024.

[101] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *arXiv:1910.13461*, 2019.

[102] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: Distributed Data-Parallelism for Existing ML Applications. In *European Conference on Computer systems*, 2015.

[103] Minghao Li, Ran Ben Basat, Shay Vargaftik, ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu. THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression. In *USENIX NSDI*, 2024.

[104] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. *NeurIPS*, 27, 2014.

[105] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter Server for Distributed Machine Learning. In *Big learning NIPS workshop*, 2013.

[106] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv:2006.15704*, 2020.

[107] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. Flow Scheduling with Imprecise Knowledge. In *USENIX NSDI*, 2024.

[108] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *ACM CoNEXT*, 2016.

[109] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv:1712.01887*, 2017.

[110] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task Deep Neural Networks for Natural Language Understanding. *arXiv:1901.11504*, 2019.

[111] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A Robustly Optimized BERT Pretraining Approach. *arXiv:1907.11692*, 2019.

[112] Pradeep Kumar Mallick, Seuc Ho Ryu, Sandeep Kumar Satapathy, Shruti Mishra, Gia Nhu Nguyen, and Prayag Tiwari. Brain MRI Image Classification for Cancer Detection using Deep Wavelet Autoencoder-Based Deep Neural Network. *IEEE Access*, 7, 2019.

[113] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent Neural Network based Language Model. In *Interspeech*, 2010.

[114] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM CCR*, 2015.

[115] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-driven Low-latency Transport Protocol using Network Priorities. In *ACM SIGCOMM*, 2018.

[116] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training Deep Networks in Spark. *arXiv:1511.06051*, 2015.

[117] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely Dataflow System. In *ACM SOSP*, 2013.

[118] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *ACM SOSP*, 2019.

[119] David Nigenda, Zohar Karnin, Muhammad Bilal Zafar, Raghu Ramesha, Alan Tan, Michele Donini, and Krishnaram Kenthapadi. Amazon Sagemaker Model Monitor: A System for Real-time Insights into Deployed Machine Learning Models. In *ACM SIGKDD*, 2022.

[120] Xue Ouyang, Changjian Wang, and Jie Xu. Mitigating Stragglers to Avoid QoS Violation for Time-critical Applications through Dynamic Server Blacklisting. *Future Generation Computer Systems*, 101, 2019.

[121] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 69(2), 2009.

[122] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE Conference on Supercomputing*, 2003.

[123] Gregory F Pfister. An Introduction to the Infiniband Architecture. *High performance mass storage and parallel I/O*, 42(617-632), 2001.

[124] William K Pratt, Julius Kane, and Harry C Andrews. Hadamard Transform Image Coding. *Proceedings of the IEEE*, 1969.

[125] Yanxing Qi, Yi Guo, and Yuanyuan Wang. Image Quality Enhancement using a Deep Neural Network for Plane Wave Medical Ultrasound Imaging. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 2020.

[126] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models Are Unsupervised Multitask Learners. *OpenAI blog*, 2019.

[127] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. *ACM SIGCOMM CCR*, 2011.

[128] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. In *USENIX NSDI*, 2024.

[129] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ Questions for Machine Comprehension of Text. *arXiv:1606.05250*, 2016.

[130] S Ramesh, C Yaashuwanth, K Prathibanandhi, Adam Raja Basha, and T Jayasankar. An Optimized Deep Neural Network based DoS Attack Detection in Wireless Video Sensor Network. *Journal of Ambient Intelligence and Humanized Computing*, 2021.

[131] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance Sparse Communication for Machine Learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[132] Alessandro Rigazzi. DC-S3GD: Delay-Compensated Stale-Synchronous SGD for Large-Scale Decentralized Neural Network Training. In *ACM/IEEE Third Workshop on Deep Learning on Supercomputers (DLS)*, 2019.

[133] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks. In *USENIX NSDI*, 2022.

[134] Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. Chopping off the Tail: Bounded Non-Determinism for Real-Time Accelerators. *IEEE Computer Architecture Letters (CAL)*, 2021.

[135] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander Berg, and Li Fei-Fei. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 2015.

[136] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*, 2021.

[137] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *USENIX NSDI*, 2023.

[138] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv:1409.1556*, 2014.

[139] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over A Sentiment Treebank. In *Empirical Methods in Natural Language Processing*, 2013.

[140] Akshitha Sriraman, Sihang Liu, Sinan Gunbay, Shan Su, and Thomas F Wenisch. Deconstructing the Tail at Scale Effect Across Network Protocols. *arXiv:1701.03100*, 2017.

[141] Dave Steinkraus, Ian Buck, and PY Simard. Using GPUs for Machine Learning Algorithms. In *IEEE ICDAR*, 2005.

[142] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with Memory. *NeurIPS*, 2018.

[143] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. In *IEEE ICCV*, 2017.

[144] Ananda Theertha Suresh, X Yu Felix, Sanjiv Kumar, and H Brendan McMahan. Distributed Mean Estimation with Limited Communication. In *ICML*, 2017.

[145] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *ICML*, 2017.

[146] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and Efficient Foundation Language Models. *arXiv:2302.13971*, 2023.

[147] Yuichiro Ueno and Rio Yokota. Exhaustive Study of Hierarchical Allreduce Patterns for Large Messages between GPUs. In *IEEE/ACM CCGRID*, 2019.

[148] Shay Vargaftik, Ran Ben Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben Itzhak, and Michael Mitzenmacher. Eden: Communication-efficient and Robust Distributed Mean Estimation for Federated Learning. In *ICML*, 2022.

[149] Shay Vargaftik, Ran Ben-Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben-Itzhak, and Michael Mitzenmacher. Drive: One-bit Distributed Mean Estimation. *NeurIPS*, 2021.

[150] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *NeurIPS*, 2017.

[151] Kashi Venkatesh Vishwanath and Amin Vahdat. Evaluating Distributed Systems: Does Background Traffic Matter? In *USENIX ATC*, 2008.

[152] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. GLUE: A Multi-task Benchmark and Analysis Platform for Natural Language Understanding. *arXiv:1804.07461*, 2018.

[153] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific Network Transport for Distributed DNN Training. In *USENIX NSDI*, 2024.

[154] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient Sparsification for Communication-efficient Distributed Optimization. *arXiv:1710.09854*, 2017.

[155] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. *NeurIPS*, 2017.

[156] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4), 2009.

[157] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Programmable Switches for In-Networking Classification. In *IEEE INFOCOM*, 2021.

[158] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking Transport Layer Design for Distributed Machine Learning. In *APNet*, 2019.

[159] Guojun Xiong, Gang Yan, Rahul Singh, and Jian Li. Straggler-resilient Distributed Machine Learning with Dynamic Backup Workers. *arXiv:2102.06280*, 2021.

[160] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *ACM HotNets*, 2019.

[161] Cheng Xu, Duo Chai, Jie He, Xiaotong Zhang, and Shihong Duan. InnoHAR: A Deep Neural Network for Complex Human Activity Recognition. *IEEE Access*, 7, 2019.

[162] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *USENIX NSDI*, 2013.

[163] Neeraja J Yadwadkar and Wontae Choi. Proactive Straggler Avoidance using Machine Learning. *White paper, University of Berkeley*, 2012.

[164] Yauhen Yakimenka, Chung-Wei Weng, Hsuan-Yin Lin, Eirik Rosnes, and Jörg Kliewer. Straggler-Resilient Differentially-Private Decentralized Learning. In *IEEE Information Theory Workshop (ITW)*, 2022.

[165] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. ACC: Automatic ECN Tuning for High-speed Datacenter Networks. In *ACM SIGCOMM*, 2021.

[166] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using Trio: Juniper Networks' Programmable Chipset for Emerging In-network Applications. In *ACM SIGCOMM*, 2022.

[167] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative Study of CNN and RNN for Natural Language Processing. *arXiv:1702.01923*, 2017.

[168] Chen Yu, Hanlin Tang, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ce Zhang, and Ji Liu. Distributed Learning over Unreliable Networks. In *ICML*, 2019.

[169] Xin Yuan, Weite Li, Kui Lin, and Jinglu Hu. A Deep Neural Network Based Hierarchical Multi-Label Classifier for Protein Function Prediction. In *International Conference on Computer, Information and Telecommunication Systems (CITS)*, 2019.

[170] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *ACM SIGCOMM*, 2012.

[171] Georgios Zervakis, Hassaan Saadat, Hussam Amrouch, Andreas Gerstlauer, Sri Parameswaran, and Jörg Henkel. Approximate Computing for ML: State-of-the-Art, Challenges and Visions. In *ACM ASPDAC*, 2021.

[172] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC*, 2017.

[173] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. *ACM SIGARCH*, 2016.

[174] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: A Fault-tolerant Resource Management and Job Scheduling System at Internet Scale. In *VLDB Endowment*, 2014.

[175] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized Stochastic Gradient Descent. *NeurIPS*, 2010.

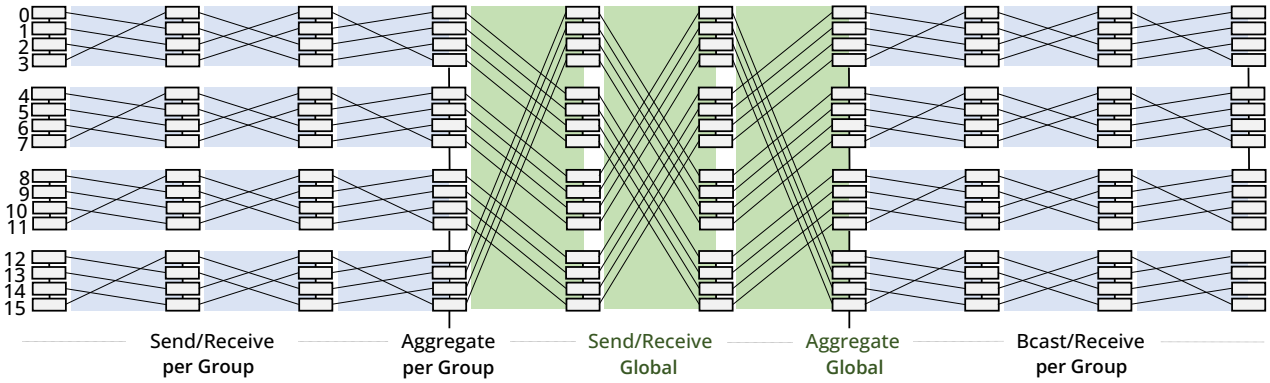


Figure 17: Hierarchical 2D TAR Algorithm.

Environment	Benchmark	Gloo		NCCL		TAR+TCP	OptiReduce:		
		Ring	*BCube*	*Ring*	*Tree*		Time	Accuracy	Test Acc.
$P_{99/50} = 1.5$	ARC	84	113	77	75	76	61	60.45 [+0.45]	39.97 [-0.47]
	MATH	195	254	180	171	175	130	30.56 [+0.18]	30.29 [+0.23]
	SQuAD	4072	5402	3391	3464	3723	3182	46.77 [-0.21]	38.64 [+0.08]
$P_{99/50} = 3.0$	ARC	155	161	128	120	86	61	60.44 [+0.44]	39.91 [-0.53]
	MATH	308	390	299	243	189	131	30.14 [-0.24]	30.09 [+0.03]
	SQuAD	5793	8057	5677	5243	4120	3220	46.45 [-0.53]	38.57 [+0.01]

Table 2: Comparing convergence time (in minutes) and accuracy (% [Δ]), as well as test accuracy (% [Δ]) for the Llama-3.2 1B model across various tasks and environments; [Δ] reports deviation from the baseline accuracy (e.g., Gloo and NCCL).

A Hierarchical 2D TAR Algorithm: Scaling to Larger Node Clusters

In the hierarchical TAR design (Figure 17), nodes are grouped to optimize both intra-group and inter-group communication, reducing the total number of rounds and connections required for AllReduce. For example, with $N = 64$ total nodes divided into $G = 16$ groups, each node communicates only with its corresponding rank across the groups. The number of rounds reduces from $2(N-1) = 126$ in traditional TAR to $2(N/G-1)+(G-1) = 21$ rounds. The algorithm works in three steps:

- **Intra-group Communication:** Nodes within each group perform send/receive operations followed by aggregation, in parallel, resulting in the *locally* aggregated shard for their rank—taking $(N/G-1)$ rounds.
- **Inter-group Communication:** Corresponding ranks across groups then perform send/receive operations, followed by aggregation, to get the *globally* aggregated shard for their rank—adding another $(G-1)$ rounds.
- **Broadcast Phase:** Finally, nodes within the group broadcast their aggregated shards, which are concatenated to form the globally aggregated gradient bucket—an additional $(N/G-1)$ rounds.

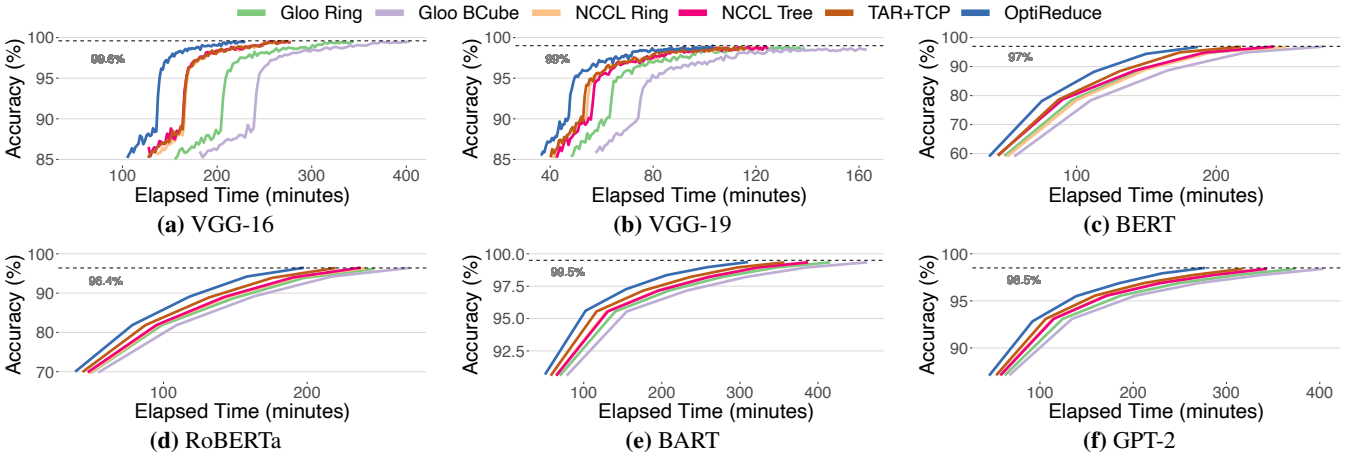
This hierarchical design significantly reduces communication overhead, improving scalability and efficiency for large-scale distributed training.

B Benchmarking Llama-3.2 1B Model

Using our local testbed (§5.1), we evaluate OptiReduce with the Llama-3.2 1B model [65] on three well-known downstream tasks: SQuAD (extractive question answering) [129], ARC (science reasoning) [58], and MATH (symbolic mathematics) [83], across both low-tail ($P_{99/50} = 1.5$) and high-tail ($P_{99/50} = 3.0$) environments. Table 2 provides a detailed comparison of training times across all schemes. OptiReduce consistently demonstrates performance improvements across all tasks. Compared to NCCL, it achieves speedups of 1.35× on MATH, 1.25× on ARC, and 1.08× on SQuAD, averaging a 1.24× improvement. The gains are even more pronounced against Gloo, with speedups of 1.73×, 1.61×, and 1.49× respectively, averaging 1.61×. These improvements scale further under high-tail conditions, reaching speedups of up to 2.1× while preserving baseline model convergence and test accuracies.

C Network and Compute Intensive Models & Base LMs

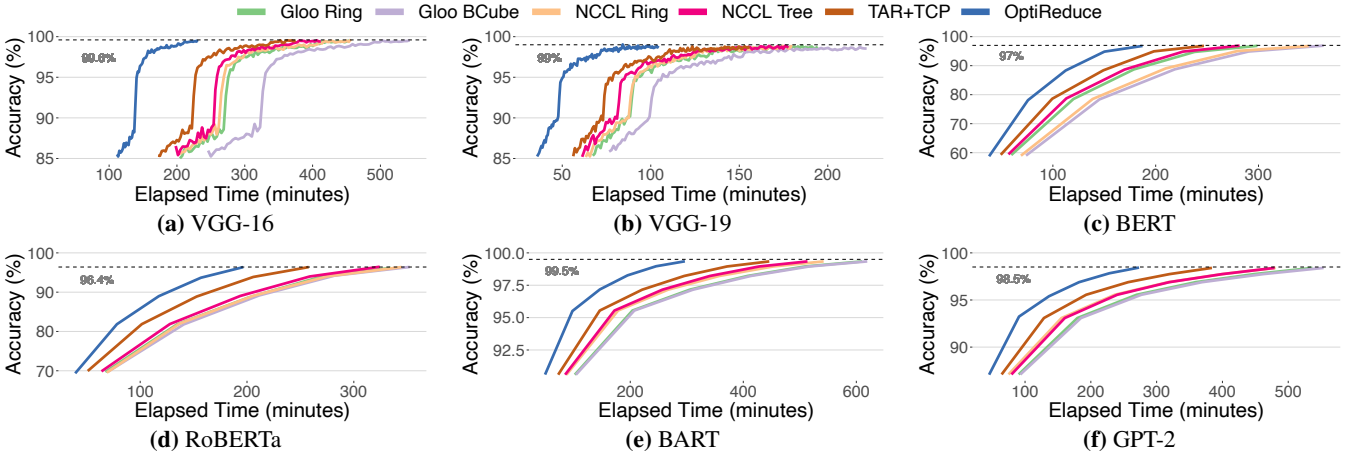
In this section, we present time-to-accuracy (TTA) plots for additional models, including computer vision models (ResNet-50/101/152, VGG-16/19) and base LMs (BERT, RoBERTa, BART, and GPT-2). The experiments use the same local testbed setup described in §5.1, but with six worker nodes (VMs). We compare results across two environment configurations, characterized by tail-to-median ratios ($P_{99/50}$) of 1.5



(a) VGG-16 (b) VGG-19 (c) BERT

(d) RoBERTa (e) BART (f) GPT-2

Figure 18: Time-to-accuracy (TTA) of baseline systems vs OptiReduce with tail-to-median ratio: $P_{99/50} = 1.5$.



(a) VGG-16 (b) VGG-19 (c) BERT

(d) RoBERTa (e) BART (f) GPT-2

Figure 19: Time-to-accuracy (TTA) of baseline systems vs OptiReduce with tail-to-median ratio: $P_{99/50} = 3$.

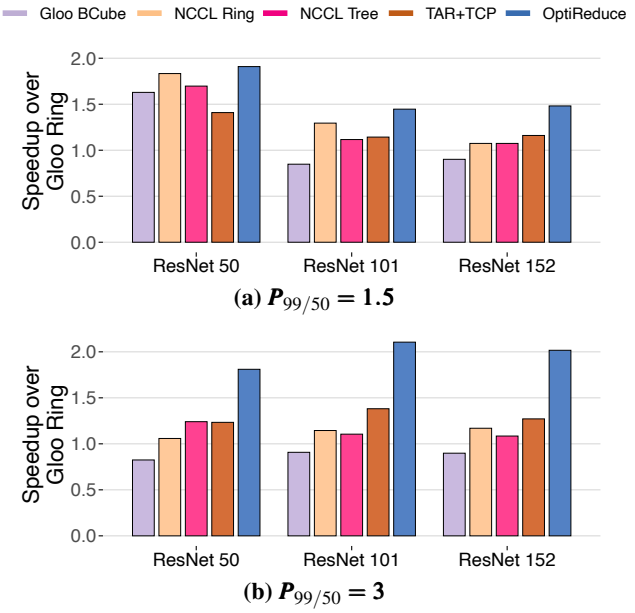
(low variability) and 3 (high variability).

C.1 Time-to-accuracy (TTA)

We observe similar gains for these network-intensive models (VGG-16/19) and base LMs, with up to (66%, 75%) and (50%, 51%) reductions in TTA, on average, compared to Gloo (Ring, BCube) and NCCL (Ring, Tree), respectively—Figure 18 ($P_{99/50} = 1.5$) and Figure 19 ($P_{99/50} = 3$). OptiReduce achieves the same convergence accuracy as the baselines while limiting lost gradients to less than 1.5%, on average, of the communicated traffic.

C.2 Training Throughput (Speedup)

While compute-intensive models like ResNets [82] typically do not gain significant advantages from optimized communication [103, 153], their performance can be impacted in shared environments (such as public clouds) due to long-tail latencies. Our evaluations reflect this, where OptiReduce demonstrates notable improvements over baseline systems, achieving average speedups of 22% over NCCL and 53% over Gloo for three ResNet models (50/101/152) across both environment configurations (Figure 20).



(a) $P_{99/50} = 1.5$

(b) $P_{99/50} = 3$

Figure 20: Training throughput for computationally-intensive ResNet models on the ImageNet dataset.

Efficient Direct-Connect Topologies for Collective Communications*

Liangyu Zhao† Siddharth Pal‡ Tapan Chugh† Weiyang Wang§ Jason Fantl‡
Prithwish Basu‡ Joud Khoury‡ Arvind Krishnamurthy†

†*University of Washington* ‡*RTX BBN Technologies* §*MIT CSAIL*

Abstract

We consider the problem of distilling efficient network topologies for collective communications. We provide an algorithmic framework for constructing direct-connect topologies optimized for the latency vs. bandwidth trade-off associated with the workload. Our approach synthesizes many different topologies and communication schedules for a given cluster size and degree, then identifies the best option for a given workload. Our algorithms start from small, optimal base topologies and associated schedules, using techniques that can be iteratively applied to derive much larger topologies and schedules. Additionally, we incorporate well-studied large-scale graph topologies into our algorithmic framework by producing efficient communication schedules for them using a novel polynomial-time algorithm. Our evaluation uses multiple testbeds and large-scale simulations to demonstrate significant performance benefits from our derived topologies and schedules.

1 Introduction

Collective communication operations involve concurrently aggregating and distributing data on a cluster of nodes and are used in both machine learning (ML) and high-performance computing (HPC). With the improved computational capabilities of accelerators, collective operations are a significant overhead in large-scale distributed ML training [20,44,64,76].

An emerging approach to address these challenges has been to employ various forms of optical circuit switching to achieve higher bandwidth at reasonable capital expenditure and energy costs [29,30,40,43,73,77,81]. Hosts communicate using a limited number of optical circuits that can be reconfigured at timescales appropriate for the hardware, thus exposing network topology as a configurable component. We refer to this setting as *direct-connect* with circuits configured and fixed for an appropriate duration.

Existing optical-circuit-based ML systems [30,43,73,81] fit this direct-connect model but do not exploit the flexibility topology reconfiguration offers. Collective operations such as allreduce are still limited to a few well-known algorithms that can fit the degree constraints of the optical fabric (e.g., rings, multi-rings, tori) and accept the consequent performance tradeoffs. For example, ring allreduce, while bandwidth-efficient, has a high graph diameter, causing high total-hop latency. A double binary tree, on the other hand, has a logarithmic diameter but suffers from load imbalances and bandwidth inefficiencies. Conversely, the broader spectrum of well-known collective algorithms that achieve desired latency and bandwidth (e.g., recursive-doubling, Bruck algorithm) [72,79] use dynamic communication patterns ideal for switch networks but are ill-suited for degree-constrained direct-connect networks.

To fill this gap, we seek to identify new custom-built topologies and communication schedules for direct-connect networks. We pose the following question: *How to efficiently construct high-performance direct-connect topologies and communication schedules for collectives given the network's performance characteristics and degree constraints?*

This question poses several challenges. First, jointly optimizing *both* the network topology and the corresponding communication schedule is intractable at a large scale. Prior efforts reduce the search cost by optimizing only one or the other (e.g., schedules for a given topology [10,65,76] or topology permutations while retaining a ring schedule [77]). The combination of topological structure and communication schedule as degrees of freedom explodes the search space, making this a seemingly intractable problem. Second, the optimization must carefully consider the workload and the network's performance characteristics when distilling a topology and schedule. For example, minimizing the topology's diameter is ideal not only for latency-sensitive allreduce at small data sizes but also for all-to-all throughput; however, this could come at the cost of load imbalance across links in bandwidth-sensitive allreduce at large data sizes. Finally, lowering the synthesized schedules to the underlying hardware and runtimes [26,49] in an efficient way requires careful scheduling to achieve the desired performance in practice.

Our work addresses these issues by developing an algorithmic toolchain for quickly synthesizing efficient topologies and schedules for collective communications.

1. We devise a range of **expansion techniques** for synthesizing custom large-scale network topologies and schedules. The expansions start with small, optimal topologies and communication schedules and expand them to achieve near-optimal large-scale topologies and schedules.

2. We devise a **polynomial-time schedule generation algorithm** to produce optimal collective communication schedules for large-scale topologies with specific symmetry properties. This exposes many well-known topologies as options for the direct-connect network fabric.

*Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

3. We devise a **topology enumeration and search algorithm** to identify the best option for a target cluster and workload by exploring the *Pareto-efficient* options that provide different tradeoffs for bandwidth efficiency, total-hop latency, and also all-to-all throughput.
4. We develop **compilers** to realize our optimized schedules. We offer efficient implementations for both GPUs and CPUs, integrating with ML frameworks (e.g., PyTorch) through the MSCCL [49] and oneCCL [26] runtimes.

We evaluate our approach using two testbeds: a 12-node GPU cluster capable of topology reconfiguration and torus clusters on Frontera [69] supercomputer with up to 54 CPU nodes. Our techniques reduce collective communication times by $>30\%$ for DNN training on the GPU testbed and up to $3.1\times$ in supercomputing settings. Simulations for large-scale DNN training show up to an order of magnitude reduction in total communication time from topology and schedule optimization. Our schedule generation algorithm is orders of magnitude faster than the state-of-the-art (e.g., SCCL [10] and TACCL [65]), capable of producing schedules for topologies with thousands of nodes in a minute.

2 Background & Related Work

2.1 Network Fabric

Our work identifies topologies and schedules helpful for a broad range of settings, such as *switchless physical circuits*, *patch-panel optical circuits*, and *optical circuit switches*. While these options differ in cost and reconfigurability [77], they are all significantly cheaper than packet-switch solution [29, 40, 77] and can benefit from our work.

Switchless physical circuits require the least amount of fabric hardware. However, the topology must remain reasonably static for long periods, as the reconfiguration is manual. *Patch-panel optical circuits* provide a higher degree of reconfigurability by using a mechanical solution (e.g., robotic arms) to perform physical reconfigurations through a patch panel. The reconfigurations occur on the scale of minutes, but the patch panel itself can scale to a large number of duplex ports and is reasonably cheap (e.g., 1008 ports at $100 per port [70]). Both options can benefit from a carefully curated topology optimized for the workload, but they require it to remain static for a job given the reconfiguration time.

Commercial *optical circuit switches (OCS)* can perform reconfigurations in $\approx$10ms, are more expensive than patch panels, and scale to fewer ports (e.g., Polatis 3D-MEMS switch has 384 ports at $520 per port). Though OCSes support faster reconfigurations, the delays are still too high to support the rewiring of the circuits *during* a typical collective operation.[1] Thus, they cannot take advantage of algorithms designed for full-bisection switches, such as recursive halving/doubling [72, 79], that exploit high logical degree over time to provide both latency and bandwidth optimality. Thus, OCSes can also benefit from the custom-built and low-degree topologies synthesized by our approach.

All of these optical technologies allow for a shared cluster to be split into multiple subclusters for running separate jobs [29], so each job can be configured with its own topology. Further, unidirectional topologies are technically feasible on optical testbeds. Unidirectionality gives greater freedom in topology design and can enable lower-diameter networks.

Evaluation Target: In this paper, we use a reconfigurable optical patch panel to configure and evaluate different topologies. Given the high reconfiguration costs for the patch panel, we identify an efficient topology that will remain static for the duration of a job. Nevertheless, our techniques could be used to derive topologies for finer reconfiguration timescales if the hardware can efficiently support frequent reconfigurations.

2.2 Related Work

Several existing optical-circuit-based ML systems [30, 43, 73, 77, 81] fit the direct-connect model; however, they rely on existing implementations of collectives. Typically, communication libraries for ML training [20, 56, 64] offer either ring collective for high-latency bandwidth-optimal transfers or tree collective, which has logarithmic latency but suffers from load-imbalances across links. Other topologies such as mesh, tori, hypercubes, etc., have also been explored in HPC systems [3, 8, 11, 12, 18, 23, 58], but their bandwidth-latency tradeoff choices are limited as well. Bandwidth and latency optimal collectives for switch networks such as recursive-doubling, Bruck algorithm [72, 79], BlueConnect [13], etc., are unsuitable for direct-connect networks, because their one-to-one communication patterns fail to utilize all available links, and they assume a fully connected network.

Our work uniquely considers joint optimization of *both* the network topology and the corresponding collective communication schedule at a large scale, while prior work either optimizes one or the other. For instance, TopoOpt [77] generates customized shifted-ring topologies to optimize concurrent collective and non-collective communications for hybrid data-parallel [15, 37, 38] and model-parallel [28, 52, 67] DNN training, respectively. The collective communications in TopoOpt still use existing ring collectives. Consequently, when data parallelism, for example, dominates the workload, TopoOpt's performance suffers from similar latency issues present in existing ring collectives. Our effort is complementary as it synthesizes new topologies and schedules for collectives that span the entire cluster, but we do not optimize sub-cluster communications for hybrid parallelism. Extending our work to jointly optimize topologies and schedules for hybrid parallelism is future work.

Recent work like Blink [76], SCCL [10], and TACCL [65] also focus on generating a collective schedule for a given

[1] Research prototypes [1, 48] support μs to ns reconfigurations using overlay-hop relays and Valiant load balancing (VLB). While this is valuable for generic workloads, collectives have structured communication patterns, and it would be ideal to realize them without incurring the VLB overheads.

M	total data size	α	single-hop latency
N	number of nodes	B	total egress bandwidth of a node
S	data shard ($\lvert S\rvert = \frac{M}{N}$)	B/d	bandwidth of a single link
C	data chunk ($C \subseteq S$)	$T_L(A)$	total-hop latency of schedule
d	degree of topology	$T_B(A)$	bandwidth runtime of schedule
V_G	vertex/node set of G	$T_L^*(N,d)$	Moore optimality (Def 10)
E_G	edge/link set of G	$T_B^*(N)$	bandwidth optimality $\frac{M}{B} \cdot \frac{N-1}{N}$
$D(G)$	graph diameter of G	$N_x^+(u)$	nodes at distance x from u
Table 9 for graph symbols		$N_x^-(u)$	nodes at distance x to u

Table 1: Summary of Important Notations

topology. However, they all involve NP-hard optimizations that severely limit their scalability. SCCL is capable of generating optimal schedules, but it fails to generate a schedule in a reasonable time when the topology size is beyond 30 nodes. TACCL improves the scalability of SCCL by using communication sketches and also handles switch networks, but it sacrifices schedule performance and is still limited in scalability. In our approach, we either synthesize the schedule along with the topology or rely on a polynomial-time schedule generation technique that is provably optimal for networks with certain symmetry properties.

Generic large-scale topologies are typically not optimized for collective communications but for general datacenter traffic [6, 31, 33, 68, 75, 80]. Our framework can incorporate any degree-constrained regular topology (e.g., low-diameter expander graphs [25, 61]) and generate candidate schedules.

2.3 All-to-All Throughput

While we optimize allreduce, reduce-scatter, and allgather, the performance of all-to-all communication is also crucial for training DNN models like Mixture of Experts (MoE) [19, 34, 36, 60] and Deep Learning Recommendation Model (DLRM) [45, 53, 54]. Unlike other collectives, the scheduling of all-to-all can be easily formulated and efficiently solved as a multi-commodity flow (MCF) problem [4, 21, 32, 75, 78]. However, the graph diameter of the underlying topology is critical for all-to-all throughput [4, 41, 42, 48, 77]. The intuition is that if the nodes are far from each other, then all-to-all flows cost more *bandwidth tax* [47, 77] (i.e., the bandwidth of the flow multiplied by the length of the flow). With a fixed *network capacity* (i.e., the total number of links times link bandwidth), longer flows reduce the available bandwidth for each flow, thus decreasing all-to-all throughput. In this work, we construct topologies and associated schedules that are high-performance in both allreduce-type collectives and all-to-all by **deriving efficient allreduce-type schedules on existing or our synthesized low-diameter topologies**.

3 Formal Model of Collective Communications

We provide a formal model of ***reduce-scatter***, ***allgather***, and ***allreduce*** collectives. In each operation, there are N nodes operating on a vector of data of total size M. The data can be divided into N **shards**. In *reduce-scatter*, each node i reduces the i-th shard from all other nodes; in *allgather*, each node i broadcasts the i-th shard to all other nodes; in *allreduce*, each node i ends up with the fully reduced vector of data.

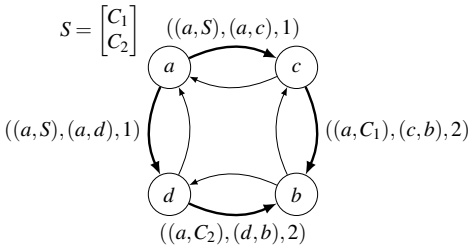


Figure 1: The allgather schedule of complete bipartite graph $K_{2,2}$. Shard S is divided into two half chunks C_1 and C_2. From a, at the 1st comm step, a sends the entire shard S to both c and d. At the 2nd comm step, c and d send the two half chunks C_1 and C_2, respectively, to b. Thus, every node receives the full shard from a. By applying similar broadcast from c, b, d in parallel, we have a complete BW-optimal *allgather* schedule with $T_L = 2\alpha, T_B = \frac{M}{B} \cdot \frac{3}{4}$.

Throughout the paper, we elaborate only on *allgather* schedule construction because the other two collectives are direct transformations. Since *allgather* and *reduce-scatter* are, respectively, simultaneous broadcasts and reductions for each node, we can **construct *reduce-scatter* schedules** in bidirectional topologies by simply reversing the communications in *allgather* schedules [11]. In unidirectional topologies, we utilize graph transposition to achieve a similar transformation (Appendix B).[2] **To construct an *allreduce* schedule,** we concatenate *reduce-scatter* and *allgather*.

3.1 Communication Topology & Schedule

The network topology is modeled as a directed graph (digraph) $G = (V, E)$, where V denotes the set of nodes ($\lvert V\rvert = N$) and E denotes the set of directed links/edges. The direct-connect network imposes a constraint that all nodes have degree d, which is the number of connection ports on each host and is typically low and independent of N.

A communication **algorithm** (G, A) uses the communication **schedule** A on topology G. Schedule A can be specified as what chunk C is communicated over which link in which **communication (comm) step** t. We define **chunk** C as a subset of shard S. Both C and S are specified as *index sets* of elements. Typically, S is interval $[0, 1]$ representing the whole shard, and C is some subinterval. We denote v's chunk C as (v, C), which is a subset of v's starting shard (v, S). Let $((v, C), (u, w), t)$ denote that v's chunk C is sent by node u to its neighbor w at comm step t. Schedule A then is specified as a list of tuples $((v, C), (u, w), t)$. Figure 1 gives an example of an allgather schedule in such a tuple notation. Within a schedule, chunks can be different-sized subsets of S. Appendix B gives formal definitions of reduce-scatter/allgather schedules.

3.2 Cost Model

We use the well-known α-β cost model [24]. The cost of sending a message of size H over a link is $\alpha + \beta H$. This cost comprises two components: the constant single-hop latency α and a bandwidth component β, which is the inverse of link

[2] The main text of this paper focuses on high-level ideas of various techniques. We provide detailed mathematical analyses in the appendix.

bandwidth, i.e., $\beta = \frac{1}{b}$. This simple model has been shown to be appropriate for GPU interconnects [10, 35, 65]. In our analysis, we use node bandwidth B with $B = db$. In this paper, we focus on homogeneous networks, although some techniques also support heterogeneous ones (Appendix E.3).

The runtime of a schedule A can be broken down into a total-hop latency component and a bandwidth component. The **total-hop latency** component $T_L(A) = t_{\max}\alpha$, where $t_{\max}$ is the number of comm steps. $T_L(A)$ represents the cost of performing schedule A on an infinitesimal amount of data. The bandwidth component $T_B(A)$, or **bandwidth (BW) runtime**, is the sum of the BW runtime of each comm step, i.e., $\sum_t T_B(A_t)$. The BW runtime of comm step t is the max amount of data transmitted by a link within the comm step, divided by link bandwidth $b = B/d$. For N-node d-regular graphs, the optimal runtime of both *reduce-scatter* and *allgather* is approximately $\alpha \log_d N + \frac{1}{B} \cdot \frac{M(N-1)}{N}$. The 1st term represents the total-hop latency required for communicating across the diameter of a topology, while the 2nd term represents the transmission time for any node to send/recv $N-1$ shards in reduce-scatter/allgather. **One should not confuse total-hop latency with overall latency,** which is the sum of total-hop latency and BW runtime. We omit the computational time of reduction and discuss this in Appendix C.4.

We analyze the optimality of total-hop latency and BW runtime separately. An algorithm (G, A) is optimal in one component if no (G', A') with the same N, d can perform better. For BW runtime, an algorithm is **bandwidth (BW) optimal** iff its T_B equals $T_B^*(N) := \frac{M}{B} \cdot \frac{N-1}{N}$. For total-hop latency, given G, the lowest T_L achievable is $\alpha \cdot D(G)$, where $D(G)$ is the graph diameter of G. Thus, the optimal total-hop latency equals the smallest diameter of any N-node d-regular graph, which remains an open question in graph theory [50]. Therefore, we define *Moore optimality* based on *Moore bound*, which provides a lower bound for diameter given N, d and thus a well-defined $T_L^*(N, d)$. An algorithm is **Moore optimal** iff $T_L = T_L^*(N, d)$. Moore optimal topologies have the lowest diameters, which is also ideal for all-to-all throughput. Appendix C gives formal definitions of optimalities.

Ring allreduce has a total-hop latency that is linear in N, while the BW runtime is optimal. *Double binary trees* (DBT), on the other hand, offer the advantage of logarithmic total-hop latency but have suboptimal BW performance. Our work offers a range of topologies that are Pareto-efficient in total-hop latency and BW performance.

4 Overview of Our Approach

Direct-connect topologies can typically be categorized as either **low-hop** topologies, which have low diameters (e.g., expander graphs) suited for all-to-all throughput and small-data allgather/reduce-scatter/allreduce, or **load-balanced** topologies, which have simplistic structure (e.g., ring, torus) with easy load-balanced schedule for large-data allgather/reduce-scatter/allreduce (see Table 2). We seek to jointly identify network topologies and schedules that achieve high performance in both categories to the extent possible. Specifically, this entails the challenging task of constructing load-balanced allgather[3] schedules for low-hop topologies.

Topology Type	Small-Data Allreduce (Total-Hop Latency T_L)	Large-Data Allreduce (BW Perf T_B)	All-to-All Throughput
Low-Hop	✓	–	✓
Load-Balanced	–	✓	–

Table 2: The tradeoffs of low-hop topology vs. load-balanced topology. Reduce-scatter and allgather perform similarly to allreduce.

At a small scale, one could handpick a topology such as the complete bipartite graph $K_{2,2}$ defined at $N = 4, d = 2$. $K_{2,2}$ is both low-hop and load-balanced that a Moore- and BW-optimal allgather could be manually constructed (Figure 1). But how do we scale the topology and the schedule to larger sizes? Our work approaches this problem with two tools: *expansion techniques* (§5) and *BFB schedule generation* (§6).

Expansion Techniques: Given a base topology and its schedule, expansion techniques can expand them into a larger topology and associated schedule with minimal loss in performance. We call the resulting topologies **synthesized topologies**. The base topologies are small in scale, such as $K_{2,2}$ in Figure 1, for which straightforward schedules exist or an exhaustive search for the schedule is feasible. The line graph expansion, for example, can then expand $K_{2,2}$ and its schedule in Figure 1 to an allgather for $N = 4 \cdot 2^n$, for arbitrarily large n, while retaining a node degree of 2. Multiple expansion techniques can be composed to achieve the desired N and d.

Breadth-First-Broadcast (BFB) Schedule Generation: Besides synthesized topologies, we can use known topologies from graph theory (e.g., twisted torus, expander graphs). We call them **generative topologies** as they can be instantiated at various Ns and ds. Generative topologies are often low-hop, beneficial for total-hop latency and all-to-all throughput. The problem, though, is that efficient load-balanced collective schedules are not known for many of these topologies, and existing schedule generation methods [10, 65] are intractable even at moderate scales. Our work offers BFB, a polynomial-time schedule generation that can yield efficient schedules for large-scale topologies. For allgather, it performs a breadth-first broadcast from each node and uses linear programs to balance the workload on links. Although not always optimal, BFB schedules are provably optimal for many topologies exhibiting certain symmetries. For instance, BFB can generate a schedule with the lowest total-hop latency and BW optimality on any torus, including those with *unequal dimensions*.

With expansion techniques and BFB schedules, our *topology finder* (§5.4) assembles a large pool of topologies and schedules, identify *Pareto-efficient* ones from a low-hop vs. load-balanced perspective, and select from them for a given workload. When two options are Pareto-efficient, one must be better than the other in either low-hop (i.e., total-hop latency) or load-balanced (i.e., BW performance) but not in both. We

[3] We construct allreduce and reduce-scatter from allgather schedules.

choose low-hop options for workloads requiring all-to-all throughput and small-data allgather/reduce-scatter/allreduce, and load-balanced options for large-data allgather/reduce-scatter/allreduce. Finally, the *compiler* (§7) lowers the chosen topology and schedule to the runtime and hardware.

5 Expansion Techniques

We present three techniques that can be applied to construct near-optimal large-scale *synthesized topologies* and schedules by expanding small-scale topologies and associated schedules. The three techniques provide different options for increasing the size of the topology and the per-node degree while preserving either total-hop latency or BW optimality of the base graph and schedule (Table 3). While we describe the techniques in the context of allgather, corollary 1.2 in §B implies equivalent constructions for reduce-scatter and allreduce.

5.1 Line Graph Expansion

We borrow the line graph transformation from graph theory [22], which transforms an input graph G into a larger graph $L(G)$ as follows: every edge in G becomes a node in $L(G)$, and two nodes in $L(G)$ are adjacent if the corresponding edges are adjacent in G (Definition 12).

Intuition: Line graph expands an N-node degree-d topology into a dN-node topology. The degree d remains the same, which is crucial since the degree is often limited by hardware constraints like the number of available ports. While the number of nodes grows by d-fold, the diameter of the topology only increases by one, which is also optimal for total-hop latency and all-to-all performance. In addition, the paths in the base topology are mapped into the expanded topology, allowing the communication schedule for the base to be expanded as well. Line graph expansion can be applied repeatedly to scale the topology and schedule to arbitrarily large sizes.

Figure 2 gives an example of the line graph expansion of the complete bipartite graph $K_{2,2}$. Any (shortest) path $w_0 \to w_1 \to \ldots \to w_n$ in $K_{2,2}$ can be mapped to a (shortest) path $w_{-1}w_0 \to w_0w_1 \to \ldots \to w_{n-1}w_n \to w_nw_{n+1}$ in $L(K_{2,2})$ from $w_{-1}w_0$ to w_nw_{n+1}, for any w_{-1}, w_{n+1} provided that $w_{-1}w_0 \neq w_nw_{n+1}$.

Given an allgather schedule A_G for G, we construct schedule $A_{L(G)}$ for $L(G)$. Pick any node $v'v$ in $L(G)$. It needs to broadcast its shard to every other node in $L(G)$. Pick any other node, say, uu'. For each element x of $v'v$'s shard, we want to send x to uu'. Since v broadcasts x to every other node in A_G, there is a path $v \to w_1 \to \ldots \to w_{n-1} \to u$ in G along which x is sent to u in A_G. Thus, the path $v'v \to vw_1 \to w_1w_2 \to \ldots \to w_{n-1}u \to uu'$ can be utilized to send x from $v'v$ to uu' in $L(G)$.

Definition 1 (Schedule of Line Graph)**.** *Given an allgather schedule A_G for topology G, let $A_{L(G)}$ be the schedule for line graph $L(G)$ containing:*

1. $((v'v, S), (v'v, vu), 1)$ *for each edge* $(v'v, vu) \in E_{L(G)}$ *with* $v'v \neq vu$. **[Insert the 1st comm step in $A_{L(G)}$.]**
2. $((v'v, C), (uw, ww'), t+1)$ *for each* $((v, C), (u, w), t) \in A_G$ *and* $v'v \neq ww'$. **[Adapt A_G to form $A_{L(G)}$.]**

At the 1st comm step, x is broadcasted by $v'v$ to every neighbor, including vw_1. Then, for every $((v, C), (w_i, w_{i+1}), t)$ in A_G with $x \in C$, there is $((v'v, C), (w_iw_{i+1}, w_{i+1}w_{i+2}), t+1)$ in $A_{L(G)}$ that takes x from w_iw_{i+1} to $w_{i+1}w_{i+2}$ and, eventually, to uu' ($v = w_0, u = w_n$). Since x and uu' are picked arbitrarily, $v'v$ broadcasts every element of its shard to all nodes in $L(G)$. Figure 2c shows an example of schedule construction.

As for the performance of $A_{L(G)}$, we leave the mathematical details in Appendix D.1. In practice, one can apply line graph expansion repeatedly to scale the topology and schedule indefinitely. The more optimal the base topology and schedule are, the more optimal the expanded topology and schedule will be. Figure 3 shows how the performance evolves as we continuously apply line graph expansion to several Moore and BW optimal base graphs. The total-hop latency always remains Moore optimal. The BW performance deviates from optimality T_B^* but remains a constant factor away asymptotically. A key observation in Figure 3 is that the larger the size of the base graph is, the closer the expanded schedule is to BW optimality. Line graph expansion is notable for its ability to construct indefinitely large-scale topologies without increasing degree d. The expansion also maintains low-hop, making it ideal for synthesizing all-to-all topologies as well.

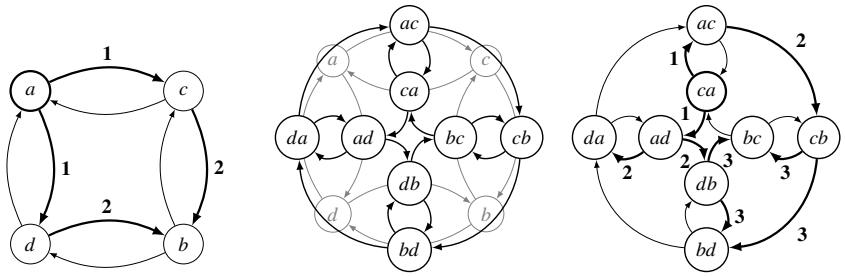
5.2 Degree Expansion

Intuition: While line graph expansion expands the number of nodes, degree expansion additionally expands the topology degree. Taking a base topology G, we make n copies of it and connect two nodes in different copies if they are adjacent in G. This process forms an expanded topology $G * n$, which multiplies both the number of nodes and the degree of G by n. Because the connections in $G * n$ are derived from G, similar to line graph expansion, we can map paths from G to $G * n$ to expand the communication schedule of G as well.

Figure 4 gives an example of expanding a 4-node unidirectional ring into an 8-node degree-2 topology (see formal definition of degree expanded topology in Definition 13). Based on the input schedule A_G for G, we construct a schedule A_{G*n} for $G * n$. For any data traveling along $v \to w^{(1)} \to \ldots \to w^{(m)} \to u$ in A_G, A_{G*n} has the data travel along $v_i \to w_i^{(1)} \to \ldots \to w_i^{(m)} \to u_j$ for all i, j. That is, data is transmitted within the i-th copy of G, except at the last step. With this construction, any node u_i has broadcasted the data to all other nodes except its own copies u_js. We add an additional comm step for u_j to collect the data from its in-neighbors (see Figure 4c).

Definition 2 (Degree Expanded Schedule)**.** *Given an allgather schedule A_G for G, construct A_{G*n} for $G * n$:*

1. *For all i, j including $i = j$ and for each $((v, C), (u, w), t) \in A_G$, add $((v_j, C), (u_j, w_i), t)$ to A_{G*n};*
2. *Divide shard S into equal-sized chunks $C_1, \ldots, C_{nd}$. Given $u_i, u_j \in V_{G*n}$ with $i \neq j$, add $((u_i, C_\alpha), (v_\alpha, u_j), t_{\max} + 1)$ to A_{G*n} for each $(v_1, u_j), \ldots, (v_{nd}, u_j) \in E_{G*n}$, where $t_{\max}$ is the max comm step in A_G.*



(a) $K_{2,2}$ $(N=4,d=2)$ (b) $L(K_{2,2})$ $(N=8,d=2)$ (c) Broadcast from ca

Figure 2: The complete bipartite topology $K_{2,2}$ with its line graph $L(K_{2,2})$. Figure (a) shows the base topology and broadcast paths from a to c,b,d in $A_{K_{2,2}}$ (see Figure 1). The number next to edge shows the comm step using the edge. Figure (b) shows the expanded topology. Observe that every edge in $K_{2,2}$ becomes a vertex in $L(K_{2,2})$, and two vertices are connected if the corresponding edges in $K_{2,2}$ have one's head node being the other's tail node. Figure (c) shows the broadcast paths of node ca, transformed from the broadcast paths of a in figure (a). At the 1st comm step, by step 1 of Def 1, ca broadcasts its shard to all its neighbors: $((ca,S),(ca,ac),1)$, $((ca,S),(ca,ad),1)$. The rest of the broadcast paths are transformed from $A_{K_{2,2}}$ by step 2 of Def 1, e.g. $((a,C_1),(c,b),2) \mapsto \{((ca,C_1),(cb,bc),3),((ca,C_1),(cb,bd),3)\}$. Each of the nodes bc and bd receives C_1,C_2 from its two in-neighbors, just like b does in $A_{K_{2,2}}$.

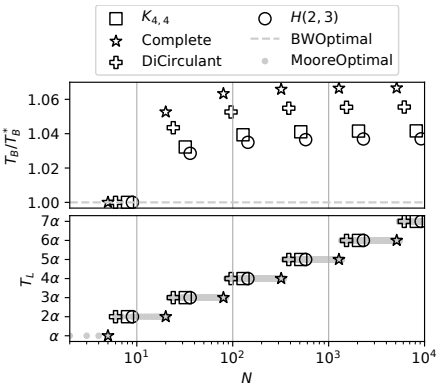
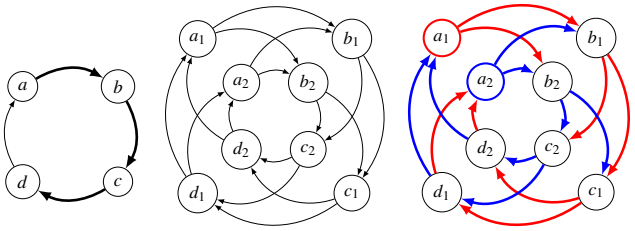


Figure 3: Line graph expansion on Moore and BW optimal degree-4 base graphs: complete bipartite graph $K_{4,4}$, complete graph, directed circulant graph, and Hamming graph $H(2,3)$. $T_B^* = \frac{M}{B} \cdot \frac{N-1}{N}$ is the optimal BW runtime.



(a) $G(N=4,d=1)$ (b) $G*2$ $(N=8,d=2)$ (c) Broadcasts from a_1,a_2

Figure 4: 4-node unidirectional ring and its degree expansion to $d=2$. Figure (a) shows the base topology and broadcast path from a to b,c,d. Figure (b) shows the expanded topology. Figure (c) shows the broadcast paths from a_1 and a_2 to other nodes marked in red and blue, respectively. For any $u \neq a$, the path from a_i to u_j stays in i until the very last step when it jumps to u_j, e.g., $a_1 \to b_1 \to c_2$ and $a_2 \to b_2 \to c_2 \to d_1$. For a_i to a_j, each in-neighbor of a_j sends an equal portion of a_i's shard to a_j in the end; for example, d_1 and d_2 each send half of a_1's shard to a_2 and half of a_2's shard to a_1. The red and blue broadcast paths are disjoint, resulting in BW optimality.

Unlike line graph expansion, degree expansion preserves BW optimality. This is because the expanded broadcast paths from copies of an original node are totally disjoint from each other (Figure 4c). However, degree expansion does not preserve Moore optimality. While line graph expansion does not change degree, degree expansion increases it, reducing the number of comm steps required for Moore optimality.

5.3 Cartesian Product Expansion

The Cartesian product of two graphs G_1,G_2 is an expanded graph $G_1 \Box G_2$ with size and degree equal to the product of G_1,G_2's sizes and the sum of their degrees, respectively.

Definition 3 (Cartesian Product)**.** *The Cartesian product digraph $G_1 \Box G_2$ of digraphs G_1 and G_2 has vertex set $V_{G_1} \times V_{G_2}$ with vertex $\mathbf{u} = (u_1,u_2)$ connected to $\mathbf{v} = (v_1,v_2)$ iff either $(u_1,v_1) \in E_{G_1}$ and $u_2 = v_2$; or $u_1 = v_1$ and $(u_2,v_2) \in E_{G_2}$.*

This definition generalizes to the Cartesian product of n digraphs: $G_1 \Box G_2 \Box \ldots \Box G_n$. When $G_1 = \ldots = G_n = G$, the product is denoted as Cartesian power $G^{\Box n}$. We use Cartesian power and product in our topology and schedule expansion.

Intuition: The Cartesian product $G_1 \Box G_2 \Box \ldots \Box G_n$ consists of n dimensions, with connections in dimension i identical to G_i. Taking the schedules of $G_1,G_2,\ldots,G_n$, we can balance the amount of traffic going through each dimension to achieve high BW performance. Cartesian product expansion greatly expands the set of topologies we construct by enabling the combination of existing topologies to form a new product topology with an efficient schedule.

Cartesian Power Expansion: Given a d-regular G and schedule A_G, we can construct a schedule $A_{G^{\Box n}}$ for $G^{\Box n}$, which is nd-regular and has $|V_G|^n$ nodes. This technique helps generate efficient topologies, including some well-known ones like hypercube and Hamming graph. We describe how to construct allgather schedules for Cartesian power graphs by using $\ell \times \ell$ torus (ℓ-ring$^{\Box 2}$) as an example. A typical allgather schedule on an $\ell \times \ell$ torus is to perform the ℓ-ring allgather along rings in one dimension first and then the other dimension, as in hierarchical ring allreduce [74]. Consider two schedules: $A^{(1)}$ performs allgather on vertical rings first and then horizontal ones; $A^{(2)}$ performs allgather in the opposite order. $A^{(1)},A^{(2)}$ use disjoint set of links at any comm step. Thus, we divide each data shard in $\ell \times \ell$ torus into two halves and let them be allgathered by $A^{(1)},A^{(2)}$ separately. The combined schedule, where $A^{(1)}$ and $A^{(2)}$ are performed in parallel, is BW-optimal, with a total-hop latency of $2T_L(A)$.

The above torus schedule has appeared in previous literature [62]. It can be generalized to generate schedules for Cartesian power of arbitrary topologies (see Appendix D.3).

Cartesian Product Expansion: One can also construct a schedule for the Cartesian product of distinct topologies. For example, an $a \times b \times c$ 3D torus is the Cartesian product of three rings with lengths a,b,c. Constructing this schedule requires BFB schedule generation technique, which we introduce in §6. If individual topologies have BW-optimal BFB schedules, as in the case of any torus, then the schedule generated for the Cartesian product is BW-optimal (Table 3).

Expansion Techniques	# of Nodes	Deg	Moore	BW	Perf
Line Graph Exp $L^n(G)$	d^nN	d	✓	×	Thm 7.1
Degree Exp $G*n$	nN	nd	×	✓	Thm 11
Cartesian Power $G^{\square n}$	N^n	nd	×	✓	Thm 12
Cartesian Prod $G_1\square\ldots\square G_n$	$\prod_i N_i$	$\sum_i d_i$	×	✓	Thm 13

Table 3: Summary of expansion techniques. The table shows the characteristics of the resulting topology and schedule after applying expansion techniques to an N-node degree-d base topology. "✓, ×" show whether the expansion preserves Moore/BW optimality. The last column refers to the theorems that give the exact performance of expanded schedules. Appendix D presents more formal definitions and analyses of expansion techniques.

Topology	T_L	T_B	$2(T_L+T_B)$	$D(G)$	All-to-All
$\Pi_{4,1024}$	5α	$1.332M/B$	323.5us	5	409.1us
$L^3(C(16,\{3,4\}))$	6α	$1.020M/B$	291.0us	6	403.5us
$L^2(\text{Diamond}^{\square 2})$	8α	$1.004M/B$	328.4us	8	446.6us
$L(\text{DBJMod}(2,4)^{\square 2})$	11α	$1.000M/B$	387.8us	9	529.9us
$(\text{UniRing}(1,4)\square\text{UniRing}(1,8))^{\square 2}$	20α	$0.999M/B$	567.6us	20	1174.4us
Theoretical Bound	**5α**	$\mathbf{0.999}M/B$	**267.6us**	**5**	**382.3us**

Table 4: Pareto-efficient topologies at $N=1024$, $d=4$. The $2(T_L+T_B)$ column shows the allreduce runtimes for $\alpha=10$us and $M/B=1$MB/100Gbps. We multiply T_L+T_B by 2 because allreduce is performed by combining reduce-scatter and allgather. The all-to-all time is computed via multi-commodity flow (Appendix A.5) with each node having 1MB of data to send (i.e., sending $1/N$ MB to each node). For comparison, the baselines Shifted Ring and Double Binary Tree (§8.2) have allreduce times of 20640us and 1434us, and all-to-all times of 10738us and 21475us, respectively. Table 9 shows the details of the base topologies.

5.4 Topology Finder

The goal of Topology Finder is to produce the best topologies and schedules for a target N and d. If we aim for asymptotic performance ($N\to\infty$ with fixed d), we want **the base topology to be as large as possible and the base schedule to be as optimal as possible** (Theorem 9). However, for a target N and d, only base topologies with certain sizes (e.g., divisors of N) and degrees can be expanded to the target. Thus, we keep a collection of known base topologies and their schedules (Table 9). These topologies and schedules are highly optimized and cover a wide range of N and d.

Given base topologies, we perform a bottom-up search for the combinations of expansion techniques to reach the target N and d. We iteratively apply expansions to candidates. At intermediate sizes, we prune candidates with inferior performance and keep the best ones for further expansion. Because each expansion multiplies the topology size (Table 3), the number of expansions that can be applied before the size gets too large—and hence the number of possible combinations—is limited, making the search feasible.

While we expand the topologies, proved theorems (Table 3) allow us to predict the performance of expanded topologies. This is vital for the search because it is intractable to construct schedules for every topology and compare their performance. Using a simple formula for prediction enables us to quickly compare different topologies and prune inferior ones. We keep a Pareto frontier of topologies for each given N and d. A topology is inferior to another only if it is worse in both total-hop latency and BW runtime. Ultimately, the search finds all Pareto-efficient topologies for the target N and d. Depending on the testbed, we may convert unidirectional topologies to bidirectional ones (Appendix A.6). Then, we determine the best-performing topology for a given workload.

Table 4 shows the result for $N=1024$ and $d=4$. From top to bottom, the Pareto frontier exhibits an increasing T_L and a decreasing T_B, with the top and bottom being Moore and BW optimal, respectively. On the all-to-all side, the diameters of the topologies also follow the same trend as T_L because of $T_L\geq\alpha\cdot D(G)$ (Theorem 3). Table 4 also shows the allreduce and all-to-all times calculated based on specific α, M, B. Notably, the line graph of circulant graph $L^3(C(16,\{3,4\}))$ has both the lowest allreduce and all-to-all times, within 9% and 6% of the theoretical bounds. Table 7 in appendix contains more results for $N=32,64,\ldots,1024$.

While low-hop/diameter indicates high all-to-all throughput, other metrics like the average distance between nodes [41, 42] also play a role. Thus, despite having a lower diameter, $\Pi_{4,1024}$ underperforms $L^3(C(16,\{3,4\}))$. Including other metrics makes the search process more complex and computationally expensive. In practice, T_L and $D(G)$ are feasible and accurate enough for predicting all-to-all throughput.

In DNN training experiments, we use one topology for the entire training due to the high reconfiguration latency of our target patch panel platform. We select the best option based on the distribution of collective sizes Ms, which depends on the communication strategy of the training framework [38]. With faster reconfiguration, one could change topology to optimize for different collective runs during training.

Our implementation runs under a minute for all $d=2,4,8,16$ and N up to 2000. While this can be sped up, we find it acceptable for now, given that the search is performed once for all Ns and ds, and results can be saved for future use.

6 Breadth-First-Broadcast (BFB) Schedule

We now present a scalable algorithm for generating schedules for *generative topologies*, which are directly borrowed from graph theory, as well as for topologies obtained through Cartesian Product expansion—the only expansion technique that does not yield a schedule. State-of-the-art schedule generations (e.g., Blink [76], SCCL [10], and TACCL [65]) can scale only to a modest number of nodes because they involve NP-hard optimization. To ensure polynomial-time generation, we impose a *breadth-first* broadcast order from each node such that (1) data always travels along the shortest paths between source and destination nodes; (2) the schedule is structured as a series of comm steps, where each comm step is responsible for eagerly transmitting data to a set of nodes that is one additional hop away. Our BFB schedule generation technique does not guarantee optimality in an arbitrary topology, given these constraints prohibit the use of longer paths or delayed (non-eager) transmissions along paths, but these constraints enable polynomial-time generation.

Despite these constraints, BFB schedules guarantee the following: (1) The schedules have the lowest total-hop latency as all data is eagerly communicated over the shortest paths.

(2) For Cartesian product topologies, BFB schedule generation yields a BW-optimal solution if the underlying product components admit BW-optimal BFB schedules (thus yielding optimal schedules for networks such as torus with arbitrary dimensions). (3) BFB schedules are also provably BW-optimal for many generative topologies with certain symmetries.

6.1 BFB Schedule Generation Linear Program

Intuition: Allgather is a simultaneous broadcast from every node in the topology. A BFB allgather schedule, as the name suggests, performs a *breadth-first* broadcast from every node. At each comm step, a node typically has the option to receive a chunk from multiple in-neighbors on the previous breadth-first frontier. To optimize BW performance, BFB uses a linear program to balance the traffic across the ingress links.

At each comm step t, a BFB schedule requires that for every node v, all nodes at a distance t from v, i.e., $N_t^+(v)$, receive v's data shard within the comm step. To achieve this, all nodes at distance $t-1$, i.e. $N_{t-1}^+(v)$, need to collectively multicast the data shard to nodes $N_t^+(v)$ in comm step t.

Given any v and $u \in N_t^+(v)$, u may have multiple in-neighbor ws in $N_{t-1}^+(v)$. All of them can provide v's data shard because they have received it in comm step $t-1$. Since the BW runtime of a comm step equals the transmission time of the most congested link, a question is **how to allocate the amount of data u receives from each w to balance the workload on links?** Figure 5 shows an example. Here, u_1 needs to get v_1's shard from w_1, w_2 and v_2's shard from w_2. The solution is simple: since u_1 can only get v_2's shard from w_2, we let w_1 send v_1's shard and w_2 send v_2's shard, achieving a perfectly balanced workload. For u_2, it is more complicated. We formulate such a problem as a linear program:

$$\begin{aligned} \text{minimize} \quad & U_{u,t} \\ \text{subject to} \quad & \sum_v x_{v,(w,u),t} \le U_{u,t}, && \forall w \in N^-(u) = N_1^-(u) \\ & \sum_w x_{v,(w,u),t} = 1, && \forall v \in N_t^-(u) \\ & 0 \le x_{v,(w,u),t} \le 1. && \forall w, v \end{aligned} \tag{1}$$

$x_{v,(w,u),t}$ is the proportion of v's shard that is sent from w to u and is defined for every v, w such that $w \in N^-(u)$ and $d(v,u) = d(v,w)+1 = t$. $U_{u,t}$ is the max workload among links to u, i.e., $(w_1,u_2),(w_2,u_2),(w_3,u_2)$ in the case of u_2. Minimizing $U_{u,t}$ is equivalent to minimizing $\frac{M/N}{B/d} \cdot U_{u,t}$, the max transmission time among links to u at comm step t. The 1st and 2nd constraints ensure correct max workload and u receiving all data shards, respectively. Appendix E gives the specific LP for u_2, and the solution is shown in blue in Figure 5. The workload is also balanced with each link sending $2/3$ shard and hence BW runtime $\frac{M/N}{B/d} \cdot \frac{2}{3}$.

SCCL [10] and TACCL [65] use NP-hard optimizations with discrete variables used to ensure each chunk is received before being sent. In contrast, we do not need discrete variables. A key observation from Figure 5 is that because w_1, w_2, w_3 all receive the entire shard of v_1 at comm step $t-1$,

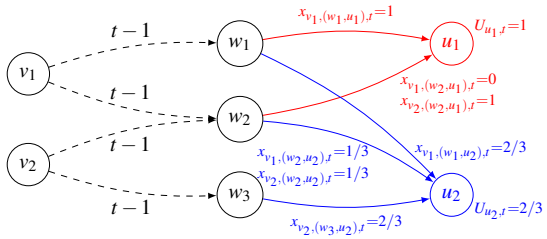


Figure 5: Example of BFB allgather schedule at comm step t. Here, $w_1, w_2 \in N_{t-1}^+(v_1)$ and $w_2, w_3 \in N_{t-1}^+(v_2)$. u_1, u_2 are at distance t from both v_1, v_2, so they both need to receive the data shards of v_1, v_2 in comm step t. Note that u_1 cannot get v_2's shard from w_3 because w_3 is not an in-neighbor of u_1. The figure also shows the solutions to LPs (1). The red and blue are independent LPs optimizing $U_{u_1,t}$ and $U_{u_2,t}$ respectively.

the $x_{v_1,(w_1,u_2),t} = 2/3$ and $x_{v_1,(w_2,u_2),t} = 1/3$ in the solution can be any portions of the data shard, as long as their union is the entire shard. Assuming $[0,1]$ is the entire shard of v_1, no matter the $2/3$ sent by w_1 to u_2 is $[0, \frac{2}{3}]$ or $[\frac{1}{3}, 1]$, the $1/3$ sent by w_2 can simply be $[\frac{2}{3}, 1]$ or $[0, \frac{1}{3}]$ accordingly. Thus, **we only need to decide the amount of data sent on each link, which are continuous variables**, enabling polynomial-time schedule generation. To obtain a complete schedule, one needs to solve an LP (1) for each $u \in V_G$ and $t \in \{1, \ldots, D(G)\}$. The BW runtime of the generated schedule is

$$T_B = \frac{M/N}{B/d} \sum_{t=1}^{D(G)} \max_{u \in V_G} U_{u,t}. \tag{2}$$

One could create an LP incorporating all $U_{u,t}$s and minimize T_B (2) *"globally"*. However, the result is equivalent to individually solving small LPs (1) for each u and t. This is because the LPs are independent of each other, e.g., the decisions made to minimize $U_{u_2,t}$ in Figure 5 do not affect $U_{u_1,t}$, and vice versa. The advantage of small LPs is that they can be solved in parallel. Due to the breadth-first nature of BFB, data always follows the shortest paths between source and destination. Thus, the number of comm steps of the BFB schedule always equals the graph diameter, i.e., $T_L = \alpha \cdot D(G)$, the lowest possible T_L given G.

Appendix E analyzes the BFB schedule and includes modifications to generate **discrete chunked schedules** (§E.2) and schedules for **heterogeneous link bandwidths** (§E.3). Corollary 1.1 implies how to generate **reduce-scatter** schedules.

6.2 Generative Topologies

Generative topologies, unlike synthesized ones, are large graphs directly borrowed from graph theory. They are too large for manual or NP-hard schedule generation. Thus, we use the BFB linear program to generate schedules. Since a BFB schedule always has the lowest T_L for a topology, if it is also BW-optimal, then it is *the optimal schedule* for that topology. Generative topologies often have symmetries that allow us to prove optimality mathematically. Their low diameters are also ideal for all-to-all throughput.

Torus is a widely used topology in parallel computing systems. Our torus schedule generated by BFB is theoretically

optimal and represents a significant improvement over traditional torus schedules [62]. Given a $d_1 \times d_2 \times \ldots \times d_n$ torus, the traditional schedule, which performs parallel ring collectives on dimensions, only works (or is efficient) when dimensions are equal, i.e., $d_1 = d_2 = \ldots = d_n$, and has $T_L = \sum_i (d_i - 1)\alpha$. BFB torus schedule, however, is BW-optimal with any d_is and $T_L = \sum_i \lfloor d_i/2 \rfloor \alpha$. The BW optimality is due to torus being the *Cartesian product* of rings, each of which has a BW-optimal BFB schedule. BFB torus opens up many more construction possibilities since d_is can be any combination.

Generalized Kautz Graph (§F.2) and **Circulant Graph** (§F.4) are a pair of versatile graphs in our toolbox. The former can be constructed for any N and d, while the latter can be constructed for any N and even-value d. Furthermore, the BFB schedule of the former is at most one α away from Moore optimality, making it the topology with the lowest T_L, while the latter always has a BW-optimal BFB schedule. Thus, they can fill gaps in N and d that expansion techniques fail to cover (e.g., prime N) or provide good candidates.

Besides the aforementioned ones, the following graphs also have optimal schedules by BFB. **Distance-Regular Graph** (§F.3) is a family of large highly-symmetric graphs that are both BW-optimal and low-hop at the same time. The **Twisted Torus** [14] used by TPU v4 [29] is also computationally verified to be BW-optimal for at least $N \leq 10^4$. A **BFB Ring Schedule** with half the T_L of traditional one is shown in §F.1.

7 Schedule Compilation

We implemented two compilers for lowering communication schedules to both GPU and CPU clusters, given the significance of collective communication for both ML and HPC workloads. We lowered over 1K schedules for various topologies and configurations. The compilers enable us to evaluate the performance of our topologies and schedules on hardware and to validate our mathematical model.

For GPUs, our compiler lowers a mathematically defined schedule to an XML file that can be executed by the MSCCL runtime [49]. MSCCL is an open-source collective communication library that extends NCCL [56] with an interpreter providing the ability to program custom schedules. Communication schedules are defined in XML as instructions (send/receive/reduce/copy) for each GPU threadblock. Our compiler also performs certain optimizations, such as consolidating non-contiguous sends using a scratch buffer and evenly distributing the computational workload across threadblocks.

For CPU-based supercomputers, we use Intel oneCCL [26] + libfabric [39] to execute schedules. We extended oneCCL with an interpreter that executes XMLs. The mathematical schedules are lowered into instructions (send/recv/reduce/copy/sync) for CPUs in an XML file and then executed.

8 Evaluation

We present performance evaluation results on a 12-node direct-connect optical GPU testbed and a supercomputing CPU torus cluster with up to 54 nodes. We also present analytical and simulation results at larger scales.

Collective Communication: On the 12-node testbed, our topologies consistently outperform baselines in allreduce, reduce-scatter, and allgather (§8.3, Fig 6, Fig 12). Analytical model shows order-of-magnitude improvements in allreduce and all-to-all performance at larger scales (Fig 7).

DNN Training: In data-parallel training experiments, our topologies achieve the best performance for both small models and GPT-2 [59] (§8.4, Fig 8). In simulated large-scale training, our topologies demonstrate order-of-magnitude improvements in all-to-all involved expert-parallel training (Fig 9).

Schedule Generation: While generating optimal schedules, BFB is orders of magnitude faster and more scalable than SCCL [10] and TACCL [65] (§8.5.1, Tab 6 & Fig 10). On supercomputing torus clusters, BFB schedules outperform traditional scheduling [62], SCCL, and TACCL (§8.5.2, Fig 11).

Finally, we also conducted experiments on our testbed to compare BFB against widely adopted solutions for switch networks (e.g., NCCL and recursive halving & doubling) (§A.1) and to validate the α-β cost model (§A.2).

8.1 Direct-Connect Optical Testbed

Our testbed consists of 12 servers, each with an NVIDIA A100-PCIe-40GB GPU [55] and an HPE Ethernet Adapter, configured as 4x25Gbps breakout interfaces. The NICs are directly connected via a Telescent optical patch panel [70]. Our testbed can realize topologies by reconfiguring the patch panel. We limit our evaluation to bidirectional topologies due to limitations in our testbed (discussed in Appendix A.6).

8.2 Experiment Setup

Baselines: We evaluate against the two baselines at $d = 4$: (1) *ShiftedRing*, which improves upon NCCL ring [56], is used by TopoOpt [77] for data-parallel training. The topology is a superposition of two bidirectional rings, each allreducing half of the data. (2) *Double Binary Tree* (DBT), implemented in NCCL [27], uses the topology and schedule from [63].

Methodology: We use the MSCCL runtime [49] to evaluate the topologies and schedules. We sweep through runtime parameters, such as the protocol (Simple or LL), number of channels (1, 2, 4, or 8), degrees of pipelining for the DBT baseline, etc., and choose the best-performing schedule for each data size. For DNN training, we run our schedules in PyTorch through MSCCL as the backend.

8.3 Collective Communication Evaluation

Figure 6 shows allreduce results for varying topology sizes N and data sizes M. Table 5 shows the topologies generated by our topology finder (§5.4). We also add *ShiftedBFBRing*, which is ShiftedRing topology but with our BFB generated schedule. We observe that in the small data regime ($M =$ 1KB), our topology beats ShiftedRing by a significant margin ($\sim$75% at $N = 12$) and also outperforms DBT ($\sim$20% at $N = 8, 10, 12$). Our ShiftedBFBRing beats ShiftedRing ($\sim$40%

N	Topology	T_L
5	Complete Graph: K_5	2α
6	Degree Expansion of Complete graph: $K_3 * 2$	4α
7	Circulant Graph: $C(7,\{2,3\})$	4α
8	Complete Bipartite Graph: $K_{4,4}$	4α
9	Hamming Graph: $H(2,3)$	4α
10	Degree Expansion of BFB augmented Bidirectional Ring: BiRing$(2,5) * 2$	4α
11	Circulant Graph: $C(11,\{2,3\})$	4α
12	Circulant Graph: $C(12,\{2,3\})$	4α

Table 5: OurBestTopo at $d=4$ generated by topology finder (§5.4). All topologies listed above are BW-optimal.

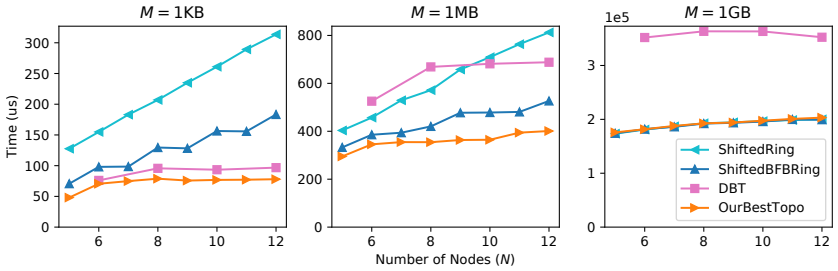


Figure 6: Allreduce experiment results on testbed at $M=1\text{KB}, 1\text{MB}, 1\text{GB}$. "OurBestTopo" topologies are listed in Table 5. Reduce-scatter and allgather results are in Appendix Figure 12.

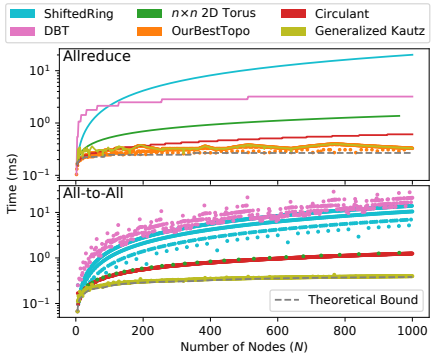
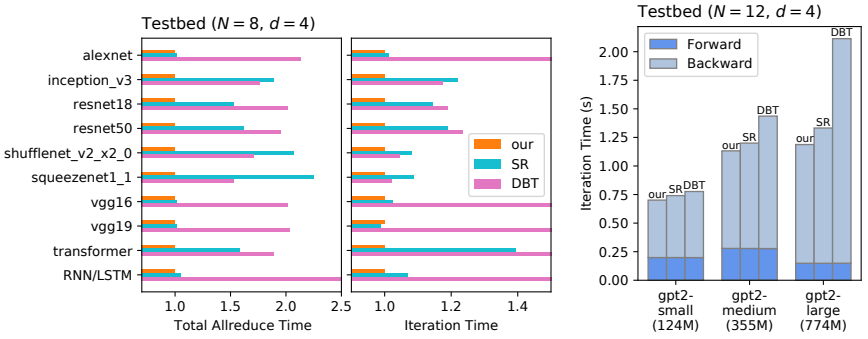


Figure 7: Comparing theoretical allreduce and all-to-all runtimes analytically at large N for $d=4$, $\alpha=10$us, and $M/B=1\text{MB}/100\text{Gbps}$. The all-to-all times are computed via multi-commodity flow (Appendix A.5) with each node having 1MB of data to send (i.e., sending $1/N$ MB to each node).



(a) Small Model testbed training results (normalized by our $K_{4,4}$)

(b) GPT2 testbed training results

Figure 8: Testbed data-parallel training results with different topologies. We compare our topologies (Table 5) against ShiftedRings (SR) and double binary trees (DBT) at 8- and 12-node scale. (a) shows results of training small models on 8 A100 GPUs of our testbed, all using batch size 64. The total allreduce time is the sum of the allreduce times for all layers in the model. (b) shows results of training GPT-2 with 12 A100 GPUs. The per-GPU batch sizes are selected to max out the 40GB GPU memory, with the small, medium, and large models having per-GPU batch sizes of 8, 4, and 1, respectively.

at $N=12$) despite using the same topology. At small data sizes, the runtime is dominated by total-hop latency T_L, and hence, we can significantly outperform ShiftedRing, which has linear instead of logarithmic T_L growth with respect to N.

In the large data regime ($M=1\text{GB}$), our topology beats DBT by a significant margin (~50% lower at $N=8,10,12$) and matches the performance of ShiftedRing. At large data sizes, the runtime is dominated by BW runtime T_B. Since the ShiftedRing is BW-optimal, we can only match its performance. Due to the influence of both total-hop latency and BW runtime at intermediate data sizes ($M=1\text{MB}$), our topology outperforms ShiftedRing (~50% at $N=12$) and DBT (~45% at $N=8,10,12$) in this regime. Our ShiftedBFBRing also outperforms ShiftedRing (~35% at $N=12$). Note that although our gains over ShiftedRing diminish as M grows, future increases in hardware bandwidth will enhance gains at large M due to T_L playing a more significant role. Appendix Figure 12 shows the reduce-scatter and allgather results, which demonstrate trends and conclusions similar to those in Figure 6.

Figure 7 shows the allreduce and all-to-all runtime comparison for large N based on our analytical model. Topologies generated by our topology finder perform orders of magnitude faster in both allreduce and all-to-all. In allreduce, our best topologies outperform ShiftedRing and DBT by 56× and 10×, respectively, near $N=1000$, due to the former's linear growth in T_L and the latter's poor BW performance. When compared against 2D torus, our topologies also achieve 4× better allreduce performance near $N=1000$ (see §A.3 for a detailed analysis of our topologies at large N for different $\alpha, M/B$). As for all-to-all, we compare baseline topologies against our lowest-diameter topology, generalized Kautz, and our highest-diameter topology, circulant, from our Pareto-frontier for any N and d. These two represent our best and worst all-to-all topologies, respectively, while also serving as the worst and best BW-efficient allreduce topologies. Nevertheless, circulant still outperforms all baselines in all-to-all: 9× and 14× better than ShiftdRing and DBT, respectively, on average from $N=900$ to 1000. It is barely matched by the 2D torus, which is limited to the square number Ns. Our lowest-diameter topology, generalized Kautz, outperforms ShiftedRing and DBT by 28× and 42×, respectively, and is within 5.2% of the theoretical bound from $N=900$ to 1000.

8.4 DNN Training Evaluation

We compare our topologies against ShiftedRings and double binary trees in DNN training. On our small-scale testbed, we demonstrate improvements in data-parallel training across various small models and also GPT-2 [59]. For full-scale LLM training involving up to 1024 nodes and all-to-all com-

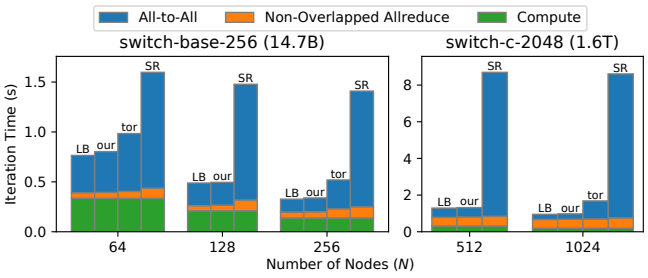


Figure 9: Simulated expert-parallel training of Switch Transformers across various topologies of different sizes. The simulation is conducted assuming $\alpha = 10$us, $B = 100$Gbps. All-to-all time is computed via multi-commodity flow (Appendix A.5). We detailed our setup in Appendix A.4.

munications, we simulate expert-parallel training of a Mixture of Experts (MoE) model to show our improvements at scale.

Testbed Training: We run PyTorch Distributed Data Parallel (DDP) [38] training experiments on our testbed. Figure 8 shows the results of training both small DNN models and GPT-2. We compare our topologies (from Table 5) against ShiftedRings and DBT. In training small models (Figure 8a), our topology improves total allreduce time by 30% and 50% on average against ShiftedRing and DBT, respectively. With optimizations such as compute-communication overlap, our topology still secures a 10% and 25% average improvement in iteration time over the baselines. In GPT-2 training (Figure 8b), despite the limited scale of our testbed, our topology enhances iteration time by 7% and 25% on average compared to ShiftedRing and DBT, respectively.

Large-Scale Simulation: While improvements over ShiftedRing and DBT have been shown in testbed training, full-scale LLM training is performed on much larger clusters. In Figure 9, we simulate expert-parallel training of Switch Transformers [19] on a much larger scale with parameter sizes up to 1.6 trillion. We collect execution timestamps from one A100 GPU to derive the compute times for each layer. Communication times are then added to simulate training, ensuring compute-communication overlap/dependency. Appendix §A.4 provides further details of the simulation.

Expert-parallel training involves not only data-parallel allreduce for non-expert layers but also all-to-all communications to transfer tokens to and from the routed experts, which are in the critical path of compute [19, 34, 36, 60]. In Figure 9, we break down the iteration time into compute time, non-overlapped allreduce time, and all-to-all time for a better understanding of performance. As previously analyzed in Figure 7, ShiftedRings (SR) exhibit all-to-all performance that is order-of-magnitude worse than our topologies. At 256-node training of 14.7B MoE model, ShiftedRing has 8× greater total all-to-all time, resulting in 4× longer iteration time compared to our topology. We also include 2D torus (tor) for comparison due to its relatively better all-to-all performance. However, it still has all-to-all and iteration times that are 2× and 1.5× greater, respectively, than our topology. The disparity is even larger at 1024-node training of 1.6T MoE model, where ShiftedRing and 2D torus show all-to-all times that are 27× and 3.3× greater, and iteration times that are 9× and 1.7× longer, respectively. At this scale, ShiftedRing and 2D torus spend 91% and 58% of iteration time on all-to-all communications, while our topology only spends 30%. We omit DBT in Figure 9 due to its significantly worse performance (∼2× of ShiftedRing). Due to high performance in both allreduce and all-to-all, our topologies consistently remain within 5% of the theoretical lower bound (LB) for iteration time.

Since large models involve large allreduce sizes and both torus and ShiftedRing are BW-optimal, the allreduce performance is similar across these topologies. For a broader spectrum of all-to-all efficient low-hop topologies like expander graphs, the lack of efficient allreduce schedules prior to our work has prevented their use in allreduce-involved training.

8.5 BFB Schedule Evaluation

We evaluate schedule generation from two aspects: schedule generation runtime and the performance of generated schedules. In §8.5.1, we compare BFB with state-of-the-art schedule generations: SCCL [10] and TACCL [65], in both generation runtime and theoretical schedule performance. In §8.5.2, we compare, on supercomputing torus clusters, the performance of torus schedules generated by BFB, traditional torus scheduling [62], SCCL, and TACCL.

8.5.1 Schedule Generation

In schedule generation, SCCL and TACCL are the closest in spirit to BFB schedule generation. Table 6 shows the runtime comparison between SCCL, TACCL, and BFB when generating allgather schedules for hypercube and 2D torus. Both SCCL and TACCL use NP-hard optimization to generate schedules. SCCL, which uses an SMT solver, fails to generate a schedule within 10^4 seconds beyond $N = 30$. TACCL formulates the scheduling problem as a mixed integer linear program (MILP). It sets an 1800s time limit for its MILP solver, after which it will return the best solution found up to that point. However, for larger topologies, TACCL's solver fails to find a solution within the time limit, resulting in an error. In comparison, BFB schedule generation is faster by orders of magnitude due to its polynomial-time generation.

In terms of theoretical schedule performance, Figure 10 compares the total-hop latency and BW runtime of generated schedules. Given a topology, SCCL and TACCL need to perform a sweep across parameters such as the number of chunks and symmetry. They have to generate schedules for different parameter sets to identify the high-performance ones, unlike BFB, which requires no parameter. In Figure 10, the schedules of SCCL and BFB can both achieve exact optimality, but TACCL's have significantly worse performance, especially at large Ns. SCCL is uniquely capable of generating all Pareto-efficient schedules. However, due to the prohibitive runtime of parameter sweep, SCCL can only do so for very small Ns.

N	SCCL				TACCL w/o Symmetry				TACCL w/ Symmetry				BFB
	$c=1$	$c=2$	$c=3$	$c=4$	$c=1$	$c=2$	$c=3$	$c=4$	$c=1$	$c=2$	$c=3$	$c=4$	
Hypercube													
4	0.59	0.64	0.68	0.72	0.89	0.50	0.83	0.75	0.62	0.51	0.71	0.60	<0.01
8	0.86	1.22	1.86	2.48	96.9	807	63.2	1800	7.97	645	7.39	1801	<0.01
16	21.4	48.4	130	573	1801	1801	1801	1802	1801	n/a	n/a	n/a	<0.01
32	$>10^4$	$>10^4$	$>10^4$	$>10^4$	1802	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0.03
64	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0.17
1024	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	52.7
2D Torus ($n \times n$)													
4	0.61	0.63	0.67	0.76	0.68	0.50	0.82	0.72	0.45	0.51	0.76	0.64	<0.01
9	1.00	1.51	2.22	3.44	1801	189	67.8	262	88.6	71.1	67.8	105	<0.01
16	17.5	60	131	603	1801	1801	1801	1802	1801	1801	1801	n/a	<0.01
25	3286	5641	$>10^4$	$>10^4$	1802	1802	1803	n/a	1802	n/a	n/a	n/a	0.01
36	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0.03
2500	$>10^4$	$>10^4$	$>10^4$	$>10^4$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	61.1

Table 6: Comparing allgather schedule generation runtimes (in seconds) of SCCL, TACCL, and BFB. The setup of SCCL is to generate schedules with the least number of comm steps. Both SCCL and TACCL were run with chunks=1,2,3,4 (number of chunks per shard), and TACCL was run w/ and w/o manually set topology symmetry. "n/a" indicates where TACCL reports an error due to failure to generate a solution within its 1800s time limit for MILP solver.

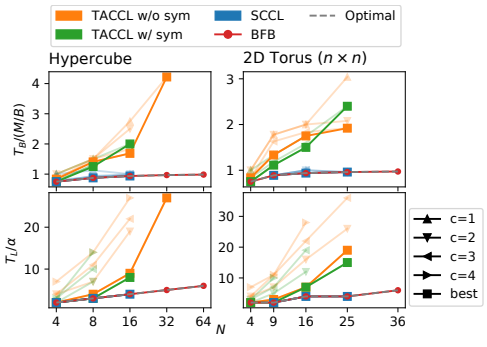


Figure 10: Comparing theoretical performances of schedules from Table 6. We show both T_L and T_B of the schedules, along with the theoretical optimal. For SCCL and TACCL, the solid lines show the best results from parameter sweeps. The inferior ones are dimmed.

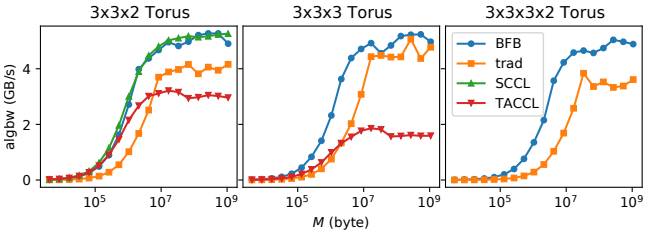


Figure 11: Comparing allreduce performances of torus schedules generated by BFB, traditional torus scheduling [62], SCCL, and TACCL on Frontera [69] supercomputer. The y-axis is algorithmic bandwidth (algbw), computed as M divided by end-to-end runtime. SCCL fails to generate a schedule for 3×3×3 and 3×3×3×2 tori, and TACCL fails to generate a schedule for 3×3×3×2 torus within the time limits.

8.5.2 Supercomputing Allreduce Experiments

In the supercomputing setting, we run torus schedules generated by BFB, traditional torus scheduling [62], SCCL, and TACCL on Frontera [69] supercomputer at the Texas Advanced Computing Center (TACC) [71]. The cluster consists of 396 nodes in a 6D torus direct-connect topology. Each node is equipped with an Intel Xeon Platinum 8280 CPU and a Rockport NC1225 network card, capable of delivering 25 Gbps per link, with degree 12. However, the total BW of a single node may be bottlenecked by the 100 Gbps host BW of PCIe Gen3 x16. Finally, the schedules are lowered and run using Intel oneCCL [26] + libfabric [39].

We run schedules on two types of sub-torus within the cluster: equal-dimension (3×3×3) and unequal-dimension (3×3×2 & 3×3×3×2). As shown in Figure 11, BFB schedules achieve the highest performance in all settings. As mentioned in §6.2, the traditional torus schedule can only achieve high BW performance in tori with equal dimensions. At large M, it matches BFB's performance in 3×3×3 torus but significantly underperforms in 3×3×2 and 3×3×3×2, where BFB has 29% and 42% higher algbw on average for $M \geq 100$MB. At small to intermediate M (< 100MB), BFB outperforms traditional schedules by 3.1× on average in all settings due to its 2× lower in total-hop latency and higher BW performance.

As for SCCL and TACCL, we adhere to the same time limits and parameter sweeps as in §8.5.1 and select the best result at each M from all parameter sets. In 3×3×2 torus, SCCL is able to generate an optimal schedule, matching BFB's performance across all M. However, it fails to generate a schedule within 10^4 seconds for other larger tori. TACCL can only generate schedules in 3×3×2 and 3×3×3, and its schedules underperform BFB's by a large margin. One additional observation is that the algbw of BFB at large M hardly changes from 18-node (3×3×2) to 54-node (3×3×3×2) torus. This can be explained by the fact that BFB schedules have theoretically achieved allreduce BW optimality ($\frac{2M}{B} \cdot \frac{N-1}{N}$), which remains nearly constant as N increases.

9 Concluding Remarks

Collective communications are critical to both ML training and HPC workloads. Current solutions often rely solely on existing topologies and schedules, resulting in high total-hop latency, bandwidth inefficiency, or low all-to-all throughput. We presented a general, highly scalable, and automated algorithmic framework for optimizing topology and schedule generation for collectives by leveraging scalable graph-theoretic approaches. Our evaluation demonstrates significant performance gains across multiple testbeds and large-scale simulations in both standalone collective communications and end-to-end ML training.

10 Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under Contract No.HR001120C0089. This work was also supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] BALLANI, H., COSTA, P., BEHRENDT, R., CLETHEROE, D., HALLER, I., JOZWIK, K., KARINOU, F., LANGE, S., SHI, K., THOMSEN, B., AND WILLIAMS, H. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 782–797.

[2] BANG, S., DUBICKAS, A., KOOLEN, J., AND MOULTON, V. There are only finitely many distance-regular graphs of fixed valency greater than two. *Advances in Mathematics 269* (2015), 1–55.

[3] BARNETT, M., SHULER, L., VAN DE GEIJN, R., GUPTA, S., PAYNE, D. G., AND WATTS, J. Interprocessor collective communication library (intercom). In *Proceedings of IEEE Scalable High Performance Computing Conference* (1994), IEEE, pp. 357–364.

[4] BASU, P., ZHAO, L., FANTL, J., PAL, S., KRISHNAMURTHY, A., AND KHOURY, J. Efficient all-to-all collective communication schedules for direct-connect topologies. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2024), HPDC '24, Association for Computing Machinery, p. 28–41.

[5] BERMOND, J.-C., HOMOBONO, N., AND PEYRAT, C. Connectivity of kautz networks. *Discrete Math. 114*, 1-3 (apr 1993), 51–62.

[6] BESTA, M., AND HOEFLER, T. Slim Fly: A cost effective low-diameter network topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), SC '14, IEEE Press, p. 348–359.

[7] BOESCH, F., AND WANG, J.-F. Reliable circulant networks with minimum transmission delay. *IEEE Transactions on Circuits and Systems 32*, 12 (1985), 1286–1291.

[8] BOKHARI, S. H., AND BERRYMAN, H. Complete exchange on a circuit switched mesh. In *Proceedings Scalable High Performance Computing Conference SHPCC-92.* (1992), IEEE, pp. 300–306.

[9] Broadcom P2200G - 2x200GbE PCIe NIC. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/200gb-nic-ocp/p2200g.

[10] CAI, Z., LIU, Z., MALEKI, S., MUSUVATHI, M., MYTKOWICZ, T., NELSON, J., AND SAARIKIVI, O. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2021), PPoPP '21, Association for Computing Machinery, p. 62–75.

[11] CHAN, E., HEIMLICH, M., PURKAYASTHA, A., AND VAN DE GEIJN, R. Collective communication: Theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper. 19*, 13 (sep 2007), 1749–1783.

[12] CHAN, E., VAN DE GEIJN, R., GROPP, W., AND THAKUR, R. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (2006), pp. 2–11.

[13] CHO, M., FINKLER, U., AND KUNG, D. BlueConnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *Proceedings of the 2nd SysML Conference* (2019).

[14] CÁMARA, J. M., MORETÓ, M., VALLEJO, E., BEIVIDE, R., MIGUEL-ALONSO, J., MARTÍNEZ, C., AND NAVARIDAS, J. Twisted torus topologies for enhanced interconnection networks. *IEEE Transactions on Parallel and Distributed Systems 21*, 12 (2010), 1765–1778.

[15] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., ET AL. Large scale distributed deep networks. *Advances in neural information processing systems 25* (2012).

[16] DistanceRegular.org. https://www.math.mun.ca/distanceregular/.

[17] ESFAHANIAN, A.-H., NI, L., AND SAGAN, B. The twisted n-cube with application to multiprocessing. *IEEE Transactions on Computers 40*, 1 (1991), 88–93.

[18] FAIZIAN, P., MOLLAH, M. A., YUAN, X., ALZAID, Z., PAKIN, S., AND LANG, M. Random regular graph and generalized de bruijn graph with k-shortest path routing. *IEEE Transactions on Parallel and Distributed Systems 29*, 1 (2017), 144–155.

[19] FEDUS, W., ZOPH, B., AND SHAZEER, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research 23*, 120 (2022), 1–39.

[20] GIBIANSKY, A. Bringing hpc techniques to deep learning. *Baidu Research, Tech. Rep.* (2017).

[21] GRINER, C., ZERWAS, J., BLENK, A., GHOBADI, M., SCHMID, S., AND AVIN, C. Cerberus: The power of choices in datacenter topology design - a throughput perspective. *Proc. ACM Meas. Anal. Comput. Syst. 5*, 3 (dec 2021).

[22] HARARY, F., AND NORMAN, R. Z. Some properties of line digraphs. *Rendiconti del Circolo matematico di Palermo 9*, 2 (1960), 161–168.

[23] HO, C.-T., AND JOHNSSON, S. L. Distributed routing algorithms for broadcasting and personalized communication in hypercubes. In *ICPP* (1986), pp. 640–648.

[24] HOCKNEY, R. W. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel computing 20*, 3 (1994), 389–398.

[25] IMASE, AND ITOH. A design for directed graphs with minimum diameter. *IEEE Transactions on Computers C-32*, 8 (1983), 782–784.

[26] INTEL. oneAPI Collective Communications Library (oneCCL). https://github.com/oneapi-src/oneCCL.

[27] JEAUGEY, S. Massively scale your deep learning training with nccl 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/, 2019.

[28] JIA, Z., ZAHARIA, M., AND AIKEN, A. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems 1* (2019), 1–13.

[29] JOUPPI, N., KURIAN, G., LI, S., MA, P., NAGARAJAN, R., NAI, L., PATIL, N., SUBRAMANIAN, S., SWING, A., TOWLES, B., YOUNG, C., ZHOU, X., ZHOU, Z., AND PATTERSON, D. A. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (2023).

[30] KHANI, M., GHOBADI, M., ALIZADEH, M., ZHU, Z., GLICK, M., BERGMAN, K., VAHDAT, A., KLENK, B., AND EBRAHIMI, E. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (New York, NY, USA, 2021), SIGCOMM '21, Association for Computing Machinery, p. 657–675.

[31] KIM, J., DALLY, W. J., SCOTT, S., AND ABTS, D. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture* (2008), pp. 77–88.

[32] KIM, J., GHAYOORI, A., AND SRIKANT, R. All-to-all communication in random regular directed graphs. *IEEE Transactions on Network Science and Engineering 1*, 1 (2014), 43–52.

[33] LAKHOTIA, K., BESTA, M., MONROE, L., ISHAM, K., IFF, P., HOEFLER, T., AND PETRINI, F. PolarFly: A cost-effective and flexible low-diameter topology. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2022), SC '22, IEEE Press.

[34] LEPIKHIN, D., LEE, H., XU, Y., CHEN, D., FIRAT, O., HUANG, Y., KRIKUN, M., SHAZEER, N., AND CHEN, Z. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations* (2021).

[35] LI, A., SONG, S. L., CHEN, J., LI, J., LIU, X., TALLENT, N. R., AND BARKER, K. J. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems 31*, 1 (2019), 94–110.

[36] LI, J., JIANG, Y., ZHU, Y., WANG, C., AND XU, H. Accelerating distributed MoE training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 945–959.

[37] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on operating systems design and implementation (OSDI 14)* (2014), pp. 583–598.

[38] LI, S., ZHAO, Y., VARMA, R., SALPEKAR, O., NOORDHUIS, P., LI, T., PASZKE, A., SMITH, J., VAUGHAN, B., DAMANIA, P., AND CHINTALA, S. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow. 13*, 12 (Aug. 2020), 3005–3018.

[39] libfabric Open Fabrics Interfaces (OFI). https://github.com/ofiwg/libfabric.

[40] LIU, H., URATA, R., YASUMURA, K., ZHOU, X., BANNON, R., BERGER, J., DASHTI, P., JOUPPI, N., LAM, C., LI, S., MAO, E., NELSON, D., PAPEN, G., TARIQ, M., AND VAHDAT, A. Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA, 2023), ACM SIGCOMM '23, Association for Computing Machinery, p. 499–515.

[41] LOGUINOV, D., CASAS, J., AND WANG, X. Graph-theoretic analysis of structured peer-to-peer systems:

routing distances and fault resilience. *IEEE/ACM Transactions on Networking 13*, 5 (2005), 1107–1120.

[42] LOGUINOV, D., KUMAR, A., RAI, V., AND GANESH, S. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2003), SIGCOMM '03, Association for Computing Machinery, p. 395–406.

[43] LU, Y., GU, H., YU, X., AND LI, P. X-NEST: A scalable, flexible, and high-performance network architecture for distributed machine learning. *Journal of Lightwave Technology 39*, 13 (2021), 4247–4254.

[44] LUO, L., WEST, P., NELSON, J., KRISHNAMURTHY, A., AND CEZE, L. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In *Proceedings of Machine Learning and Systems* (2020), I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, pp. 82–97.

[45] LUO, L., ZHANG, B., TSANG, M., MA, Y., CHU, C.-H., CHEN, Y., LI, S., HAO, Y., ZHAO, Y., LAKSHMINARAYANAN, G., WEN, E., PARK, J., MUDIGERE, D., AND NAUMOV, M. Disaggregated multi-tower: Topology-aware modeling technique for efficient large scale recommendation. In *Proceedings of Machine Learning and Systems* (2024), P. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., vol. 6, pp. 266–278.

[46] MEIJER, P. T. *Connectivities and diameters of circulant graphs*. PhD thesis, Theses (Dept. of Mathematics and Statistics)/Simon Fraser University, 1991.

[47] MELLETTE, W. M., DAS, R., GUO, Y., MCGUINNESS, R., SNOEREN, A. C., AND PORTER, G. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 1–18.

[48] MELLETTE, W. M., MCGUINNESS, R., ROY, A., FORENCICH, A., PAPEN, G., SNOEREN, A. C., AND PORTER, G. RotorNet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017).

[49] Microsoft Collective Communication Library (MSCCL). https://github.com/microsoft/msccl.

[50] MILLER, M., AND SIRAN, J. Moore graphs and beyond: A survey of the degree/diameter problem. *Electronic Journal of Combinatorics 1000* (2013).

[51] MONAKHOVA, E. A. A survey on undirected circulant graphs. *Discrete Mathematics, Algorithms and Applications 04*, 01 (2012), 1250002.

[52] NARAYANAN, D., SHOEYBI, M., CASPER, J., LEGRESLEY, P., PATWARY, M., KORTHIKANTI, V., VAINBRAND, D., KASHINKUNTI, P., BERNAUER, J., CATANZARO, B., ET AL. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), pp. 1–15.

[53] NAUMOV, M., KIM, J., MUDIGERE, D., SRIDHARAN, S., WANG, X., ZHAO, W., YILMAZ, S., KIM, C., YUEN, H., OZDAL, M., NAIR, K., GAO, I., SU, B.-Y., YANG, J., AND SMELYANSKIY, M. Deep learning training in facebook data centers: Design of scale-up and scale-out systems, 2020.

[54] NAUMOV, M., MUDIGERE, D., SHI, H.-J. M., HUANG, J., SUNDARAMAN, N., PARK, J., WANG, X., GUPTA, U., WU, C.-J., AZZOLINI, A. G., DZHULGAKOV, D., MALLEVICH, A., CHERNIAVSKII, I., LU, Y., KRISHNAMOORTHI, R., YU, A., KONDRATENKO, V., PEREIRA, S., CHEN, X., CHEN, W., RAO, V., JIA, B., XIONG, L., AND SMELYANSKIY, M. Deep learning recommendation model for personalization and recommendation systems, 2019.

[55] NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/.

[56] NVIDIA Collective Communications Library (NCCL). https://github.com/NVIDIA/nccl.

[57] NVIDIA ConnectX-6 Dx Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[58] PATARASUK, P., AND YUAN, X. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing 69*, 2 (2009), 117–124.

[59] RADFORD, A., WU, J., CHILD, R., LUAN, D., AMODEI, D., SUTSKEVER, I., ET AL. Language models are unsupervised multitask learners. *OpenAI blog 1*, 8 (2019), 9.

[60] RAJBHANDARI, S., LI, C., YAO, Z., ZHANG, M., AMINABADI, R. Y., AWAN, A. A., RASLEY, J., AND HE, Y. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *Proceedings of the 39th International Conference on Machine Learning* (17–23 Jul 2022), K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and

S. Sabato, Eds., vol. 162 of *Proceedings of Machine Learning Research*, PMLR, pp. 18332–18346.

[61] ROLIM, J., TVRDIK, P., TRDLIČKA, J., AND VRTO, I. Bisecting de bruijn and kautz graphs. *Discrete Applied Mathematics 85*, 1 (1998), 87–97.

[62] SACK, P., AND GROPP, W. Collective algorithms for multiported torus networks. *ACM Trans. Parallel Comput. 1*, 2 (feb 2015).

[63] SANDERS, P., SPECK, J., AND TRÄFF, J. L. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput. 35*, 12 (dec 2009), 581–594.

[64] SERGEEV, A., AND BALSO, M. D. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[65] SHAH, A., CHIDAMBARAM, V., COWAN, M., MALEKI, S., MUSUVATHI, M., MYTKOWICZ, T., NELSON, J., SAARIKIVI, O., AND SINGH, R. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 593–612.

[66] SHAZEER, N., MIRHOSEINI, A., MAZIARZ, K., DAVIS, A., LE, Q., HINTON, G., AND DEAN, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations* (2017).

[67] SHOEYBI, M., PATWARY, M., PURI, R., LEGRESLEY, P., CASPER, J., AND CATANZARO, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[68] SINGLA, A., HONG, C.-Y., POPA, L., AND GODFREY, P. B. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, Apr. 2012), USENIX Association, pp. 225–238.

[69] STANZIONE, D., WEST, J., EVANS, R. T., MINYARD, T., GHATTAS, O., AND PANDA, D. K. Frontera: The evolution of leadership computing at the national science foundation. In *Practice and Experience in Advanced Research Computing* (New York, NY, USA, 2020), PEARC '20, Association for Computing Machinery, p. 106–111.

[70] Telescent G4 Network Topology Manager. https://www.telescent.com/products.

[71] Texas Advanced Computing Center (TACC). https://www.tacc.utexas.edu/.

[72] THAKUR, R., RABENSEIFNER, R., AND GROPP, W. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[73] TRUONG, T.-N., AND TAKANO, R. Hybrid electrical/optical switch architectures for training distributed deep learning in large-scale. *IEICE Transactions on Information and Systems E104.D*, 8 (2021), 1332–1339.

[74] UENO, Y., AND YOKOTA, R. Exhaustive study of hierarchical allreduce patterns for large messages between gpus. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2019), pp. 430–439.

[75] VALADARSKY, A., SHAHAF, G., DINITZ, M., AND SCHAPIRA, M. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2016), CoNEXT '16, Association for Computing Machinery, p. 205–219.

[76] WANG, G., VENKATARAMAN, S., PHANISHAYEE, A., DEVANUR, N., THELIN, J., AND STOICA, I. Blink: Fast and generic collectives for distributed ML. *Proceedings of Machine Learning and Systems 2* (2020), 172–186.

[77] WANG, W., KHAZRAEE, M., ZHONG, Z., GHOBADI, M., JIA, Z., MUDIGERE, D., ZHANG, Y., AND KEWITSCH, A. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 739–767.

[78] WEBB, K. C., SNOEREN, A. C., AND YOCUM, K. Topology switching for data center networks. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 11)* (2011).

[79] WICKRAMASINGHE, U., AND LUMSDAINE, A. A survey of methods for collective communication optimization and tuning. *arXiv preprint arXiv:1611.06334* (2016).

[80] YOUNG, S., AKSOY, S., FIROZ, J., GIOIOSA, R., HAGGE, T., KEMPTON, M., ESCOBEDO, J., AND RAUGAS, M. SpectralFly: Ramanujan graphs as flexible and efficient interconnection networks. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2022), pp. 1040–1050.

[81] ZHU, Z., TEH, M. Y., WU, Z., GLICK, M. S., YAN, S., HATTINK, M., AND BERGMAN, K. Distributed deep learning training using silicon photonic switched architectures. *APL Photonics 7*, 3 (2022), 030901.

[82] ZOPH, B., BELLO, I., KUMAR, S., DU, N., HUANG, Y., DEAN, J., SHAZEER, N., AND FEDUS, W. St-moe: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906* (2022).

Appendix

In this appendix, we give additional evaluation results, along with formal mathematical definitions and analyses of the various techniques and concepts discussed in the main text:

- §A provides supplementary materials to evaluation section.
- §B gives formal definitions of reduce-scatter/allgather schedule and how one can be transformed into another.
- §C gives formal definitions of total-hop latency and bandwidth optimality, along with discussions on optimal allreduce schedule and computational cost of reduction.
- §D provides formal definitions of expansion techniques and optimality analysis of their expanded schedules.
- §E provides optimality analysis of BFB schedule generation and discusses variant formulations that support generating schedules for a fixed number of chunks and for heterogeneous network topology.
- §F discusses various generative topologies and the performance of their generated BFB schedules.
- §G provides proofs of all theorems in this paper.
- §H contains supplementary tables and figures. In particular, Table 9 gives a summary of topologies in this paper.

A Evaluation Appendix

- §A.1 presents experiment results that compare BFB schedule generation with communication solutions for switch networks: NCCL [56] and recursive halving & doubling.
- §A.2 shows experiment results to validate α-β cost model.
- §A.3 gives an analysis of Pareto-efficient topologies/schedules under different hardware and workload specifications.
- §A.4 details setup of simulated DNN training and the topologies generated by our topology finder.
- §A.5 provides the multi-commodity flow (MCF) formulation used to compute all-to-all throughput.
- §A.6 shows how to convert unidirectional topologies/schedules into bidirectional ones.

A.1 Comparison Against Switch Solutions

NCCL [56] and recursive halving & doubling (RH&D) are widely adopted collective communication solutions on switch networks. We assess the schedule performance of BFB against these solutions over two direct-connect 8-node topologies: hypercube and twisted hypercube [17]. Hypercube is widely used in HPC settings, and its connections perfectly match the communication pattern of RH&D. Twisted hypercube is a variant of hypercube with a lower diameter.

Figure 13 compares the baselines against our BFB schedule when run over either hypercube or twisted hypercube with $N=8$, $d=3$ on the testbed. At small M, all schedules and topologies perform roughly the same, except BFB can take advantage of the lower diameter of twisted hypercube and

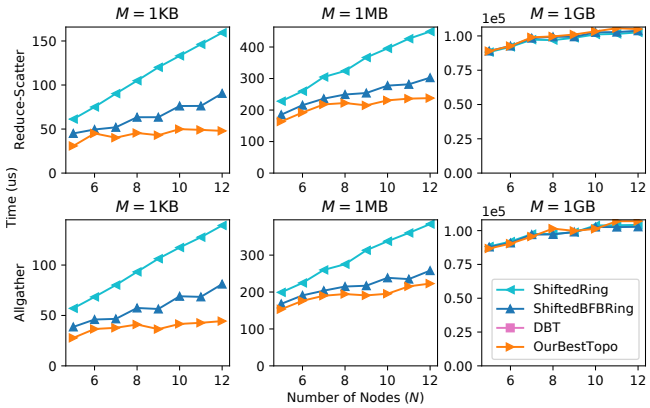


Figure 12: Comparing reduce-scatter and allgather runtimes of topologies. This figure shows the corresponding reduce-scatter and allgather results of Figure 6.

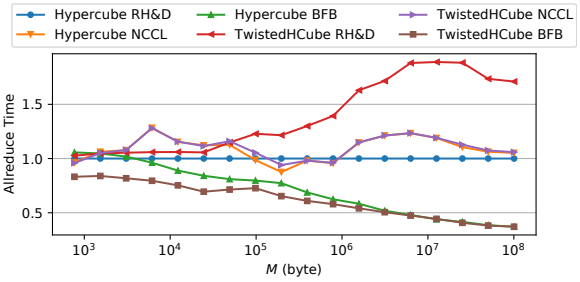


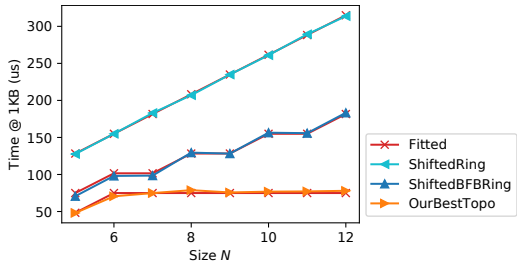
Figure 13: Comparing switch allreduce solutions (recursive halving & doubling (RH&D), NCCL) against BFB schedule on hypercube and twisted hypercube on $N=8, d=3$ testbed. The runtimes are normalized by the runtime of recursive halving & doubling on hypercube.

achieve ∼20% lower runtime. At large M, because BFB achieves BW optimality on both topologies, it performs even better with 60% lower runtime. RH&D and NCCL perform poorly as M grows because they cannot utilize all $d=3$ links simultaneously. At every comm step of RH&D, a node only communicates with one of the three neighbors, utilizing at most $1/3$ of the total bandwidth (similarly with NCCL). Also, because the schedule is not matched to the twisted hypercube, some nodes communicate with nodes multiple hops away, occupying more links and causing congestion.

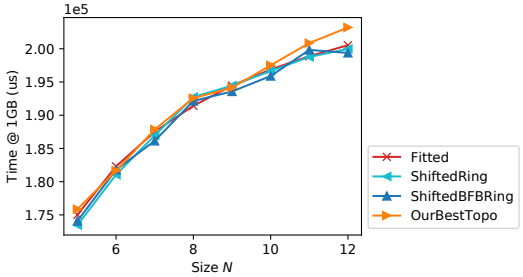
A.2 Cost Model Validation

Despite the wide acceptance of α-β cost model by previous literature [10, 11, 24, 63, 65], we also conducted a linear regression analysis to validate the cost model on our testbed. In particular, we want to verify that (1) total-hop latency follows $T_L = \alpha \cdot x + \varepsilon$ and (2) BW runtime follows $T_B = \frac{M}{B} \cdot y$, where x and y are the number of comm steps and bandwidth factor respectively ($y = 2 \cdot \frac{N-1}{N}$ if BW-optimal). ε is the constant latency[4] including time costs such as GPU kernel launching. Here, we use linear regression to derive the values of α, ε, and $1/B$, and compute the relative errors between the observed

[4]This part of latency is a fixed constant for all topologies and schedules, so it is omitted earlier.



(a) Total-Hop Latency T_L (Relative error: avg 1.71%, max 6.21%)



(b) BW Runtime T_B (Relative error: avg 0.47%, max 1.32%)

Figure 14: Linear regression results.

runtimes and expected runtimes. We fit the allreduce runtimes at 1KB to the total-hop latency, since BW runtime is negligible at such a small M. Similarly, we fit the runtimes at 1GB to the BW runtime, since total-hop latency is negligible at such a large M.

Figure 14 shows our results of linear regression analysis to verify our cost model. For total-hop latency, we obtain estimates $\alpha \approx 13.33$us and $\varepsilon \approx 21.60$us with low errors (average and maximum relative errors of 1.71% and 6.21% respectively). As one can see from Figure 14a, ShiftedRing and ShiftedBFBRing have a straight and a stair-step shape of runtime growth respectively, which match the expected numbers of comm steps $2(N-1)$ and $2\lfloor N/2 \rfloor$ respectively. For BW runtime, we get an estimate $1/B \approx 1.018 \times 10^{-4}$us/byte or $B \approx 79$Gbps with low errors (average and maximum relative errors of 0.47% and 1.32% respectively). As one can see from Figure 14b, all three topologies follow the fitted curve $2\frac{1\text{GB}}{B} \cdot \frac{N-1}{N} = 2T_B^*(N)$ since they are all BW-optimal. However, there is a gap between $B \approx 79$Gbps and the hardware theoretical bandwidth 4x25Gbps=100Gbps. Besides inevitable loss of bandwidth in actual communication, the gap can also be explained by the fact that computational cost of reduction also accounts for part of $1/B$ as discussed in §C.4.

A.3 Pareto-Efficiency Analysis

There could exist multiple Pareto-efficient topologies at given N and d. For different α and M/B, the Pareto-efficient topology with minimum allreduce runtime is also different. To see how N affects the best choice of topologies, we use topology finder (§5.4) to generate Pareto-efficient topologies at $d=4$ for N up to 2000 and pick the best one based on specific values of α and M/B.

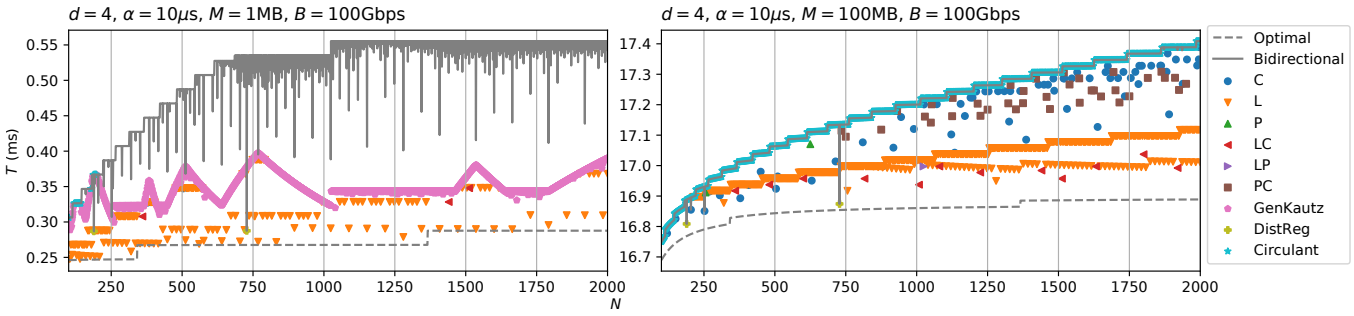


Figure 15: The minimum allreduce runtimes at different N for $d = 4$, $\alpha = 10$us, and $M/B = 1$MB/100Gbps, 100MB/100Gbps. "L", "P", and "C" stand for line graph, Cartesian power, and Cartesian product (of different graphs) respectively. For example, "LC" means the runtime is achieved by a topology whose construction involves line graph expansion and Cartesian product. "GenKautz", "DistReg", and "Circulant" stand for generalized Kautz graph (§F.2), distance regular graph (§F.3), and circulant graph (§F.4) respectively. The figures also show the best bidirectional topology known at different Ns. Degree expansion does not show up due to target $d = 4$ being relatively small.

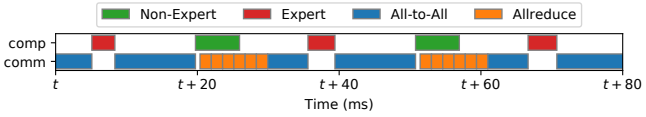


Figure 16: Example of a training timeline for Switch Transformers.

Figure 15 shows two examples of such analysis. At $M =$ 1MB, total-hop latency is more important than BW runtime. Thus, we see that generalized Kautz graph is the most popular one, being the best topology at many Ns. On the contrary, at $M = 100$MB, BW performance becomes the dominant factor, and thus circulant graph becomes the most popular one. Line graphs are also popular in both settings; however, line graph expansion requires target N to be divisible by some power of d, so it does not work for any N.

A.4 Details of Simulated Distributed Training

We simulate distributed ML training by first collecting actual compute times for model layers, running the models on an NVIDIA A100-SXM-80GB GPU, and then adding communication times according to the specific parallelism, e.g., data or expert parallelism. The communication time is calculated using α-β model for allreduce (§A.2) and multi-commodity flow for all-to-all (§A.5), assuming $\alpha = 10$us, $B = 100$Gbps over $d = 4$. Our simulation is designed to match the compute-communication overlap pattern of PyTorch Distributed Data Parallel [38]. As in PyTorch DDP, we bucket gradients that are ready for allreduce during backward propagation. Once the gradient volume reaches a predefined bucket capacity, an allreduce is performed. While a large bucket size results in less latency overhead, a small bucket size enhances compute-communication overlap. We choose the best bucket size by comparing the iteration times of bucket sizes {1MB, 10MB, 100MB, 1GB}. To ensure overlap, computation and communication are handled as independent streams, with the communication stream executing one collective at a time.

For the simulated training of Mixture-of-Experts (MoE) models, we follow the standard practice of expert parallelism [19, 34, 36, 60], where experts are sharded across all nodes while non-expert layers are replicated. All-to-all communications are needed before the expert layers to route tokens to the nodes of the assigned experts, and afterward to return tokens to the original nodes for the continuation of the forward/backward pass, thus blocking the computation stream. Furthermore, all-to-all and allreduce are not allowed to be overlapped as they occupy the same network bandwidth [36]. Figure 16 shows a timeline example of the simulated expert-parallel training. For simplicity, we assume a uniform token distribution among the experts, as MoE models are trained to balance expert load [19, 66, 82]. Consequently, the all-to-all communication is uniform across the nodes. We use the multi-commodity flow formulation (3) to compute the all-to-all communication time.

All hyperparameters, including sequence lengths and global batch sizes, are chosen according to the original paper of Switch Transformers [19]. The topology degree is fixed at 4, and the topology sizes are chosen such that the local batch size at each node is ≥ 1 and not so large as to run out of GPU memory. Table 7 includes all the Pareto-efficient topologies used in the simulation. For each model and topology size, we choose the topology that results in the smallest iteration time.

A.5 All-to-All Throughput

The problem of deriving the throughput of all-to-all communication on a topology can be nicely formulated as a multi-commodity flow (MCF) problem [4, 21, 32, 75, 78]. In an all-to-all MCF, each pair of nodes $(s,t) \in V_G^2$ acts as the source and sink of a commodity. The objective is to simultaneously route f units of flows from each s to t such that f is maximized, with flow allocation subject to flow conservation and edge capacities. In [4], the authors have devised an efficient LP formulation to compute the optimal f:

$$\begin{aligned} \text{maximize} \quad & f \\ \text{subject to} \quad & \sum_s y_{s,(u,v)} \leq 1, && \forall u,v \\ & f + \sum_v y_{s,(u,v)} \leq \sum_w y_{s,(w,u)}, && \forall s,u : s \neq u \\ & y_{s,(u,v)} \geq 0. && \forall s,u,v \end{aligned} \tag{3}$$

In LP (3), we assume the capacity/bandwidth of each link is 1 unit. Therefore, if the bandwidth of each link is B/d,

Topology	Allreduce		All-to-All	
$N=32,\ d=4$	T_L	T_B	$D(G)$	MCF
$L(K_{4,4})$	3α	$1.000M/B$	3	5.71e−2
DistReg$(4,32)$	4α	$0.969M/B$	4	5.26e−2
Theoretical Bound	3α	$0.969M/B$	3	5.80e−2
$N=64,\ d=4$	T_L	T_B	$D(G)$	MCF
$\Pi_{4,64}$	3α	$1.312M/B$	3	2.17e−2
L(DBJMod$(4,2)$)	4α	$1.000M/B$	4	2.21e−2
Diamond$^{\square 2}$	6α	$0.984M/B$	6	1.87e−2
Theoretical Bound	3α	$0.984M/B$	3	2.42e−2
$N=128,\ d=4$	T_L	T_B	$D(G)$	MCF
$L^2(K_{4,4})$	4α	$1.031M/B$	4	9.89e−3
L(DistReg$(4,32)$)	5α	$1.000M/B$	5	9.26e−3
BiRing$(2,8)\square$UniRing$(1,4)^{\square 2}$	10α	$0.992M/B$	10	5.21e−3
Theoretical Bound	4α	$0.992M/B$	4	1.00e−2
$N=256,\ d=4$	T_L	T_B	$D(G)$	MCF
DBJ$(4,4)$	4α	$1.328M/B$	4	4.04e−3
L^2(DBJMod$(4,2)$)	5α	$1.016M/B$	5	4.10e−3
L(Diamond$^{\square 2}$)	7α	$1.000M/B$	7	3.62e−3
DBJMod$(2,4)^{\square 2}$	10α	$0.996M/B$	8	2.94e−3
Theoretical Bound	4α	$0.996M/B$	4	4.39e−3
$N=512,\ d=4$	T_L	T_B	$D(G)$	MCF
$L^3(K_{4,4})$	5α	$1.039M/B$	5	1.88e−3
L^2(DistReg$(4,32)$)	6α	$1.008M/B$	6	1.78e−3
L(BiRing$(1,4)^{\square 3}\square$UniRing$(1,2)$)	11α	$1.000M/B$	11	1.12e−3
UniRing$(1,4)^{\square 3}\square$UniRing$(1,8)$	16α	$0.998M/B$	16	5.58e−4
Theoretical Bound	5α	$0.998M/B$	5	1.90e−3
$N=1024,\ d=4$	T_L	T_B	$D(G)$	MCF
$\Pi_{4,1024}$	5α	$1.332M/B$	5	8.01e−4
$L^3(C(16,\{3,4\}))$	6α	$1.020M/B$	6	8.12e−4
L^2(Diamond$^{\square 2}$)	8α	$1.004M/B$	8	7.34e−4
L(DBJMod$(2,4)^{\square 2}$)	11α	$1.000M/B$	9	6.18e−4
(UniRing$(1,4)\square$UniRing$(1,8)$)$^{\square 2}$	20α	$0.999M/B$	20	2.79e−4
Theoretical Bound	5α	$0.999M/B$	5	8.57e−4

Table 7: Pareto-efficient topologies at $N\in\{32,64,128,256,512,1024\}$, $d=4$. The results are generated from the topology finder (§5.4). For notations of the topologies, see Table 3 and 9. For distance regular graphs (DistReg), see Table 8. The MCF values are computed using LP (3).

then fB/d represents the rate at which every node can send to every other node simultaneously.

A.6 Unidirectional to Bidirectional

Unidirectional topologies are technically feasible on optical testbeds. The optical cable contains two fibers, one for each direction, and the fabric can link them to two distinct end-hosts, thus enabling unidirectional topologies at no additional hardware cost.

However, in our evaluation, we only use bidirectional topologies. While unidirectional topologies can be realized by configuring the patch panel in simplex mode, the requisite overlay routing for the reverse path traffic (acks, etc.) is currently only supported using routing rules performed by the host kernel as opposed to the NIC, leading to unpredictable RTTs. Therefore, we can functionally validate unidirectional topologies on our testbed, but we cannot accurately evaluate their performance. Note that newer NICs [9, 57] do support hardware offloading for these rules, which we will examine in future work.

While this paper considers unidirectional topologies a lot, many of the techniques can be conveniently applied to bidirectional topologies as well. For example, BFB schedule generation, degree expansion, and Cartesian product can all be used on bidirectional topologies by replacing each bidirectional edge with two opposite unidirectional edges. The resulting degree expanded and Cartesian product topologies still have unidirectional edges in opposite pairs. Although line graph expansion only works within unidirectional topologies, there is a way to convert unidirectional topology and schedule to bidirectional ones with zero performance sacrifice. In this section, we will show how to convert a reverse-symmetric (see Definition 6) d-regular unidirectional topology G and its allgather schedule A to a $2d$-regular bidirectional topology G' and its schedule A' such that $T_L(A)=T_L(A')$ and $T_B(A)=T_B(A')$.

Let $g:V_G\to V_{G^T}$ be the isomorphism from G to G^T, then it is trivial to see that $g(A)$ (see Definition 7) is an allgather schedule for G^T. Observe that $G'=G\cup G^T$ is a $2d$-regular bidirectional topology. Consider both A and $g(A)$ as allgather schedules for bidirectional topology G'. Schedules A and $g(A)$ use disjoint sets of edges, because they use opposite directions. Thus, we can divide each shard into two halves. Let one half follow schedule A and the other half follow $g(A)$. Let such a schedule be A'.

It is trivial to see that $T_L(A)=T_L(A')$. As for $T_B(A)=T_B(A')$, it follows the fact that the total data size is halved for each of A and $g(A)$, but the bandwidth per edge is also halved due to the doubling of degree. Note that if A is BW-optimal, then A' is BW-optimal; however, A' is not necessarily Moore optimal if A is Moore optimal.

B Reduce-Scatter & Allgather

We use tuple $((v,C),(u,w),t)$ to denote that u sends v's chunk C to w at comm step t. Node v is the source and destination node of chunk C in allgather and reduce-scatter respectively. A communication schedule is thus a collection of tuples.

Definition 4 (Allgather)**.** *An algorithm (G,A) is an allgather algorithm if for arbitrary $x\in S$ and distinct $u,v\in V_G$, there exists a sequence in A:*

$$((v,C_1),(w_0,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots ((v,C_n),(w_{n-1},w_n),t_n),$$

where $w_0=v$, $w_n=u$, $t_1<t_2<\ldots<t_n$, and $x\in C_1\cap C_2\cap\cdots\cap C_n$.

This sequence serves to broadcast x from v to u. A reduce-scatter algorithm has the same definition except $w_0=u$, $w_n=v$. In reduce-scatter, we assume any chunk received by a node is immediately reduced with the node's local chunk.

In this paper, many of the techniques are discussed under allgather only. We will show that anything holds in either reduce-scatter or allgather has an equivalent version for the other collective operation. To do so, we use the concept of *transpose graph* from graph theory and define *reverse schedule*. We say a schedule A is *for* topology G if every $((v,C),(u,w),t)\in A$ satisfies $u,v,w\in V_G$ and $(u,w)\in E_G$.

Definition 5 (Reverse Schedule)**.** *Suppose A is a schedule for G. A reverse schedule A^T of A is a schedule for trans-*

pose graph G^T such that $((v,C),(u,w),t_{\max}-t+1) \in A^T$ iff $((v,C),(w,u),t) \in A$, where $t_{\max}$ is the max comm step in A.

It is trivial to see that $T_L(A) = T_L(A^T)$ and $T_B(A) = T_B(A^T)$. Note that $(u,w) \in E_{G^T}$ if and only if $(w,u) \in E_G$ by definition of transpose graph.

Theorem 1. *If A is a reduce-scatter/allgather schedule for G, then A^T is an allgather/reduce-scatter schedule for G^T.*

Theorem 1 has the following two corollaries:

Corollary 1.1. *Suppose $G \mapsto f(G)$ is a function to construct reduce-scatter/allgather schedule given graph G, then $G \mapsto f(G^T)^T$ is a function to construct allgather/reduce-scatter schedule given graph G.*

Corollary 1.2. *Suppose $(G,A) \mapsto (f(G), f(A))$ is a mapping within reduce-scatter/allgather algorithms, then $(G,A) \mapsto (f(G^T)^T, f(A^T)^T)$ is a mapping within allgather/reduce-scatter algorithms.*

For example, the line graph expansion in §5.1 can be seen as a mapping within allgather, and the BFB linear program (1) can be seen as a function to construct allgather schedule. Thus, Corollary 1.1 and 1.2 have shown that they both have equivalent versions in reduce-scatter.

In undirected topology, it is well-known that reduce-scatter and allgather are a pair of dual operations such that one can be transformed into another by reversing the communication in schedule [11]. It is similar for directed topology but with extra requirement and more complicated transformation. We define the following property for directed graphs:

Definition 6 (Reverse-Symmetry)**.** *A digraph G is reverse-symmetric if it is isomorphic to its own transpose graph G^T.*

In graph theory, there is a similar concept called *skew-symmetric graph*. Reverse-symmetry is a weaker condition than skew-symmetry.

We define a way to transform the schedule for G into a schedule for G^T based on graph isomorphism:

Definition 7 (Schedule Isomorphism)**.** *Suppose G and G' are isomorphic. Let $f : V_G \to V_{G'}$ be the graph isomorphism and A be a schedule for G, then $f(A)$ is a schedule for G' that $((f(v),C),(f(u),f(w)),t) \in f(A)$ iff $((v,C),(u,w),t) \in A$.*

Theorem 2. *Suppose G is reverse-symmetric. Let G^T be the transpose graph, and let $f : V_{G^T} \to V_G$ be the isomorphism from G^T to G. If (G,A) is a reduce-scatter/allgather algorithm, then $(G, f(A^T))$ is an allgather/reduce-scatter algorithm with $T_L(f(A^T)) = T_L(A)$ and $T_B(f(A^T)) = T_B(A)$.*

Theorem 2 establishes that given any reverse-symmetric topology, if we have either reduce-scatter or allgather, then we can construct both reduce-scatter and allgather. Since allreduce can be achieved by applying a reduce-scatter followed by an allgather, we only need one of reduce-scatter and allgather to construct a complete allreduce algorithm. Furthermore, if the reduce-scatter or allgather algorithm has runtime T, then the resulting allreduce algorithm has runtime $2T$.

Most of our base topologies are reverse-symmetric (Table 9). In addition, all of our expansion techniques also preserve reverse-symmetry. Thus, one can almost always use Theorem 2 to derive reduce-scatter and allreduce schedules from allgather schedule on our synthesized topologies. For non-reverse-symmetric topologies like generalized Kautz graph, one can apply Corollary 1.1 or 1.2 to construct reduce-scatter and allgather separately.

C Topology-Schedule Optimality

Because our cost model is only concerned with total-hop latency and BW runtime, the optimality of reduce-scatter/allgather algorithm is only related to total-hop latency optimality and BW optimality in this paper. Note that we also consider topology as a dimension that can be optimized, so optimality is discussed in the space of all topology-schedule combinations, i.e., *algorithms* by our definition.

C.1 Total-Hop Latency Optimality

Definition 8 (Total-Hop Latency Optimal)**.** *Given an N-node degree-d reduce-scatter/allgather algorithm (G,A), if any other N-node degree-d reduce-scatter/allgather algorithm (G',A') satisfies $T_L(A') \geq T_L(A)$, then (G,A) is total-hop latency optimal.*

Because in reduce-scatter/allgather, every node needs to send a shard of data to every other node, the number of comm steps is lower bounded by the graph diameter:

Theorem 3. *Every reduce-scatter/allgather algorithm (G,A) satisfies $T_L(A) \geq \alpha \cdot D(G)$, where $D(G)$ is the diameter of G.*

Because we can always construct a BFB schedule A for topology G with $T_L(A) = \alpha \cdot D(G)$, it follows the corollary:

Corollary 3.1. *An N-node degree-d reduce-scatter/allgather algorithm (G,A) is total-hop latency optimal if and only if $T_L(A) = \alpha \cdot D(G) = \alpha \cdot \min\{D(G') : |V_{G'}| = N, \deg(G') = d\}$.*

The minimum diameter of a directed graph given a number of vertices and degree is still an open question. One can check *degree/diameter problem* [50] for more information. However, as a close upper bound of number of vertices given degree and diameter, the *Moore bound* for digraph is sufficient to tell the total-hop latency optimality in most cases.

Definition 9 (Moore Bound)**.** *Let G be any degree-d digraph of diameter k. The Moore bound is an upper bound on the number of vertices in G:*

$$M_{d,k} = \sum_{i=0}^{k} d^i = \frac{d^{k+1}-1}{d-1}.$$

Definition 10 (Moore Optimal)**.** *Let (G,A) be an N-node degree-d reduce-scatter/allgather algorithm with $T_L(A) = k\alpha$, then (G,A) is Moore optimal if $N > M_{d,k-1}$.*

Because for any degree-d digraph G, $D(G) \geq k$ must be true as long as $|V_G| > M_{d,k-1}$, Moore optimality is a stronger condition than total-hop latency optimality. We define a function T_L^* such that $T_L^*(N,d)$ equals the Moore optimal total-hop latency of N-node degree-d reduce-scatter/allgather algorithms.

C.2 Bandwidth Optimality

Definition 11 (Bandwidth Optimal)**.** *Given an N-node degree-d reduce-scatter/allgather algorithm (G,A), if any other N-node degree-d reduce-scatter/allgather algorithm (G',A') satisfies $T_B(A') \geq T_B(A)$, then (G,A) is BW-optimal.*

In reduce-scatter/allgather, each node needs to send/receive at least $M \cdot \frac{N-1}{N}$ amount of data. Thus, the following holds:

Theorem 4. *$\frac{M}{B} \cdot \frac{N-1}{N}$ is a lower bound of $T_B(A)$ for any N-node reduce-scatter/allgather algorithm (G,A).*

Note that one can always construct a ring of degree d by sending d parallel edges from one node to the next node. The trivial ring reduce-scatter/allgather schedule has $\frac{M}{B} \cdot \frac{N-1}{N}$ BW runtime. Therefore, we have:

Corollary 4.1. *An N-node reduce-scatter/allgather algorithm (G,A) is BW-optimal if and only if $T_B(A) = \frac{M}{B} \cdot \frac{N-1}{N}$.*

We define a function T_B^* such that $T_B^*(N) = \frac{M}{B} \cdot \frac{N-1}{N}$ is the optimal BW runtime of N-node reduce-scatter/allgather algorithms. From Corollary 4.1, we have the following necessary and sufficient condition for BW optimality:

Theorem 5. *An allgather algorithm (G,A) is BW-optimal if and only if:*

1. *$\frac{1}{B/d}\sum_{((v,C),(u,w))\in A_t}|C| = T_B(A_t)$ for all $(u,v)\in E_G$ and $t \in \{1,\ldots,t_{\max}\}$. A_t is the subschedule of A at comm step t.*
2. *Pick any distinct $u,v \in V_G$. For each $x \in S$, there exists a unique $((v,C),(w,u),t) \in A$ such that $x \in C$.*

Condition 1 ensures that at each comm step, every link of topology G has equal workload, so no link finishes early and results in waste of bandwidth. Condition 2 ensures that no piece of data is received twice by some node, so no duplicated send exists.

C.3 Allreduce Optimality

In this paper, we construct an allreduce algorithm through a reduce-scatter followed by allgather. In such construction, the lower bound of allreduce algorithm is $2(T_L^*(N,d)+T_B^*(N))$. To compare this with the lower bound of any allreduce construction, in [58], the authors have proved that $2T_B^*(N)$ is indeed the lower bound of BW runtime of any allreduce algorithm. As for total-hop latency, a reduce-scatter followed by allgather has at least $2D(G)$ number of comm steps, so $2T_L^*(N,d)$ is the lower bound of total-hop latency. Although one can use all-to-all to construct an allreduce with number of comm steps equal to one diameter $D(G)$ (lower bound being $T_L^*(N,d)$ instead of $2T_L^*(N,d)$), the lower bound of BW runtime for all-to-all is $\frac{M}{B} \cdot (N-1) = N \cdot T_B^*(N)$, which is much worse than $2T_B^*(N)$.

There is also another way of constructing allreduce: reduce followed by broadcast. In such an approach, the number of comm steps can be twice the radius of G instead of twice the diameter. However, the Moore bound for graph diameter also applies to graph radius, so $2T_L^*(N,d)$ is still a lower bound of allreduce via reduce+broadcast. By Theorem 16, the total-hop latency optimal allreduce via reduce+broadcast is at most 2α lower than the total-hop latency of generalized Kautz graph can do with reduce-scatter plus allgather. Furthermore, reduce+broadcast is usually poor in BW performance.

C.4 Computational Cost

In this paper, we omit the computational cost of reduction operation in performance analysis. While this approach is commonly adopted in previous literature [10, 63, 65, 76], we give a formal reasoning why this approach is legitimate. It is not only because computational cost is generally orders of magnitude lower than network cost, but also because computational cost can be incorporated into network cost.

Assume a cost model where computation and network communication do not overlap at each node.[5] In particular, at each comm step of reduce-scatter, the computation to reduce chunks happens immediately after the node receives all chunks and before the node starts to send out chunks for the next comm step. We adopt notations from [11], where γ denotes the computational time cost per size of data. Like total-hop latency and BW runtime, we also let $T_C(A)$ be the total time spent on computation by schedule A. As argued in [11], a lower bound of computational cost is $T_C \geq M \cdot \gamma \cdot \frac{N-1}{N}$ for both reduce-scatter and allreduce, which is identical to the BW optimality of reduce-scatter and half of that of allreduce. The following theorem shows that BW runtime of a schedule can act as an upper bound for the computational time.

Theorem 6. *Given a reduce-scatter algorithm (G,A), suppose $T_B(A) = \frac{M}{B} \cdot y$, then $T_C(A) \leq M \cdot \gamma \cdot y$.*

The rationale behind Theorem 6 is that the amount of computation for any node at a given comm step equals the amount of data the node receives during that comm step. Thus, **as we balance network transmission, it naturally leads to a more balanced computation.** With Theorem 6, if the BW runtime of some allreduce schedule A is $T_B(A) = 2\frac{M}{B} \cdot y$, then $T_B(A) + T_C(A) \leq M \cdot (\frac{2}{B} + \gamma) \cdot y$. We can thus simply define $B' = (\frac{1}{B} + \frac{\gamma}{2})^{-1}$, and then $2\frac{M}{B'} \cdot y$ can represent the sum of BW runtime and computational runtime altogether. The value of y is all that matters. The following corollary shows that if an algorithm is BW-optimal, then such representation is exact.

Corollary 6.1. *If allreduce algorithm (G,A) is BW-optimal, i.e., $T_B(A) = 2\frac{M}{B} \cdot \frac{N-1}{N}$, then $T_C(A) = M \cdot \gamma \cdot \frac{N-1}{N}$ and $T_B(A) + T_C(A) = 2M \cdot (\frac{1}{B} + \frac{\gamma}{2}) \cdot \frac{N-1}{N}$.*

When profiling a testbed, one can simply derive the value of $\frac{1}{B} + \frac{\gamma}{2}$ using BW-optimal topologies and use it as the new $1/B$ to apply the results of this paper. While it is still possible for two schedules with the same BW runtime to have different computational runtimes, such difference is bounded by the aforementioned theorems and orders of magnitude smaller than BW runtime.

[5] Otherwise, the computational cost would be even more negligible.

D Optimality of Expansion Techniques

In this section, we provide formal definitions and detailed performance analysis of expansion techniques.

D.1 Line Graph Expansion

Definition 12 (Line Graph)**.** *Given a directed graph (or digraph) G, each edge $(u,v) \in E_G$ corresponds to a vertex uv in the line graph $L(G)$. For every uv, vw pair in $V_{L(G)}$, there exists an edge $(uv, vw) \in E_{L(G)}$.*

In the case of multiedges between u and v, the line graph also contains multiple vertex uv.

Definition 1 (Schedule of Line Graph)**.** *Given an allgather schedule A_G for topology G, let $A_{L(G)}$ be the schedule for line graph $L(G)$ containing:*

1. *$((v'v, S), (v'v, vu), 1)$ for each edge $(v'v, vu) \in E_{L(G)}$ with $v'v \neq vu$.* **[Insert the 1st comm step in $A_{L(G)}$.]**
2. *$((v'v, C), (uw, ww'), t+1)$ for each $((v,C),(u,w),t) \in A_G$ and $v'v \neq ww'$.* **[Adapt A_G to form $A_{L(G)}$.]**

The following theorem gives the performance of the expanded schedule:

Theorem 7. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm, then $(L(G), A_{L(G)})$ is a dN-node allgather algorithm satisfying:*

$$T_L(A_{L(G)}) = T_L(A_G) + \alpha, \tag{4}$$

$$T_B(A_{L(G)}) \leq T_B(A_G) + \frac{M}{B} \cdot \frac{1}{N}. \tag{5}$$

From Theorem 7, one can see that the performance of the expanded schedule depends on that of the base schedule. Note that T_B also depends on M and B. For simplicity, we write $T_B(A_G)$ instead of $T_B(A_G, M, B)$ when there is no ambiguity. Theorem 7 makes an implicit assumption that $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ. This assumption, suggesting that T_B scales linearly with data size and inversely with bandwidth, should hold for any reasonably designed schedule.

Consequently, if we apply line graph expansion n times, the performance of the expanded schedule is:

Corollary 7.1. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $(L^n(G), A_{L^n(G)})$ is a $d^n N$-node allgather algorithm satisfying:*

$$T_L(A_{L^n(G)}) = T_L(A_G) + n\alpha, \tag{6}$$

$$T_B(A_{L^n(G)}) \leq T_B(A_G) + \frac{M}{B} \cdot \frac{d}{d-1} \left(\frac{1}{N} - \frac{1}{d^n N} \right). \tag{7}$$

In terms of the optimality of line graph expansion:

Theorem 8. *$(L^n(G), A_{L^n(G)})$ is Moore optimal if and only if (G, A_G) is Moore optimal.*

Theorem 9. *If (G, A_G) is BW-optimal with N nodes, then $T_B(A_{L^n(G)}) / T_B^*(d^n N) \leq 1 + [(d-1)N]^{-1}$ for all n.*

As mentioned in the main text, by Theorem 9, the key metric for the quality of base graph is how large it is while achieving both Moore and BW optimality. Currently, our largest such base graph that works for any even degree is Hamming graph $H(2, 1+d/2)$, which has $(1+d/2)^2 = \Theta(d^2)$ number of nodes. The corresponding line graph expanded topology is always Moore optimal and at most $O(1/d^3)$ away from BW optimality by Theorem 9.

Line graph expansion is closely related to BFB schedule for two reasons: (1) most of our base topologies like complete bipartite graph and Hamming graph use BFB schedule as the base schedule, and (2) the line graph expansion of BFB schedule is still a BFB schedule. To see the performance bound in Theorem 7 is tight, we have the following results in the context of BFB schedule:

Theorem 10. *Let A_G be a BFB allgather schedule for d-regular topology G with $|N^+(u)| > 1$ for all $u \in V_G$, then the expanded schedule $A_{L(G)}$ is a BFB allgather schedule for $L(G)$. In particular, if A_G is the optimal BFB schedule for G, then $A_{L(G)}$ is the optimal BFB schedule for $L(G)$ satisfying:*

$$T_B(A_{L(G)}) = T_B(A_G) + \frac{M}{B} \cdot \frac{1}{N}. \tag{8}$$

Corollary 10.1. *Let A_G be a BFB allgather schedule for d-regular topology G with $|N^+(u)| > 1$ for all $u \in V_G$, then the expanded schedule $A_{L^n(G)}$ is a BFB allgather schedule for $L^n(G)$. In particular, if A_G is the optimal BFB schedule for G, then $A_{L^n(G)}$ is the optimal BFB schedule for $L^n(G)$ satisfying:*

$$T_B(A_{L^n(G)}) = T_B(A_G) + \frac{M}{B} \cdot \frac{d}{d-1} \left(\frac{1}{N} - \frac{1}{d^n N} \right).$$

D.2 Degree Expansion

Definition 13 (Degree Expanded Topology)**.** *Given an N-node d-regular topology G without self-loops, construct the degree expanded nN-node nd-regular topology $G * n$:*

1. *For each vertex $v \in V_G$, add $v_1, \ldots, v_n$ to V_{G*n},*
2. *For each edge $(u,v) \in E_G$, add (u_i, v_j) to E_{G*n} for all i, j including $i = j$.*

Definition 2 (Degree Expanded Schedule)**.** *Given an allgather schedule A_G for G, construct A_{G*n} for $G * n$:*

1. *For all i, j including $i = j$ and for each $((v,C),(u,w),t) \in A_G$, add $((v_j, C), (u_j, w_i), t)$ to A_{G*n};*
2. *Divide shard S into equal-sized chunks $C_1, \ldots, C_{nd}$. Given $u_i, u_j \in V_{G*n}$ with $i \neq j$, add $((u_i, C_\alpha), (v_\alpha, u_j), t_{\max} + 1)$ to A_{G*n} for each $(v_1, u_j), \ldots, (v_{nd}, u_j) \in E_{G*n}$, where $t_{\max}$ is the max comm step in A_G.*

Theorem 11. *Given a d-regular topology G without self loops, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $(G * n, A_{G*n})$ is an nN-node allgather algorithm satisfying:*

$$T_L(A_{G*n}) = T_L(A_G) + \alpha, \tag{9}$$

$$T_B(A_{G*n}) = T_B(A_G) + \frac{M}{B} \cdot \frac{n-1}{nN}. \tag{10}$$

Corollary 11.1. *If (G, A_G) is BW-optimal and $T_B(A_G, M, B) = \tau(M/B)$ for some τ, then $(G * n, A_{G*n})$ is BW-optimal.*

Degree expansion preserves BW optimality. As for total-hop latency of degree expanded topology, observe that $T_L^*(N,d) = \Theta(\log_d N)$ and $\log_{nd} nN < \log_d N$, so T_L^* decreases as we apply degree expansion. Since T_L increases in degree expansion, Moore optimality is not preserved.

D.3 Cartesian Product Expansion

Definition 3 (Cartesian Product)**.** *The Cartesian product digraph $G_1 \Box G_2$ of digraphs G_1 and G_2 has vertex set $V_{G_1} \times V_{G_2}$ with vertex $\mathbf{u} = (u_1, u_2)$ connected to $\mathbf{v} = (v_1, v_2)$ iff either $(u_1, v_1) \in E_{G_1}$ and $u_2 = v_2$; or $u_1 = v_1$ and $(u_2, v_2) \in E_{G_2}$.*

Definition 3 generalizes to Cartesian product of multiple digraphs: $G_1 \Box G_2 \Box G_3 = (G_1 \Box G_2) \Box G_3$. The Cartesian product of n identical digraphs is denoted as Cartesian power $G^{\Box n}$.

Definition 14 (Schedule of Cartesian Power)**.** *Given an allgather schedule A_G for topology G and $n \in \mathbb{N}$, construct the schedule $A_{G^{\Box n}}$ for $G^{\Box n}$:*

1. *Construct the schedule $A^{(1)}$ as follows:*
2. *For $j = 1, \ldots, n$, for each $((w,C),(u,v),t) \in A_G$, add $(((\mathbf{x},w,\mathbf{z}),C),((\mathbf{y},u,\mathbf{z}),(\mathbf{y},v,\mathbf{z})),t+(j-1)t_{\max})$ to $A^{(1)}$ for all $\mathbf{x},\mathbf{y} \in V_G^{j-1}$ and $\mathbf{z} \in V_G^{n-j}$. $t_{\max}$ is the max comm step in A_G.*
3. *Similarly, construct $A^{(i)}$ for $i = 2, \ldots, n$ that each vertex $\mathbf{v}$ in $A^{(1)}$ is shifted by $i-1$ to $(\mathbf{v}[n-i+2:n], \mathbf{v}[1:n-i+1])$.*
4. *Divide each shard into n equal-sized subshards. Construct schedule $A_{G^{\Box n}}$ such that $A^{(i)}$ performs allgather over the i-th subshards of all nodes.*

Theorem 12. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $G^{\Box n}$ is an nd-regular topology, and $(G^{\Box n}, A_{G^{\Box n}})$ is an N^n-node allgather algorithm satisfying:*

$$T_L(A_{G^{\Box n}}) = n \cdot T_L(A_G), \tag{11}$$

$$T_B(A_{G^{\Box n}}) = T_B(A_G) \cdot \frac{N}{N-1} \cdot \frac{N^n - 1}{N^n}. \tag{12}$$

We then have the following corollary:

Corollary 12.1. *If (G, A_G) is BW-optimal and $T_B(A_G, M, B) = \tau(M/B)$ for some τ, then $(G^{\Box n}, A_{G^{\Box n}})$ is BW-optimal.*

Like degree expansion, Cartesian power expansion does not preserve Moore optimality.

We use BFB schedule generation when dealing with Cartesian product of distinct topologies:

Theorem 13. *Let $G_1, G_2, \ldots, G_n$ be topologies that*

1. *$G_1, \ldots, G_n$ are nontrivial simple digraphs;*
2. *Every G_i has BW-optimal BFB allgather schedule.*

Then, the optimal BFB allgather schedule, i.e. the schedule generated by BFB LP (1), for $G_1 \Box \ldots \Box G_n$ is also BW-optimal. The total-hop latency of the schedule equals $\alpha \cdot D(G_1 \Box \ldots \Box G_n) = \alpha \cdot \sum_i D(G_i)$.

The BFB schedule generation can also be used when individual topologies do not have BW-optimal BFB schedules; however, in such a case, we do not have performance bound for the schedule of the Cartesian product.

E BFB Schedule Generation

The LP formulation for u_2 in Figure 5 is:

$$\begin{aligned}
\text{minimize} \quad & U_{u_2,t} \\
\text{subject to} \quad & x_{v_1,(w_1,u_2),t} \le U_{u_2,t}, \\
& x_{v_1,(w_2,u_2),t} + x_{v_2,(w_2,u_2),t} \le U_{u_2,t}, \\
& x_{v_2,(w_3,u_2),t} \le U_{u_2,t}, \\
& x_{v_1,(w_1,u_2),t} + x_{v_1,(w_2,u_2),t} = 1, \\
& x_{v_2,(w_2,u_2),t} + x_{v_2,(w_3,u_2),t} = 1, \\
& 0 \le x_{v,(w,u_2),t} \le 1. \qquad \forall v, w
\end{aligned}$$

Definition 15 (BFB schedule)**.** *An allgather schedule A for G is a BFB schedule if A satisfies: $((v,C),(w,u),t) \in A$ only if $d(v,u) = d(v,w) + 1 = t$.*

Theorem 14. *A schedule A for G is a BFB allgather schedule if and only if the following are satisfied:*

1. *If $((v,C),(w,u),t) \in A$, then $d(v,u) = d(v,w) + 1 = t$;*
2. *For any distinct $u, v \in V_G$, the collection of chunks $\mathcal{C}_v = \{C \mid ((v,C),(w,u),t) \in A\}$ satisfies $S = \bigcup_{C \in \mathcal{C}_v} C$.*

Condition 1 ensures the schedule follows the breadth-first broadcast order. Condition 2 ensures every node receives the entire shard from every other node and thus a valid allgather.

E.1 Optimality

Theorem 15. *If A is a BFB schedule for G, then the total-hop latency $T_L(A) = \alpha \cdot D(G)$.*

There may exist many BFB schedules for a given topology G. They all have the same T_L but may have different T_Bs. Thus, the optimal BFB schedule is the one with the lowest T_B. Since every BFB schedule can be expressed as a solution to linear program (1), we have the following result:

Theorem 16. *Given any topology G, linear program (1) gives the optimal BFB schedule of G.*

An important implication of Theorem 16 is that **if we can show a BW-optimal BFB schedule exists for a topology G, then linear program (1) is guaranteed to generate one.** This has become an important tool for us to prove that BFB schedule generation can always generate BW-optimal schedules for some families of topologies (see §F). For the rest of this section, we show conditions that, if met by a topology, ensure it has a BW-optimal BFB schedule.

The following theorem shows the necessary and sufficient conditions for a BFB allgather schedule to be BW-optimal:

Theorem 17. *Suppose (G,A) is a BFB allgather schedule. (G,A) is BW-optimal if and only if:*

1. *There exists a sequence $N_1^-, N_2^-, \ldots, N_{D(G)}^- \in \mathbb{N}$ such that for any $x \in \mathbb{N}$ and $u \in V_G$, $|N_x^-(u)| = N_x^-$.*
2. *For any $(w,u) \in E_G$, $\sum_{((v,C),(w,u)) \in A_t} |C| = \frac{M}{N} |N_t^-(u)| / d = \frac{M}{N} N_t^- / d$.*

We assume G is d-regular. Condition 1 and 2 together ensure that at each comm step, all links have perfectly balanced workloads. In Theorem 13, we have already proven that *a Cartesian product graph has BW-optimal BFB schedule if it is the product of graphs that each have a BW-optimal BFB schedule.* Here, Theorem 17 also leads to the following sufficient condition for a bidirectional topology to have a BW-optimal BFB schedule:

Theorem 18. *There exists a BW-optimal BFB schedule for undirected graph G if for every distance x, two of the following constants exist:*

1. $N_x = |N_x(u)|$ *for any* $u \in V_G$;
2. $a_x = |N_x(u) \cap N_{x-1}(w)|$ *for any* $u \in V_G$ *and* $w \in N(u)$;
3. $b_x = |N(u) \cap N_{x-1}(v)|$ *for any* $u \in V_G$ *and* $v \in N_x(u)$.

Moreover, if two of N_x, a_x, b_x *exist, then the third one must also exist with* $N_x = da_x/b_x$.

Note that in undirected graphs, we have $N_x^+(u) = N_x^-(u) = N_x(u)$. To understand these constants, N_x is the number of data shards u needs to receive at comm step x; a_x is the number of data shards that can be transmitted by each link (w,u) at comm step x; b_x is the number of link (w,u)s that each data shard can use to transmit the data to u at comm step x. These three constants collectively ensure that links are perfectly balanced with each link transmitting $\frac{M}{N}N_x/d = \frac{M}{N}a_x/b_x$ amount of data at comm step x.

Now, we give a *necessary and sufficient condition* for any topology to have a BW-optimal BFB schedule. The condition is derived based on the observation that the BFB optimization problem is equivalent to a job scheduling problem. In each comm step t, for each node u, we have a set of jobs $\{j_1, j_2, \ldots, j_m\}$ (data from the source nodes $v \in N_t^-(u)$) and a set of processors $\{p_1, p_2, \ldots, p_d\}$ (links from in-neighbors $w \in N^-(u)$). There exists a map f from any job to a set of processors that j_i can only be scheduled to the processors in $f(j_i)$ (in-neighbor ws satisfying $d(v,u) = d(v,w) + 1 = t$). Assuming jobs can be arbitrarily divided into subjobs for parallel execution on multiple processors, the problem is how to schedule these jobs to processors so that workloads are balanced across all processors. We have the following result:

Theorem 19. *The workloads can be balanced if and only if there exists no subset* $J \subseteq \{j_1, j_2, \ldots, j_m\}$ *such that*

$$\frac{|J|}{\left|\bigcup_{j \in J} f(j)\right|} > \frac{m}{d}.$$

Note that there is an independent scheduling problem for each comm step t and node u. Therefore, topology G has a BW-optimal BFB schedule if and only if:

1. At each comm step t, $|N_t^-(u)|$ is the same for all $u \in V_G$.
2. The scheduling problem w.r.t. each t and u satisfies the condition in Theorem 19.

E.2 Discrete Chunked BFB Schedule

The BFB LP (1) makes an assumption that shards can be divided arbitrarily and infinitesimally. However, to compile the schedule into an executable form, one may need a *discrete chunked schedule*, where each shard is divided into a fixed number of equal-sized chunks. In practice, $x_{v,(w,u),t}$s are usually solved to be rational numbers. We can divide each shard into a number of chunks equal to the LCM of $x_{v,(w,u),t}$s' denominators so that each $x_{v,(w,u),t}$ represents some integer number of chunks. This approach has worked for us in evaluations. However, there exists the case where each shard of the data can only be divided into P equal chunks (i.e., the whole data M can only be divided into PN equal chunks). In such a case, we show that *we can approximate the optimal discrete chunked BFB schedule in polynomial time*.

Consider the following integer program given u, t:

$$\begin{aligned} \min \quad & W_{u,t} \\ \text{s.t.} \quad & \sum_v y_{v,(w,u),t} \le W_{u,t}, && \forall w \in N^-(u) \\ & \sum_w y_{v,(w,u),t} = P, && \forall v \in N_t^-(u) \\ & y_{v,(w,u),t} \in \{0, 1, \ldots, P\}, && \forall w, v. \end{aligned} \tag{13}$$

Compared with (1), one can easily see that the optimal solution of (13) gives the optimal BFB allgather schedule when each shard of the data can only be divided into P chunks. One can also easily solve the LP relaxation of (13) in polynomial time. Let T_B^{OPT} be the optimal BW runtime of the schedule obtained by directly solving integer program (13). Suppose the LP relaxation gives a schedule with BW runtime T_B^{LP}, then it holds that $T_B^{\text{LP}} \le T_B^{\text{OPT}}$.

Let $y_{v,(w,u),t}^{\text{LP}}$s be the solution to the LP relaxation of (13). We can obtain an integer solution $y_{v,(w,u),t}$s of (13) by rounding $y_{v,(w,u),t}^{\text{LP}}$s up or down to integers. For each v, we have

$$\sum_w \left\lfloor y_{v,(w,u),t}^{\text{LP}} \right\rfloor \le P \le \sum_w \left\lceil y_{v,(w,u),t}^{\text{LP}} \right\rceil.$$

Thus, it is trivial to round $y_{v,(w,u),t}^{\text{LP}}$s to integer $y_{v,(w,u),t}$s that $\sum_w y_{v,(w,u),t} = P$ and $y_{v,(w,u),t} < y_{v,(w,u),t}^{\text{LP}} + 1$. We give the following approximation bound for the resulting schedule:

Theorem 20. *Rounding LP gives a solution with BW runtime* $T_B \le T_B^{OPT} + \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PN}$. *In addition, if topology G is Moore optimal, then* $T_B \le T_B^{OPT} + \frac{M}{B} \cdot \frac{d}{P}$.

The cost $\frac{M}{B} \cdot \frac{d}{P}$ is negligible since P can easily be hundreds or even thousands while degree d is usually a small integer.

E.3 Heterogeneous BFB Schedule

The BFB LP (1) assumes a homogeneous network. It turns out that with little modification, (1) can become an LP for

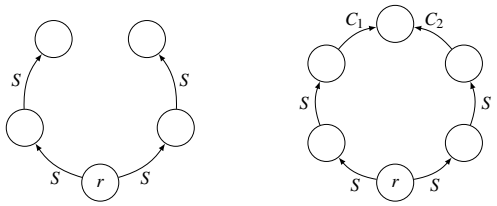


Figure 17: The broadcast paths of ring BFB allgather schedule. The left and right figures respectively show the broadcast patterns for odd- and even-sized bidirectional rings. Edges of the rings are omitted. C_1 and C_2 are two halves of shard S.

heterogeneous network too:

$$\begin{aligned} \min \quad & U_{u,t} \\ \text{s.t.} \quad & \alpha_{w,u} + \tfrac{M/N}{B_{w,u}} \sum_v x_{v,(w,u),t} \le U_{u,t}, \quad \forall w \in N^-(u) \\ & \sum_w x_{v,(w,u),t} = 1, \quad \forall v \in N_t^-(u) \\ & 0 \le x_{v,(w,u),t} \le 1, \quad \forall w, v. \end{aligned} \tag{14}$$

$\alpha_{w,u}$ and $B_{w,u}$ are the hop latency and bandwidth of link (w,u). In some cases, the α of some link (w,u) is so high that $\alpha_{w,u}$ alone dominates $U_{u,t}$ in (14) even though $\sum_v x_{v,(w,u),t} = 0$. This is problematic because one should not pay $\alpha_{w,u}$ if link (w,u) is not used. However, such a scenario can be easily detected after solving LP (14). One can avoid the issue by simply removing link (w,u) and solving the LP again.

F Generative Topologies

In this section, we introduce several topologies for which applying BFB schedule generation yields high-performance communication schedules.

F.1 Bidirectional Ring

Ring is the most common topology for allreduce. The traditional schedule on ring is to make each shard go a full circle to do reduce-scatter/allgather. In a bidirectional ring, one can simply make half the shard go clockwise and the other half go counterclockwise to utilize both directions of the links. Such a reduce-scatter/allgather schedule is BW-optimal but poor in total-hop latency with $T_L = (N-1)\alpha$. With BFB schedule generation, we discovered a new ring reduce-scatter/allgather schedule that achieves half the total-hop latency ($T_L = \lfloor N/2 \rfloor \alpha$) while maintaining BW optimality. From each node, the BFB allgather schedule broadcasts the *entire* shard clockwise and counterclockwise in parallel. Thus, each direction only needs to go half a circle instead of a full circle. If N is even, then the farthest node across the ring receives each half of the shard from each of its two neighbors in the end. Figure 17 shows examples in odd- and even-sized rings respectively.

F.2 Generalized Kautz Graph

Generalized Kautz graph [5, 25] is a low-T_L unidirectional topology that can be constructed for every N and d.

Definition 16 (Generalized Kautz Graph)**.** *The $\Pi_{d,m}$ digraph has the set of integers modulo m as vertex set. Its arc set A is defined as follows:*

$$A = \{(x,y) \mid y \equiv -dx - a, 1 \le a \le d\}.$$

If $m = d^{n+1} + d^n$, then $\Pi_{d,m} = K(d,n)$, where $K(d,n)$ is the Kautz graph $L^n(K_{d+1})$.

We apply BFB schedule generation to generalized Kautz graph. The resulting schedule is not always Moore optimal, but the following theorem shows that it is at most one α away from Moore optimality, i.e., $T_L \le T_L^*(N,d) + \alpha$:

Theorem 21. *Suppose $D(\Pi_{d,m}) = k$, then $m > M_{d,k-2}$.*

Remember Moore optimality is stricter than total-hop latency optimality, so it is possible that generalized Kautz graph is total-hop latency optimal. The special case, Kautz graph $K(d,n)$, is always Moore optimal and is, in fact, the largest known digraph in *degree/diamter problem* for any degree $d > 2$ [50].

As for BW performance, from Figure 18, one can see that generalized Kautz graph is also close to BW optimality, especially at higher degrees.

F.3 Distance-Regular Graph

In graph theory, distance-regular graphs are a family of highly symmetric undirected graphs. We can show that there exists a BW-optimal BFB schedule for any distance-regular graph, and thus LP (1) can always generate one. We borrow the following definition from [2]:

Definition 17 (Distance-Regular Graph)**.** *A connected graph G is distance-regular if for any vertices $x, y \in V_G$ and integers i, j, the number of vertices at distance i from x and distance j from y depends only on i, j and $d(x,y)$.*

In other words, there exists a constant $s_{i,j}^h$ for every h, i, j such that $s_{i,j}^h = |N_i(x) \cap N_j(y)|$ whenever $x, y \in V_G$ satisfy $d(x,y) = h$. Thus, we can apply Theorem 18 with $N_x = s_{x,x}^0$, $a_x = s_{x,x-1}^1$, and $b_x = s_{1,x-1}^x$.

The significance of distance-regular graph is not only about BW optimality. Many of distance-regular graphs have low diameters, so their schedules are not only BW-optimal but also close to, and in some cases exactly, Moore optimal. Table 8 gives examples of distance-regular graphs at $d = 4$. In addition, many of the base graphs mentioned in this paper are also distance-regular like complete bipartite graphs (Figure 1) and Hamming graphs. One can refer to [16] for a repository of distance-regular graphs.

F.4 Circulant Graph

Circulant graph is a well-studied topology in both graph theory and network design. Many popular network topologies like shifted ring, chordal ring, and loop network are either subcategories of or closely related to circulant graphs. The definition of circulant graph is as follows:

Definition 18. *The circulant graph $C(n, \{a_1, \ldots, a_k\})$ is a bidirectional graph with vertex set $\{0, 1, \ldots, n-1\}$ and each node i is adjacent to nodes $i \pm a_1, \ldots, i \pm a_k \pmod n$.*

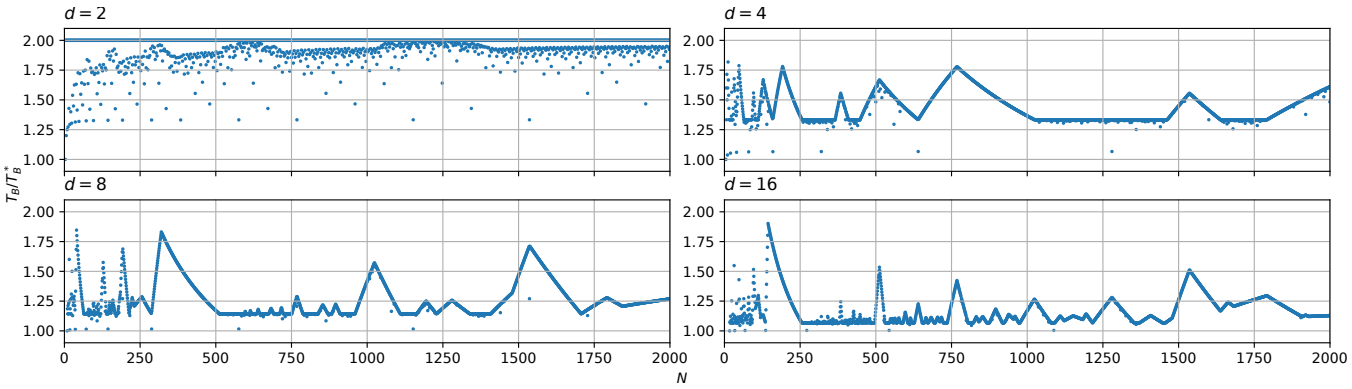


Figure 18: T_B/T_B^* **of generalized Kautz graph** $\Pi_{d,N}$ **up to** $N = 2000$**.** As shown, the BW runtime of $\Pi_{d,N}$ is less than or equal to $2T_B^*$ at all times for $d = 2,4,8,16$. In particular, the higher the degree is, the closer T_B is to optimal. As for total-hop latency, Theorem 21 shows that $T_L \le T_L^*(N,d) + \alpha$.

Graph Name	N	T_L	T_L^*	$T_L - T_L^*$	T_L^{**}	$T_L - T_L^{**}$
Octahedron J(4,2)	6	2	2	0	2	0
Paley graph P9≅H(2,3)	9	2	2	0	2	0
K5,5-I	10	3	2	1	2	1
Distance-3 graph of Heawood graph	14	3	2	1	2	1
Line graph of Petersen graph	15	3	2	1	2	1
4-cube Q4≅H(4,2)	16	4	2	2	2	2
Line graph of Heawood graph	21	3	2	1	3	0
Incidence graph of PG(2,3)	26	3	3	0	3	0
Incidence graph of AG(2,4) minus a parallel class	32	4	3	1	3	1
Odd graph O4	35	3	3	0	3	0
Line graph of Tutte's 8-cage	45	4	3	1	3	1
Doubled Odd Graph D(O4)	70	7	3	4	4	3
Incidence graph of GQ(3,3)	80	4	3	1	4	0
Line graph of Tutte's 12-cage	189	6	4	2	5	1
Incidence graph of GH(3,3)	728	6	5	1	6	0

Table 8: Examples of distance-regular graphs at $d = 4$ **[16].** T_L^{**} is the bidirectional Moore optimality.

Note that in this paper, we only consider connected circulant graphs, and $C(n,\{a_1,\ldots,a_k\})$ is connected if and only if $\gcd(n,a_1,\ldots,a_k) = 1$ [46, 51]. It is easy to see that $C(n,\{a_1,\ldots,a_k\})$ is an n-node $2k$-regular topology.

We have found that the BFB schedule generation seems to give BW-optimal schedules for all circulant graphs. In particular, we have the following conjecture:

Conjecture 1. *For any circulant graph* $C(n,\{a_1,\ldots,a_k\})$*, there exists a BW-optimal BFB schedule.*

While we leave a complete proof or disproof of this conjecture for future work, we have proved the conjecture holds when $k = 2$, which corresponds to the graph having degree 4.

Circulant graph revolutionized our Pareto frontier of topologies since it can be constructed for every N and even value d. It can provide a BW-optimal topology if our expansion techniques fail to produce one at some N and d. Since all circulant graphs seem to be BW-optimal, the question is what choices of $a_1,\ldots,a_k$ result in minimum total-hop latency, or equivalently, minimum diameter for a given n and k. While this remains largely an open question in graph theory [51], the case of $k = 2$ has been solved in [7]:

Theorem 22. *Given* $n > 6$ *and* $m = \lceil(-1+\sqrt{2n-1})/2\rceil$*, circulant graph* $C(n,\{m,m+1\})$ *has a diameter equal to* m*, which is the minimum diameter over all circulant graphs* $C(n,\{a_1,a_2\})$*.*

We can certainly use multiedge to apply this construction for any even degree that is ≥ 4. The resulting topology has $\Theta(\sqrt{N})$ diameter, which is a significant improvement in terms of total-hop latency when BW optimality is required. Previously, the only topology that is known to be BW-optimal for any N and d is ring, which has $\Theta(N)$ diameter.

G Proofs

Theorem 7. *Given a* d*-regular topology* G*, if* (G,A_G) *is an* N*-node allgather algorithm, then* $(L(G),A_{L(G)})$ *is a* dN*-node allgather algorithm satisfying:*

$$T_L(A_{L(G)}) = T_L(A_G) + \alpha, \tag{4}$$

$$T_B(A_{L(G)}) \le T_B(A_G) + \frac{M}{B}\cdot\frac{1}{N}. \tag{5}$$

Proof. Let $v'v,uw$ be arbitrary two distinct vertices in $L(G)$. We want to show there exists a sequence in $A_{L(G)}$ going from $v'v$ to uw like in Definition 4 for any $x \in S$. If $u = v$, then $((v'v,S),(v'v,uw),1)$ at the first comm step suffices. If $u \ne v$, because A_G is allgather, there exists a sequence in A_G:

$$((v,C_1),(v,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots \\ ((v,C_n),(w_{n-1},u),t_n),$$

where $t_1 < t_2 < \cdots < t_n$ and $x \in C_1 \cap C_2 \cap \cdots \cap C_n$. Thus, by Definition 1, there exists a sequence in $A_{L(G)}$:

$$((v'v,S),(v'v,vw_1),1),((v'v,C_1),(vw_1,w_1w_2),t_1+1),\ldots \\ ((v'v,C_n),(w_{n-1}u,uw),t_n+1),$$

as desired. The new algorithm $(L(G),A_{L(G)})$ has dM total data length, because the number of nodes has grown d-fold while the size of a shard remains the same.

As for $T_L(A_{L(G)})$ and $T_B(A_{L(G)})$, equality (4) trivially follows the Definition 1. Let $[A_{L(G)}]_t$ and $[A_G]_t$ be the subschedules of $A_{L(G)}$ and A_G at comm step t. Given $v \in V_G$, because G is d-regular, we have $|\{v'v \mid v'v \in V_{L(G)}\}| = |\{(v',v) \mid (v',v) \in E_G\}| = d$. Given any edge (uw,ww') and t, there are at most d number of $((v'v,C),(uw,ww'),t+1) \in A_{L(G)}$ for each

$((v,C),(u,w),t) \in A_G$ by Definition 1. Thus, given (uw, ww'),

$$\sum_{((v'v,C),(uw,ww'))\in[A_{L(G)}]_{t+1}} |C| \le \sum_{((v,C),(u,w))\in[A_G]_t} d\cdot|C|.$$

It follows that $T_B([A_{L(G)}]_{t+1}, dM, B) \le d\cdot T_B([A_G]_t, M, B)$ and hence $\sum_{t=2}^{t_{\max}+1} T_B([A_{L(G)}]_t, dM, B) \le d\cdot T_B(A_G, M, B)$. For the first comm step, we have

$$T_B([A_{L(G)}]_1, dM, B) = \frac{|S|}{B/d} = \frac{M/N}{B/d}.$$

Assuming $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, we have $d\cdot T_B(A_G, M, B) = T_B(A_G, dM, B)$. It follows that

$$T_B(A_{L(G)}, dM, B) = \sum_{t=1}^{t_{\max}+1} T_B([A_{L(G)}]_t, dM, B)$$
$$\le \frac{M/N}{B/d} + d\cdot T_B(A_G, M, B) = T_B(A_G, dM, B) + \frac{dM}{B}\cdot\frac{1}{N}.$$

Replacing dM by M gives (5) as desired. □

Corollary 7.1. *Given a d-regular topology G, if (G, A_G) is an N-node allgather algorithm with $T_B(A_G, M, B) = \tau(M/B)$ for some constant τ, then $(L^n(G), A_{L^n(G)})$ is a d^nN-node allgather algorithm satisfying:*

$$T_L(A_{L^n(G)}) = T_L(A_G) + n\alpha, \tag{6}$$

$$T_B(A_{L^n(G)}) \le T_B(A_G) + \frac{M}{B}\cdot\frac{d}{d-1}\left(\frac{1}{N} - \frac{1}{d^nN}\right). \tag{7}$$

Theorem 9. *If (G, A_G) is BW-optimal with N nodes, then $T_B(A_{L^n(G)})/T_B^*(d^nN) \le 1 + [(d-1)N]^{-1}$ for all n.*

Proof. If (G, A_G) is BW-optimal, then $T_B(A_G) = \frac{M}{B}\cdot\frac{N-1}{N}$ and

$$T_B(A_{L^n(G)}) \le \frac{M}{B}\left[1 + \frac{1}{d-1}\left(\frac{1}{N} - \frac{d}{d^nN}\right)\right]. \tag{15}$$

It is trivial to see that $(15)/T_B^*(d^nN) \nearrow 1 + [(d-1)N]^{-1}$ as $n \to \infty$. □

Theorem 1. *If A is a reduce-scatter/allgather schedule for G, then A^T is an allgather/reduce-scatter schedule for G^T.*

Proof. Suppose (G, A) is a reduce-scatter algorithm. For arbitrary $x \in S$ and distinct $u, v \in V_G$, there exists a sequence of tuples in A:

$$((v,C_1),(u,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots ((v,C_n),(w_{n-1},v),t_n),$$

where $t_1 < t_2 < \cdots < t_n$ and $x \in C_1 \cap C_2 \cap \cdots \cap C_n$. It follows that there exists a sequence of tuples in A^T:

$$((v,C_n),(v,w_{n-1}),t_n'),\ldots ((v,C_2),(w_2,w_1),t_2'),((v,C_1),(w_1,u),t_1').$$

where $t_i' = t_{\max} - t_i + 1$, so $t_n' < \cdots < t_2' < t_1'$. Since u, v, x are abitrary, A^T is an allgather schedule on G^T. One can similarly show that if (G, A) is an allgather algorithm, then (G^T, A^T) is a reduce-scatter algorithm. □

Corollary 1.1. *Suppose $G \mapsto f(G)$ is a function to construct reduce-scatter/allgather schedule given graph G, then $G \mapsto f(G^T)^T$ is a function to construct allgather/reduce-scatter schedule given graph G.*

Corollary 1.2. *Suppose $(G, A) \mapsto (f(G), f(A))$ is a mapping within reduce-scatter/allgather algorithms, then $(G, A) \mapsto (f(G^T)^T, f(A^T)^T)$ is a mapping within allgather/reduce-scatter algorithms.*

Theorem 2. *Suppose G is reverse-symmetric. Let G^T be the transpose graph, and let $f : V_{G^T} \to V_G$ be the isomorphism from G^T to G. If (G, A) is a reduce-scatter/allgather algorithm, then $(G, f(A^T))$ is an allgather/reduce-scatter algorithm with $T_L(f(A^T)) = T_L(A)$ and $T_B(f(A^T)) = T_B(A)$.*

Proof. By definition of $f(A^T)$,

$$((f(v),C),(f(w),f(u)),t_{\max}-t+1) \in f(A^T)$$
$$\Leftrightarrow ((v,C),(w,u),t_{\max}-t+1) \in A^T$$
$$\Leftrightarrow ((v,C),(u,w),t) \in A.$$

Note that $(u,w) \in E_G \Leftrightarrow (w,u) \in E_{G^T} \Leftrightarrow (f(w), f(u)) \in E_G$, so $f(A^T)$ is a valid schedule for G.

Suppose (G, A) is a reduce-scatter algorithm. For any $x \in S$ and distinct $u, v \in V_G$, there exists a sequence of tuples in A:

$$((v,C_1),(u,w_1),t_1),((v,C_2),(w_1,w_2),t_2),\ldots ((v,C_n),(w_{n-1},v),t_n),$$

where $t_1 < t_2 < \cdots < t_n$ and $x \in C_1 \cap C_2 \cap \cdots \cap C_n$. It follows that there exists a sequence of tuples in $f(A^T)$:

$$((f(v),C_n),(f(v),f(w_{n-1})),t_n'),$$
$$((f(v),C_{n-1}),(f(w_{n-1}),f(w_{n-2})),t_{n-1}'),$$
$$\vdots$$
$$((f(v),C_1),(f(w_1),f(u)),t_1'),$$

where $t_i' = t_{\max} - t_i + 1$, and $x \in C_n \cap C_{n-1} \cap \cdots \cap C_1$. Because f is a bijection, $(G, f(A^T))$ is an allgather algorithm. $T_L(A) = T_L(f(A^T))$ and $T_B(A) = T_B(f(A^T))$ are trivial, and one can similarly prove that if (G, A) is an allgather algorithm, then $(G, f(A^T))$ is a reduce-scatter algorithm. □

Theorem 3. *Every reduce-scatter/allgather algorithm (G, A) satisfies $T_L(A) \ge \alpha\cdot D(G)$, where $D(G)$ is the diameter of G.*

Proof. The proof is mentioned in text. □

Corollary 3.1. *An N-node degree-d reduce-scatter/allgather algorithm (G, A) is total-hop latency optimal if and only if $T_L(A) = \alpha\cdot D(G) = \alpha\cdot\min\{D(G') : |V_{G'}| = N, \deg(G') = d\}$.*

Theorem 4. *$\frac{M}{B}\cdot\frac{N-1}{N}$ is a lower bound of $T_B(A)$ for any N-node reduce-scatter/allgather algorithm (G, A).*

Proof. The proof is mentioned in text. □

Corollary 4.1. *An N-node reduce-scatter/allgather algorithm (G, A) is BW-optimal if and only if $T_B(A) = \frac{M}{B}\cdot\frac{N-1}{N}$.*

Theorem 5. *An allgather algorithm (G, A) is BW-optimal if and only if:*

1. $\frac{1}{B/d}\sum_{((v,C),(u,w))\in A_t}|C| = T_B(A_t)$ *for all* $(u,v)\in E_G$ *and* $t\in\{1,\ldots,t_{\max}\}$. A_t *is the subschedule of* A *at comm step* t.
2. *Pick any distinct* $u,v\in V_G$. *For each* $x\in S$, *there exists a unique* $((v,C),(w,u),t)\in A$ *such that* $x\in C$.

Proof. If $T_B(A) = T_B^*(N) = \frac{M}{B}\cdot\frac{N-1}{N}$, then the amount of data received by each vertex must be equal to $M\cdot\frac{N-1}{N}$, and the ingress bandwidth B must be fully utilized. If condition 1 does not hold, then some link (w,u) is not fully utilized. If condition 2 does not hold, then the amount of data received by some node is greater than $M\cdot\frac{N-1}{N}$.

If both 1 and 2 hold, then every vertex receives exactly $M\cdot\frac{N-1}{N}$ in total and bandwidth are fully utilized. Thus, $T_B(A) = T_B^*(N)$ and (G,A) is BW-optimal. □

Theorem 6. *Given a reduce-scatter algorithm* (G,A), *suppose* $T_B(A) = \frac{M}{B}\cdot y$, *then* $T_C(A)\le M\cdot\gamma\cdot y$.

Proof. At any comm step t, suppose the BW runtime is $T_B(A_t) = \frac{M}{B}\cdot y_t$. It follows at comm step t, the amount of data each node receives is at most $B\cdot T_B(A_t) = M\cdot y_t$, so $T_C(A_t)\le M\cdot\gamma\cdot y_t$. The theorem trivially follows $y = \sum_t y_t$. □

Corollary 6.1. *If allreduce algorithm* (G,A) *is BW-optimal, i.e.,* $T_B(A) = 2\frac{M}{B}\cdot\frac{N-1}{N}$, *then* $T_C(A) = M\cdot\gamma\cdot\frac{N-1}{N}$ *and* $T_B(A) + T_C(A) = 2M\cdot(\frac{1}{B}+\frac{\gamma}{2})\cdot\frac{N-1}{N}$.

Theorem 8. $(L^n(G), A_{L^n(G)})$ *is Moore optimal if and only if* (G,A_G) *is Moore optimal.*

Proof. Suppose $T_L(A_G) = \alpha k$. Thus, (G,A_G) is Moore optimal if and only if

$$N > M_{d,k-1} = \sum_{i=0}^{k-1} d^i = \frac{d^k}{d-1} - \frac{1}{d-1}. \tag{16}$$

$(L^n(G), A_{L^n(G)})$ is Moore optimal if and only if

$$d^nN > M_{d,k+n-1} \;\Leftrightarrow\; N > \frac{d^k}{d-1} - \frac{1}{d^n(d-1)}. \tag{17}$$

Because (17) − (16) < 1 and (16) is an integer, (16) and (17) are equivalent. □

Theorem 10. *Let* A_G *be a BFB allgather schedule for* d*-regular topology* G *with* $|N^+(u)|>1$ *for all* $u\in V_G$, *then the expanded schedule* $A_{L(G)}$ *is a BFB allgather schedule for* $L(G)$. *In particular, if* A_G *is the optimal BFB schedule for* G, *then* $A_{L(G)}$ *is the optimal BFB schedule for* $L(G)$ *satisfying:*

$$T_B(A_{L(G)}) = T_B(A_G) + \frac{M}{B}\cdot\frac{1}{N}. \tag{8}$$

Proof. It is trivial to see that $A_{L(G)}$ is a BFB allgather schedule on $L(G)$. For the sake of contradiction, suppose there exists a BFB schedule $A'_{L(G)}$ that $T_B(A'_{L(G)}) < T_B(A_G) + \frac{M}{B}\cdot\frac{1}{N}$. Let $x^*_{v'v,(wu,uu'),t}$s be the solution of BFB LP (1) corresponding to $A'_{L(G)}$. We build a schedule A'_G by constructing a solution of (1) such that

$$x_{v,(w,u),t} = \frac{1}{d}\sum_{v'\in N^-(v)} x^*_{v'v,(wu,uu'),t+1},$$

where $u'\in N^+(u)\setminus\{v\}$ is arbitrary. To verify the construction is a valid solution, given any $u\in V_G$ and $v\in N_t^-(w)$, the equality of (1) follows:

$$\sum_w x_{v,(w,u),t} = \frac{1}{d}\sum_{v'}\sum_w x^*_{v'v,(wu,uu'),t+1} = \frac{1}{d}\sum_{v'}\sum_{wu} x^*_{v'v,(wu,uu'),t+1} = \frac{1}{d}\sum_{v'} 1 = \frac{1}{d}\cdot d = 1.$$

The third equality follows the equality constraint in (1). Now, given $(w,u)\in E_G$, observe that

$$\sum_v x_{v,(w,u),t} = \frac{1}{d}\sum_v\sum_{v'} x^*_{v'v,(wu,uu'),t+1} = \frac{1}{d}\sum_{v'v} x^*_{v'v,(wu,uu'),t+1} \le \frac{1}{d}U^*_{uu',t+1}.$$

Thus, $U_{u,t} = \max_w\sum_v x_{v,(w,u),t} \le \frac{1}{d}U^*_{uu',t+1}$ and hence

$$\max_{u\in V_G} U_{u,t} \le \frac{1}{d}\max_{uu'\in V_{L(G)}} U^*_{uu',t+1}.$$

Note that $U^*_{uu',1} = 1$ for all $uu'\in V_{L(G)}$, as each node must send the full shard to every neighbor at the 1st comm step in any BFB allgather schedule. By (2), we have

$$T_B(A'_G) \le T_B(A'_{L(G)}) - \frac{M/(dN)}{B/d} = T_B(A'_{L(G)}) - \frac{M}{B}\cdot\frac{1}{N} < T_B(A_G),$$

contradicting A_G being the optimal BFB schedule. Thus, combined with inequality (5), we have proven $A_{L(G)}$ being optimal as well as the equality (8). □

Corollary 10.1. *Let* A_G *be a BFB allgather schedule for* d*-regular topology* G *with* $|N^+(u)|>1$ *for all* $u\in V_G$, *then the expanded schedule* $A_{L^n(G)}$ *is a BFB allgather schedule for* $L^n(G)$. *In particular, if* A_G *is the optimal BFB schedule for* G, *then* $A_{L^n(G)}$ *is the optimal BFB schedule for* $L^n(G)$ *satisfying:*

$$T_B(A_{L^n(G)}) = T_B(A_G) + \frac{M}{B}\cdot\frac{d}{d-1}\left(\frac{1}{N} - \frac{1}{d^nN}\right).$$

Theorem 11. *Given a* d*-regular topology* G *without self loops, if* (G,A_G) *is an* N*-node allgather algorithm with* $T_B(A_G,M,B) = \tau(M/B)$ *for some constant* τ, *then* $(G*n, A_{G*n})$ *is an* nN*-node allgather algorithm satisfying:*

$$T_L(A_{G*n}) = T_L(A_G) + \alpha, \tag{9}$$

$$T_B(A_{G*n}) = T_B(A_G) + \frac{M}{B}\cdot\frac{n-1}{nN}. \tag{10}$$

Proof. Let u_i, v_j be arbitrary two distinct vertices in $G*n$. Suppose $u\ne v$ in G, then for any $x\in S$, there exists a sequence in A_G:

$$((v,C_1),(v,w^{(1)}),t_1),((v,C_2),(w^{(1)},w^{(2)}),t_2),\ldots$$

$$((v,C_n),(w^{(n-1)},u),t_n),$$

where $t_1 < t_2 < \cdots < t_n$ and $x \in C_1 \cap C_2 \cap \cdots \cap C_n$. By Definition 2, there exists a sequence in A_{G*n}:

$$((v_j,C_1),(v_j,w_j^{(1)}),t_1),((v_j,C_2),(w_j^{(1)},w_j^{(2)}),t_2),\ldots$$
$$((v_j,C_n),(w_j^{(n-1)},u_i),t_n),$$

as desired. Now, suppose $u=v$ in G. By previous proof, the shard of v_j reaches every in-neighbor u'_α of u_i by the end of comm step $t_{\max}$ since $u' \neq v$. Then, the last comm step $t_{\max}+1$ added in step 2 of Definition 2 delivers the shard to u_i with each edge (u'_α,u_i) delivering $1/nd$ of a shard. Thus, A_{G*n} is a complete allgather.

In step 1 of Definition 2, we have $T_B([A_{G*n}]_t,nM,nB) = T_B([A_G]_t,M,B)$ and hence $\sum_{t=1}^{t_{\max}} T_B([A_{G*n}]_t,nM,nB) = T_B(A_G,M,B)$. The nM and nB are due to the fact that both the number of nodes and degree have grown n-fold. Thus,

$$\begin{aligned}T_B(A_{G*n},M,B) &= T_B(A_{G*n},nM,nB)\\ &= T_B(A_G,M,B)+T_B([A_{G*n}]_{t_{\max}+1},nM,nB)\\ &= T_B(A_G,M,B)+(n-1)\cdot\frac{(nM)/(nN)}{nd}\cdot\frac{1}{nB/(nd)}\\ &= T_B(A_G,M,B)+\frac{M}{B}\cdot\frac{n-1}{nN}.\end{aligned}$$

The first equality follows the assumption that $T_B(A_G,M,B) = \tau(M/B)$ for some constant τ. □

Corollary 11.1. *If (G,A_G) is BW-optimal and $T_B(A_G,M,B) = \tau(M/B)$ for some τ, then $(G*n,A_{G*n})$ is BW-optimal.*

Theorem 12. *Given a d-regular topology G, if (G,A_G) is an N-node allgather algorithm with $T_B(A_G,M,B) = \tau(M/B)$ for some constant τ, then $G^{\square n}$ is an nd-regular topology, and $(G^{\square n},A_{G^{\square n}})$ is an N^n-node allgather algorithm satisfying:*

$$T_L(A_{G^{\square n}}) = n\cdot T_L(A_G), \tag{11}$$

$$T_B(A_{G^{\square n}}) = T_B(A_G)\cdot\frac{N}{N-1}\cdot\frac{N^n-1}{N^n}. \tag{12}$$

Proof. We will show that $A^{(1)}$ is a valid allgather schedule. Since $A^{(i)}$s are simply starting at different dimensions, this also shows that $A^{(i)}$s and hence $A_{G^{\square n}}$ are all valid allgather schedules for $G^{\square n}$.

Let $\mathbf{u}$ be arbitrary vertex in $G^{\square n}$. For any $x \in S$, we will show that schedule $A^{(1)}$ broadcasts x from $\mathbf{u}$ to all vertices in $G^{\square n}$. At $j=1$, $A^{(1)}$ performs an allgather over vertices $\{(v_1,\mathbf{u}[2:n]) \mid v_1 \in V_G\}$ which induce a subgraph of $G^{\square n}$ isomorphic to G. Thus, x has been broadcast to all vertices in $\{(v_1,\mathbf{u}[2:n]) \mid v_1 \in V_G\}$. At $j=2$, $A^{(1)}$ performs an allgather over vertices $\{(v_1,v_2,\mathbf{u}[3:n]) \mid v_2 \in V_G\}$ for each v_1. By the end of $j=2$, x has been broadcast to all vertices in $\{(v_1,v_2,\mathbf{u}[3:n]) \mid v_1,v_2 \in V_G\}$. By the end of $j=n$, x has been broadcast to all vertices in $\{\mathbf{v} \mid \mathbf{v} \in V_G^n\} = V_{G^{\square n}}$. Since $\mathbf{u}$ and x are arbitrary, $A^{(1)}$ is a valid allgather schedule for $G^{\square n}$.

As for performance, (11) is trivial. To prove (12), observe that at each j in $A^{(1)}$, allgather A_G is performed with a data size $N^{j-1}M/n$ over the subgraph induced by $\{(\mathbf{y},v,\mathbf{z}) \mid v \in V_G\}$ for each $\mathbf{y} \in V_G^{j-1}, \mathbf{z} \in V_G^{n-j}$. The bandwidth of each node within the subgraph is $1/n$ of that in $G^{\square n}$. It follows that

$$\begin{aligned}T_B(A^{(1)},N^{n-1}M/n,nB) &= \sum_{j=1}^{n} T_B(A_G,N^{j-1}M/n,B)\\ &= \sum_{j=1}^{n}\frac{N^{j-1}}{n}T_B(A_G,M,B)\\ &= \frac{N^n-1}{n(N-1)}T_B(A_G,M,B).\end{aligned}$$

Therefore,

$$\begin{aligned}T_B(A_{G^{\square n}},M,B) &= \frac{n}{N^{n-1}}T_B(A_{G^{\square n}},N^{n-1}M,nB)\\ &= \frac{n}{N^{n-1}}T_B(A^{(1)},N^{n-1}M/n,nB)\\ &= \frac{n}{N^{n-1}}\cdot\frac{N^n-1}{n(N-1)}T_B(A_G,M,B)\\ &= T_B(A_G,M,B)\cdot\frac{N}{N-1}\cdot\frac{N^n-1}{N^n}.\end{aligned}$$

□

Corollary 12.1. *If (G,A_G) is BW-optimal and $T_B(A_G,M,B) = \tau(M/B)$ for some τ, then $(G^{\square n},A_{G^{\square n}})$ is BW-optimal.*

Theorem 13. *Let $G_1,G_2,\ldots,G_n$ be topologies that*

1. *$G_1,\ldots,G_n$ are nontrivial simple digraphs;*
2. *Every G_i has BW-optimal BFB allgather schedule.*

Then, the optimal BFB allgather schedule, i.e. the schedule generated by BFB LP (1), for $G_1\square\ldots\square G_n$ is also BW-optimal. The total-hop latency of the schedule equals $\alpha\cdot D(G_1\square\ldots\square G_n) = \alpha\cdot\sum_i D(G_i)$.

Proof. To prove the theorem, it is sufficient to show that if G_1 and G_2 have BW-optimal BFB schedules, then $G_1\square G_2$ has a BW-optimal BFB schedule. By Theorem 16, let $x^*_{v_1,(w_1,u_1),t_1}$s and $x^*_{v_2,(w_2,u_2),t_2}$s be the solutions of (1) on G_1 and G_2 respectively. Let $\mathbf{u}=(u_1,u_2),\mathbf{v}=(v_1,v_2)$. Define $r \in [0,1]$, which we will decide later. We construct a solution of (1) for $G_1\square G_2$ such that:

$$\begin{aligned}x_{\mathbf{v},((w_1,u_2),\mathbf{u}),t_1+t_2} &= \begin{cases} r\cdot x^*_{v_1,(w_1,u_1),t_1} & \text{if } u_2 \neq v_2,\\ x^*_{v_1,(w_1,u_1),t_1} & \text{if } u_2 = v_2,\end{cases}\\ x_{\mathbf{v},((u_1,w_2),\mathbf{u}),t_1+t_2} &= \begin{cases} (1-r)\cdot x^*_{v_2,(w_2,u_2),t_2} & \text{if } u_1 \neq v_1,\\ x^*_{v_2,(w_2,u_2),t_2} & \text{if } u_1 = v_1.\end{cases}\end{aligned} \tag{18}$$

First of all, because $d_{G_1\square G_2}(\mathbf{v},\mathbf{u}) = d_{G_1}(v_1,u_1)+d_{G_2}(v_2,u_2)$, it is easy to verify that (18) gives a BFB schedule. In addition, for any distinct $\mathbf{u},\mathbf{v} \in G_1\square G_2$ with $u_1 \neq v_1$ and $u_2 \neq v_2$,

$$\begin{aligned}\sum_{\mathbf{w}} x_{\mathbf{v},(\mathbf{w},\mathbf{u}),t_1+t_2} &= r\sum_{w_1} x^*_{v_1,(w_1,u_1),t_1}+(1-r)\sum_{w_2} x^*_{v_2,(w_2,u_2),t_2}\\ &= r+(1-r)\\ &= 1\end{aligned}$$

satisfying the equality in (1). The $u_1=v_1$ or $u_2=v_2$ case is trivial. Because G_1 and G_2 have BW-optimal BFB schedule,

by Theorem 17, for any $(w_1,u_1)\in E_{G_1}$,

$$\sum_{v_1\in N_t^{-G_1}(u_1)} x^*_{v_1,(w_1,u_1),t} = \frac{N_t^{-G_1}}{d_1}, \tag{19}$$

where $N_t^{-G_1}(u_1)$ is $N_t^-(u_1)$ in G_1. Define $N_{t_1,t_2}^{-G_1\Box G_2}(\mathbf{u}) = N_{t_1}^{-G_1}(u_1)\times N_{t_2}^{-G_2}(u_2)$, then it holds that

$$N_t^{-G_1\Box G_2}(\mathbf{u}) = \bigcup_{t_1=0}^{t} N_{t_1,t-t_1}^{-G_1\Box G_2}(\mathbf{u})$$

$$= N_t^{-G_1}(u_1)\times\{u_2\}\cup\{u_1\}\times N_t^{-G_2}(u_2)\cup\bigcup_{t_1=1}^{t-1} N_{t_1,t-t_1}^{-G_1\Box G_2}(\mathbf{u}).$$

Thus, (19) gives

$$\sum_{\mathbf{v}\in N_t^{-G_1\Box G_2}(\mathbf{u})} x_{\mathbf{v},((w_1,u_2),\mathbf{u}),t} = \frac{N_t^{-G_1}}{d_1} + r\sum_{t_1=1}^{t-1}\frac{N_{t_1}^{-G_1}N_{t-t_1}^{-G_2}}{d_1}. \tag{20}$$

for any $((w_1,u_2),\mathbf{u})\in E_{G_1\Box G_2}$. For G_2, one can similarly get

$$\sum_{\mathbf{v}\in N_t^{-G_1\Box G_2}(\mathbf{u})} x_{\mathbf{v},((u_1,w_2),\mathbf{u}),t} = \frac{N_t^{-G_2}}{d_2} + (1-r)\sum_{t_2=1}^{t-1}\frac{N_{t-t_2}^{-G_1}N_{t_2}^{-G_2}}{d_2}. \tag{21}$$

The value of r is the solution to (20)$=$(21):

$$\frac{N_t^{-G_1}}{d_1} + r\sum_{t_1=1}^{t-1}\frac{N_{t_1}^{-G_1}N_{t-t_1}^{-G_2}}{d_1} = \frac{N_t^{-G_2}}{d_2} + (1-r)\sum_{t_2=1}^{t-1}\frac{N_{t-t_2}^{-G_1}N_{t_2}^{-G_2}}{d_2}.$$

To see there is always a solution $r\in[0,1]$, we have $N_t^{-G_1}\le d_1\cdot N_{t-1}^{-G_1}$ and $N_t^{-G_2}\le d_2\cdot N_{t-1}^{-G_2}$, so

$$\frac{N_t^{-G_1}}{d_1} - \frac{N_t^{-G_2}}{d_2} \le \frac{N_t^{-G_1}}{d_1} \le N_{t-1}^{-G_1} \le \sum_{t_2=1}^{t-1}\frac{N_{t-t_2}^{-G_1}N_{t_2}^{-G_2}}{d_2},$$

$$\frac{N_t^{-G_2}}{d_2} - \frac{N_t^{-G_1}}{d_1} \le \frac{N_t^{-G_2}}{d_2} \le N_{t-1}^{-G_2} \le \sum_{t_1=1}^{t-1}\frac{N_{t_1}^{-G_1}N_{t-t_1}^{-G_2}}{d_1}.$$

The last inequality follows that because G_2 is nontrivial simple digraph, $N_1^{-G_2}=d_2$ and hence $N_{t-1}^{-G_1}=N_{t-1}^{-G_1}N_1^{-G_2}/d_2$. Note that $a+rb=c+(1-r)d$ always has a solution $r\in[0,1]$ if $a-c\le d$ and $c-a\le b$. With (20)$=$(21), by Theorem 17, we have constructed a BW-optimal solution of (1) for $G_1\Box G_2$. The theorem trivially follows by induction. □

Theorem 14. *A schedule A for G is a BFB allgather schedule if and only if the following are satisfied:*

1. *If $((v,C),(w,u),t)\in A$, then $d(v,u)=d(v,w)+1=t$;*
2. *For any distinct $u,v\in V_G$, the collection of chunks $\mathcal{C}_v=\{C\mid((v,C),(w,u),t)\in A\}$ satisfies $S=\bigcup_{C\in\mathcal{C}_v}C$.*

Proof. Let v_0,v_k be arbitrary two distinct vertices in V_G with $d(v_0,v_k)=k$. For any $x\in S$, we want to show that there exists a path taking x from v_0 to v_k. At comm step k, conditions 1 and 2 guarantee that there exists $v_{k-1}\in N^-(v_k)$ and $((v_0,C_k),(v_{k-1},v_k),k)\in A$ such that $d(v_0,v_{k-1})=k-1$ and $x\in C_k$. At comm step $k-1$, similarly, it is guaranteed that there exists $v_{k-2}\in N^-(v_{k-1})$ and $((v_0,C_{k-1}),(v_{k-2},v_{k-1}),k-1)\in A$ such that $d(v_0,v_{k-2})=k-2$ and $x\in C_{k-1}$. Thus, we have a sequence of tuples in A:

$$((v_0,C_1),(v_0,v_1),1),((v_0,C_2),(v_1,v_2),2),\ldots ((v_0,C_k),(v_{k-1},v_k),k),$$

where $x\in C_1\cap C_2\cap\cdots\cap C_k$ as desired. In the other direction, if condition 1 fails, then A is not a BFB schedule; if condition 2 fails, then A is not a valid allgather schedule. □

Theorem 15. *If A is a BFB schedule for G, then the total-hop latency $T_L(A)=\alpha\cdot D(G)$.*

Proof. The proof is trivial. □

Theorem 16. *Given any topology G, linear program (1) gives the optimal BFB schedule of G.*

Proof. The proof is mentioned in text. □

Theorem 17. *Suppose (G,A) is a BFB allgather schedule. (G,A) is BW-optimal if and only if:*

1. *There exists a sequence $N_1^-,N_2^-,\ldots,N_{D(G)}^-\in\mathbb{N}$ such that for any $x\in\mathbb{N}$ and $u\in V_G$, $|N_x^-(u)|=N_x^-$.*
2. *For any $(w,u)\in E_G$, $\sum_{((v,C),(w,u))\in A_t}|C|=\frac{M}{N}|N_t^-(u)|/d=\frac{M}{N}N_t^-/d$.*

Proof. At comm step t, each vertex needs to receive shards from vertices in $N_t^-(u)$. By condition 1 of Theorem 5, each in-edge of vertex u receives equal amount of data, so each in-edge receives $\frac{M}{N}|N_t^-(u)|/d$. In addition, condition 1 of Theorem 5 also forces every edge in G receiving equal amount of data at any given comm step, so BW optimality is achieved if and only if $\frac{M}{N}|N_t^-(u)|/d=\frac{M}{N}|N_t^-(v)|/d=\frac{M}{N}N_t^-/d$ for all $u,v\in V_G$. Note that condition 2 of Theorem 5 is automatically satisfied. Thus, the conditions of Theorem 17 lead to Theorem 5, and vice versa. □

Theorem 18. *There exists a BW-optimal BFB schedule for undirected graph G if for every distance x, two of the following constants exist:*

1. *$N_x=|N_x(u)|$ for any $u\in V_G$;*
2. *$a_x=|N_x(u)\cap N_{x-1}(w)|$ for any $u\in V_G$ and $w\in N(u)$;*
3. *$b_x=|N(u)\cap N_{x-1}(v)|$ for any $u\in V_G$ and $v\in N_x(u)$.*

Moreover, if two of N_x,a_x,b_x exist, then the third one must also exist with $N_x=da_x/b_x$.

Proof. It is easy to see $N_x=da_x/b_x$. Constant N_xs satisfy condition 1 of Theorem 17. As for 2 of Theorem 17, at comm step t, consider a BFB schedule such that for any $u,v,w\in V_G$ with $d(v,u)=d(v,w)+1=t$, node w sends $1/b_t$ of v's shard to u. Thus, $\sum_{((v,C),(w,u))\in A_t}|C|=\frac{M}{N}a_t/b_t=\frac{M}{N}N_t/d$. □

Theorem 19. *The workloads can be balanced if and only if there exists no subset $J\subseteq\{j_1,j_2,\ldots,j_m\}$ such that*

$$\frac{|J|}{\left|\bigcup_{j\in J}f(j)\right|} > \frac{m}{d}.$$

Proof. Consider a flow network, where each j_a is connected to each $p_b \in f(j_a)$ with ∞ capacity. Source s is connected to each j_a with capacity 1, and each p_b is connected to sink t with capacity m/d. Thus, the workloads can be balanced if and only if the max flow is m. Given any subset J, consider the s-t cut $(A, \bar{A})$ that $A = s + J + f(J)$. The cut has capacity $m - |J| + \frac{m}{d}|f(J)|$, which is less than m if and only if the inequality is true. □

Theorem 20. *Rounding LP gives a solution with BW runtime* $T_B \le T_B^{OPT} + \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PN}$. *In addition, if topology G is Moore optimal, then* $T_B \le T_B^{OPT} + \frac{M}{B} \cdot \frac{d}{P}$.

Proof. For any (w,u) at comm step t, since $|N_{t-1}^-(w)| \le d^{t-1}$,

$$\sum_v y_{v,(w,u),t} < \sum_v 1 + y^{\text{LP}}_{v,(w,u),t} \le d^{t-1} + \sum_v y^{\text{LP}}_{v,(w,u),t}.$$

Thus, we have $W_{u,t} \le W^{\text{LP}}_{u,t} + d^{t-1}$. By (2),

$$\begin{aligned} T_B - T_B^{\text{OPT}} &\le T_B - T_B^{\text{LP}} \\ &\le \frac{M/N}{B/d} \cdot \frac{1}{P} \sum_{t=1}^{D(G)} d^{t-1} = \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PN}. \end{aligned}$$

Note that we need to divide (2) by P, because $y_{v,(w,u),t} \in [0,P]$ in (13) while $x_{v,(w,u),t} \in [0,1]$ in (1). If G is Moore optimal (i.e., $N > M_{d,D(G)-1}$), it follows that

$$T_B - T_B^{\text{OPT}} < \frac{M}{B} \cdot \frac{d(d^{D(G)}-1)}{(d-1)PM_{d,D(G)-1}} = \frac{M}{B} \cdot \frac{d}{P}.$$

□

Theorem 21. *Suppose* $D(\Pi_{d,m}) = k$, *then* $m > M_{d,k-2}$.

Proof. From [25], we know that $k \le \lceil \log_d m \rceil$. Then,

$$m \ge d^{k-1} > \frac{d^{k-1}-1}{d-1} = M_{d,k-2}.$$

□

Theorem 22. *Given* $n > 6$ *and* $m = \lceil (-1+\sqrt{2n-1})/2 \rceil$, *circulant graph* $C(n, \{m, m+1\})$ *has a diameter equal to m, which is the minimum diameter over all circulant graphs* $C(n, \{a_1, a_2\})$.

Proof. See [7]. □

H Supplementary Tables and Figures

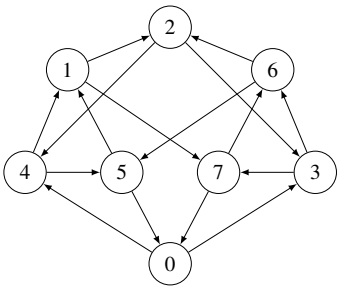


Figure 19: Diamond Topology $(N = 8, d = 2)$**.**

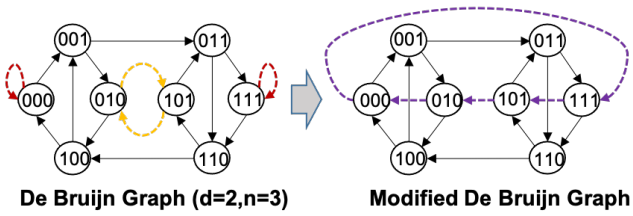


Figure 20: An Example of Modified de Bruijn Graph $(N = 8, d = 2)$**.** The modification rewires the self loops and 2-cycles in de Bruijn graph to form a single long cycle without violating degree constraint.

Topology	Notation	Degree	Size	Reverse-Symmetric	Bandwidth Optimal	Moore Optimal	BFB Schedule	Self-Loop	MultiEdge
Complete	K_m	$m-1$	m	✓	✓	✓	✓	×	×
Complete Bipartite (Fig 1)	$K_{d,d}$	d	$2d$	✓	✓	✓	✓	×	×
Hamming	$H(n,q) = K_q^{\square n}$	$n(q-1)$	q^n	✓	✓	$T_L = n$	✓	×	×
Kautz	$K(d,n) = L^n(K_{d+1})$	d	$d^n(d+1)$	✓	when $n=0$	✓	✓	×	×
Generalized Kautz (§F.2)	$\Pi_{d,m}$	d	$m \geq d+1$	×	when $m = d+1$	$T_L \leq T_L^* + 1$	✓	when $m \bmod (d+1) \neq 0$	×
Circulant (§F.4)	$C(m, \{a_1, \ldots, a_d\})$	d	m	✓	✓	×	✓	×	×
Directed Circulant		d	$d+2$	✓	✓	✓	✓	×	×
Bidirectional Ring	BiRing(d,m)	even d	$m \geq 3$	✓	✓	$T_L = \lfloor m/2 \rfloor$	✓	×	when $d > 2$
Unidirectional Ring	UniRing(d,m)	d	m	✓	✓	$T_L = m-1$	✓	×	when $d > 1$
Diamond (Fig 19)		2	8	×	✓	✓	×	×	×
de Bruijn	DBJ(d,n)	d	d^n	✓	when $n \leq 1$	✓	✓	✓	×
Modified de Bruijn (Fig 20)	DBJMod$(2,3)$	2	8	✓	✓	$T_L = 4$	×	×	×
	DBJMod$(2,4)$	2	16	×	✓	$T_L = 5$	×	×	×
	DBJMod$(3,2)$	3	9	×	✓	$T_L = 3$	×	×	×
	DBJMod$(4,2)$	4	16	×	✓	$T_L = 3$	×	×	×
Distance-Regular Graphs (§F.3)	DistReg(d,m)	d	m	✓	✓		✓	×	×

Table 9: Summary of Important Topologies.

SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads

Alind Khare[1], Dhruv Garg[1], Sukrit Kalra[2], Snigdha Grandhi[*3], Ion Stoica[2], and Alexey Tumanov[1]

[1]Georgia Tech [2]UC Berkeley [3]Adobe
{alindkhare, dgarg39, sgrandhi32, atumanov}@gatech.edu
{sukrit.kalra, istoica}@berkeley.edu

Abstract

The increasing deployment of ML models on the critical path of production applications requires ML inference serving systems to serve these models under unpredictable and bursty request arrival rates. Serving many models under such conditions requires a careful balance between each application's latency and accuracy requirements and the overall efficiency of utilization of scarce resources. Faced with this tension, state-of-the-art systems either choose a single model representing a static point in the latency-accuracy tradeoff space to serve all requests or incur latency target violations by loading specific models on the critical path of request serving. Our work instead resolves this tension through a resource-efficient serving of the entire range of models spanning the latency-accuracy tradeoff space. Our novel mechanism, SubNetAct, achieves this by carefully inserting specialized control-flow operators in pre-trained, weight-shared super-networks. These operators enable SubNetAct to dynamically route a request through the network to actuate a specific model that meets the request's latency and accuracy target. Thus, SubNetAct can serve a vastly higher number of models than prior systems while requiring upto 2.6× lower memory. More crucially, SubNetAct's near-instantaneous actuation of a wide-range of models unlocks the design space of fine-grained, reactive scheduling policies. We design one such extremely effective policy, SlackFit, and instantiate both SubNetAct and SlackFit in a real system, SuperServe. On real-world traces derived from a Microsoft workload, SuperServe achieves 4.67% higher accuracy for the same latency targets and 2.85× higher latency target attainment for the same accuracy.

1 Introduction

Recent advancements in machine learning (ML) techniques have unlocked vast improvements in both accuracy and efficiency of a wide-variety of tasks [5, 11, 19, 31, 41, 50]. As a result, ML models have been quickly deployed across a wide-range of applications in both datacenters [26, 40, 44, 49, 62] and the edge [6, 20, 35, 42], where they are subjected to the stringent requirements of production applications.

Notably, ML models on the critical path of these applications must deal with *unpredictable request rates* that rapidly change at a sub-second granularity. For example, web applications in datacenters increasingly rely on ML models [49, 62], and have extremely bursty request rates, with a 50× higher peak demand than average [34]. Similarly, request rates in autonomous vehicles change rapidly as a function of the terrain (city vs. freeway driving), time of the day etc. [20].

In the presence of these unpredictable request rates, ML inference serving systems that cater to production applications must strike a careful balance between three key requirements:

R1: Latency. Production applications have extremely stringent latency requirements, quantified through a Service-Level Objective (SLO) [33]. For example, both web serving [25, 44, 49] in datacenters and autonomous vehicles [20, 35] on the edge must maximize the number of requests completed within a specified SLO ranging from $10-100$ ms [20, 60].

R2: Accuracy. Production applications demand the highest-accuracy results possible within the latency targets of their requests. For example, higher accuracy has been intricately tied to a better user experience for web applications [26, 56]. Similarly, the safety of an autonomous vehicle heavily relies on the accuracy of its different ML models [20, 21].

R3: Resource-Efficiency. Web applications at Facebook process 200 trillion ML model requests daily [53] – a significant fraction of datacenter usage [44]. The proliferation of ML models and their reliance on expensive resources such as GPUs, TPUs [1], AWS Inferentia [2] etc. has led to resource tensions in both datacenters and the edge [42]. Thus, inference serving systems must make judicial use of these resources.

The first-generation of inference serving systems [13, 14, 25, 32, 39, 40, 60] resolve this tension by choosing a static point in the tradeoff space between **R1**-**R3** and serving all requests of an application using the chosen model. As a result, applications either miss their SLO targets (**R1**) under bursty

[*]Work done as a student at Georgia Tech

request rates or suffer degraded accuracy (**R2**) under normal conditions. More recently, state-of-the-art inference serving systems [45, 61] enable applications to register multiple ML models spanning the entire pareto frontier of latency (**R1**) and accuracy (**R2**) targets, and *automatically* choose a model based on the incoming request rates. To achieve this, these systems must either keep the entire set of models in memory or rely on *model switching* to load the required models at runtime [61]. As GPU memory remains the key resource bottleneck in both datacenter and edge [34, 42], these systems choose between **R3** – effectively utilizing the resources (by incurring the prohibitive latency penalties of switching models), or **R1** – meeting SLO targets under bursty request rates.

Conventional wisdom in inference serving literature touts the "*non-negligible provisioning time* [for ML models due to switching], *which can exceed the request processing times*" as a "*key characteristic of ML workloads*", and "*rules out reactive techniques*" for responding to bursty request rates [24]. This wisdom has been widely accepted [14, 25, 45] leading to the development of coarse-grained scheduling policies for inference serving that must account for the enormous latency penalty of switching models when reacting to bursty request rates. As a result, these coarse-grained policies typically avoid or minimize switching models by design [45], and are hence, unable to optimally navigate the tradeoff space between **R1**-**R3** under rapidly-changing, unpredictable request rates.

In this work, we challenge this conventional wisdom that forces a choice between **R1** and **R3**. We describe a mechanism, SubNetAct, to simultaneously serve the entire range of models spanning the latency-accuracy tradeoff space (**R1**-**R2**) in a resource-efficient manner (**R3**). At the core of our mechanism are novel control flow operators that SubNetAct carefully inserts into trained super-network [8, 47, 58] (SuperNet) neural architectures. SuperNets enable a fine-grained latency-accuracy tradeoff (**R1**-**R2**) by training a set of shared model weights for many neural networks, without duplication. Prior works [8, 47] propose efficient mechanisms for training SuperNets for both vision and NLP tasks, but require each model instance to be individually extracted for inference. This leads inference serving systems to a similar choice as before – either load all individual models or switch between them at runtime. However, SubNetAct's novel operators obviate the need to extract individual models and load them at runtime. Instead, SubNetAct dynamically routes requests within one SuperNet deployment with negligible overhead, enabling near-instantaneous *actuation* of different models. Thus, it unlocks orders of magnitude improvements in the navigation of the latency-accuracy tradeoff space (**R1**-**R2**), while substantially reducing the memory footprint (**R3**) (see §2).

In addition to being resource-efficient (**R3**), SubNetAct's agility in navigating the latency-accuracy tradeoff space (**R1**-**R2**) fundamentally changes the design space of scheduling policies. Instead of complex scheduling policies that must reason about future request rates in a bid to avoid paying the latency of switching ML models dynamically under bursts, SubNetAct enables the specification of simple policies that directly optimize for the key success metrics: **R1**-**R3**. While conventional wisdom deems such reactive policies infeasible, we explore one example point in this design space with a simple, yet effective policy that we call SlackFit. SlackFit is a reactive scheduling policy that exploits the near-instantaneous actuation property of SubNetAct to make fine-grained decisions about how many requests to serve in a batch, and which latency/accuracy choice to select for serving in real-time.

We summarize the contributions of this paper as follows:

- We introduce SubNetAct (§3), a novel mechanism that enables a resource-efficient, fine-grained navigation of the latency-accuracy tradeoff space. SubNetAct achieves this by carefully inserting novel control flow operators that dynamically route requests through a single SuperNet.
- We unlock the design of fine-grained, reactive scheduling policies and provide a mathematical formulation of their objective (§4.1). We then propose SlackFit (§4.2), a simple, yet effective greedy heuristic scheduling policy , and show how it accurately approximates the optimal objective.
- We instantiate SubNetAct and SlackFit in a real-world system, SuperServe, a real-time asynchronous model serving system with pluggable scheduling policies (§5).
- We evaluate SuperServe with both SlackFit and several state-of-the-art scheduling policies (§6). We find that SuperServe achieves 4.67% higher accuracy for the same SLO attainment and 2.85× higher SLO attainment for the same accuracy on the real-world Microsoft Azure Functions trace.

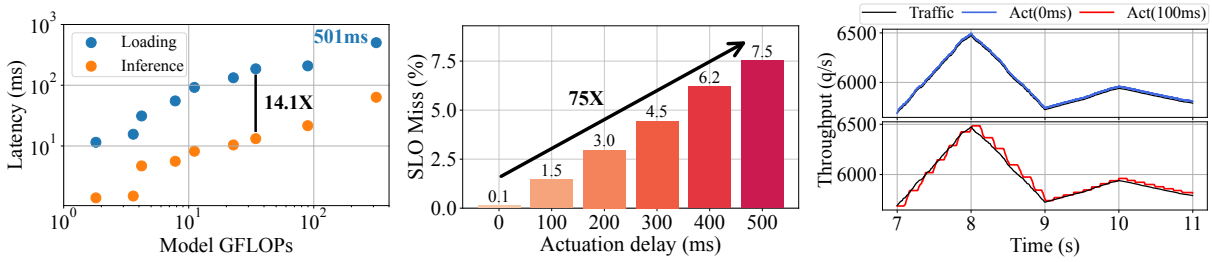
2 Motivation

In §2.1, we first motivate the need for a reactive, fine-grained scheduling policy for handling unpredictable, bursty request rates. To meet the SLO of the requests while maximizing the accuracy of the responses, we motivate the use of SuperNets (§2.2) that enable a fine-grained exploration of the latency-accuracy tradeoff space (**R1**-**R2**) for multiple latency targets.

2.1 Reactive, Fine-Grained Scheduling

Prior works in inference serving systems [34, 60] have exhaustively analyzed both production traces from Microsoft Azure Functions (MAF) [48] and synthetic application traces with a goal of highlighting their bursty request arrival patterns. For example, Zhang et al. [60] underscore the high coefficient of variance in request arrivals in production traces [48]. Further, the authors claim that the bursty "*sub-second request arrival patterns* [are] *nearly impossible to predict*", thus frustrating the goal of meeting the stringent SLO requirements (**R1**) of requests in an ML-based production applications.

A strawman solution to meeting SLOs under bursty request rates requires these systems to load the entire set of models



(a) Model switching is expensive **(b)** SLO misses wrt. actuation delay **(c)** Fine-grained scheduling

Fig. 1: Fine-grained scheduling policies are beneficial. (a) The latency of loading convolutional neural networks [27, 38, 59] and transformer-based neural networks [37] is greater than their inference latency across multiple batch sizes, making model switching expensive. This gap widens as model sizes increase, with a peak difference of up to 14.1×. (b) The higher actuation delay (due to model loading) leads to up to 75× higher SLO misses while serving the entire real-world, bursty MAF [48] trace. (c) A high actuation delay on a snapshot of the MAF trace shows 2% of the requests missing their SLO (**R1**) as request rates increase, and an inefficient utilization of resources (**R3**) as request rates decrease.

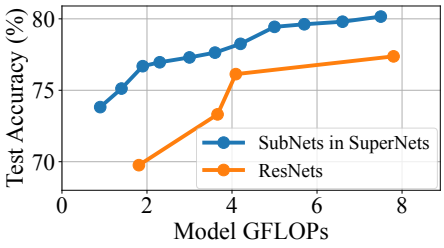


Fig. 2: Enhanced, fine-grained latency-accuracy tradeoff with SuperNets. The accuracy of SubNets extracted from OFAResNet SuperNet [8] is vastly superior to the hand-tuned ResNets from Fig. 1a for the same FLOP requirements (**R1**). Moreover, SuperNets can instantiate vastly higher number of points in the space.

spanning the latency-accuracy tradeoff space into GPU memory and switch between them as request rate fluctuates. While this reduces the latency of switching models, allowing serving systems to rapidly degrade accuracy (**R2**) under bursts to meet SLO targets (**R1**), it is resource-inefficient (**R3**).

As a result, state-of-the-art inference serving systems [45, 61] page models in and out when required, to efficiently utilize GPU memory (**R3**). However, Fig. 1a shows that the loading time of ML models into GPU memory is vastly more than the inference time, and the gap widens as the model sizes increase. Thus, reactive approaches that either provision resources or switch models upon arrival of bursty request rates must incur an *actuation delay* (i.e., the latency penalty of loading ML models) on the critical path of request serving, leading to an order-of-magnitude increase in missed SLOs (see Fig. 1b). In a bid to offset this latency penalty, inference serving systems rely on *predictive* scheduling policies that make coarse-grained estimations of future request arrival patterns [24]. Such policies are bound to be suboptimal owing to the difficulty of predicting the short bursts in request arrival rates coupled with their stringent SLO requirements [60].

We believe that the key to optimally serving bursty request rates instead lies in the ability to rapidly switch between ML models thus obviating the need for coarse-grained predictive scheduling policies. To validate our hypothesis, we simulate a coarse-grained policy with an actuation delay of 100ms and an ideal fine-grained policy with an actuation delay of 0ms. Fig. 1c plots the effects of these policies on a small bursty subtrace from the MAF trace. We observe that the coarse-grained policy leads to higher SLO misses (**R1**) under increasing request rates and wasted resources (**R3**) under decreasing request rates. Conversely, the fine-grained policy is able to instantaneously adjust to the request rates leading to no missed SLOs and effective utilization of the GPU.

2.2 Weight-Shared SuperNets

The problem of navigating the pareto-optimal frontier of the latency-accuracy tradeoff space (**R1**-**R2**) by finding highest accuracy ML models for a specific latency target is well studied in ML literature. Conventional Neural Architecture Search (NAS) [10, 36, 51, 63, 64] couple the search and training of ML models to produce a single architecture with the highest accuracy for a particular latency target. To satisfy multiple latency targets, these approaches must repeat the entire train and search procedure, thus becoming prohibitively expensive.

To solve this issue, recent NAS works [8, 28, 47, 58] decouple the search and training procedures by allowing multiple architectures to share their weights while training. These approaches first train one SuperNet and then extract subsets of its layers to form multiple *SubNets*, without requiring any further retraining. These SubNets are extracted to target vastly superior points in the latency-accuracy tradeoff space (**R1**-**R2**), as compared to hand-tuned ML models. For example, Fig. 2 highlights the accuracy benefits of SubNets extracted from a ResNet-based SuperNet when compared to the hand-tuned ResNets for an equivalent number of FLOPs.

SuperNets enable a fine-grained exploration of the latency-accuracy tradeoff space (**R1**-**R2**) by yielding specialized ML model architectures for a wide-variety of latency targets. This

is enabled by a search for multiple architectures, which relies on the following parameters: (*i*) *Depth* ($\mathbb{D}$), which describes the depth of a SubNet, and (*ii*) *Width Multiplier* ($\mathbb{W}$), which describes the width of a SubNet i.e., the number of heads in a multi-head attention layer or the number of channels in a convolution layer to be used. These parameters create a combinatorially large architecture space, Φ ($|\Phi| \approx 10^{19}$) [8], from which SubNets are extracted *statically* for inference. However, this static extraction in prior work [8, 28, 47, 58] again yields individual models that must either be simultaneously deployed (wasting resources; **R3**) or paged in as request rates fluctuate (missing SLOs; **R1**).

3 SubNetAct: Instantaneous Model Actuation

Motivated by §2, we seek to exploit SuperNets' fine-grained exploration of the latency-accuracy tradeoff to unlock the development of reactive scheduling mechanisms. To achieve our goal, we introduce SubNetAct, which addresses the challenges posed by static extraction of individual models in SuperNets, which force a choice between **R1** and **R3**.

Key Idea. To resolve this fundamental tension, we make the key observation that by virtue of performing architectural search *post* training, a SuperNet subsumes the entire architectural space of SubNets. Specifically, we follow the training procedure of prior NAS approaches [8, 28, 47, 58] to retrieve the architecture of the SuperNet, $\mathcal{M}$, along with its weights W[1]. The architecture of the SuperNet, $\mathcal{M}$, captures the set of all layers and weights that can be used in inference. Thus, instead of performing the search procedure and extracting individual SubNets, SubNetAct automatically modifies $\mathcal{M}$ to introduce its novel control flow operators. This allows SubNetAct to deploy the entire SuperNet $\mathcal{M}$, and *dynamically route* requests to the appropriate SubNet. §3.1 provides an overview of SubNetAct's novel control flow operators, and Appendix A.1 describes the procedure by which SubNetAct automatically inserts these operators into $\mathcal{M}$.

SubNetAct exploits the weight-sharing among the SubNets of a SuperNet to enable a memory-efficient model *actuation* mechanism (**R3**). Its fine-grained control flow operators near-instantaneously switch between SubNets in order to pick the optimal point in the latency-accuracy tradeoff space (**R1**-**R2**).

3.1 SubNetAct's Operators

SubNetAct's key insight lies in the introduction of the three novel operators (Fig. 3) that enable it to route requests to the required subnet dynamically and selectively use the trained supernet's weights. SubNetAct works on both convolution [8] and transformer-based [28] supernets. The three operators introduced in SubNetAct are as follows:

[1]We highlight that many NAS approaches make the trained $\mathcal{M}$ and W publicly available. Our evaluation uses these trained models, and does not require any further re-training.

LayerSelect operates at a block level in $\mathcal{M}$, where each block is a collection of multiple layers. The LayerSelect operator enforces control flow by either passing the input activation to the wrapped layers in the block or skipping the block and directly forwarding the input to the next block.

- *Convolution-Based SuperNet.* A convolution-based supernet [8] contains multiple stages, with each stage consisting of set of repeating blocks (bottleneck in OFAResNets [9]). The LayerSelect operator dynamically selects the first $\mathbb{D}_m$ blocks within m^{th} stage of the supernet. $\mathbb{D}_m$ is an external input provided to SubNetAct, as shown in Fig. 3 (2^{nd} row) for 2-stage supernet where $\mathbb{D}$ takes depth values per stage (m=2).
- *Transformer-Based SuperNet.* Unlike convolution-based supernets, transformer-based supernets [28] have a single stage. Hence, these supernets contain repeatedly stacked transformer blocks. The LayerSelect operator selects or deselects transformer blocks based on a single external input $\mathbb{D}$ that specifies the number of layers to use for inference. The block selection is based on the "every-other" strategy [17, 28]. In this strategy, n^{th} block is selected for inference if $n \mod \frac{L}{(L-\mathbb{D})} \equiv 0$, where L is the maximum number of stacked transformer blocks.

Overall, this operator enables layer-sharing among subnets that differ in depth ($\Phi_{\mathbb{D}} \subset \Phi$), which helps reduce GPU memory consumption (**R3**). For example, in a subnet with depth $\mathbb{D} = 1$, the blocks selected for execution by SubNetAct are shared with those in a subnet with depth $\mathbb{D} = 2$.

WeightSlice operates at each layer in a block of $\mathcal{M}$. For each layer, it dynamically selects the slice of the SuperNet's trained weights that participate in inference.

- *Convolution-Based SuperNet.* Each block in this supernet contains convolution layers. WeightSlice operator dynamically selects the number of channels of the convolution layer weights in a block (1^{st} row and 1^{st} column in Fig. 3). Precisely, the operator selects the first $\lceil \mathbb{W}_k * C_k \rceil$ channels in the k^{th} block (C_k represents the maximum channels). $\mathbb{W}_k$ is an input to the operator that is specified independently for each block. It denotes the fraction of channels to use in the weights of the convolution layers of the k^{th} block.
- *Transformer-Based SuperNet.* The blocks in this supernet consist of multi-head attention (MHA) and feed-forward layers [52]. WeightSlice operator dynamically selects the number of attention heads in the MHA layers (1^{st} row and 2^{nd} column in Fig. 3). Particularly, WeightSlice selects the first $\lceil \mathbb{W}_k * H_k \rceil$ heads, H_k represents the maximum heads in the k^{th} block. Here, $\mathbb{W}_k$ denotes the fraction of attention heads to use for inference in the k^{th} transformer block.

The operator enables partial layer-sharing among SubNets ($\Phi_{\mathbb{W}} \subset \Phi$,), thus increasing the number of available SubNet architectures. For example, for a SubNet with $\mathbb{W}_0 = 0.5$, WeightSlice selects the first 50% heads and shares its weights with the SubNet with $\mathbb{W}_0 = 0.75$. Together, LayerSelect and

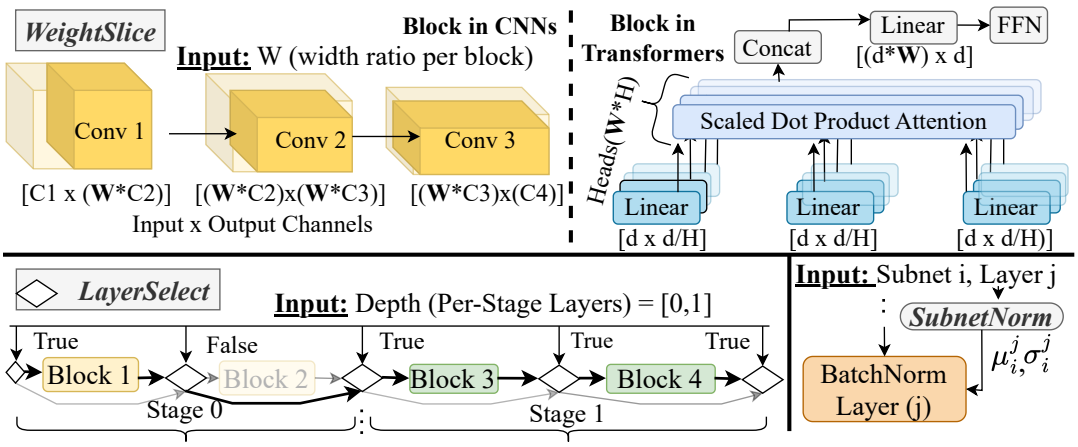


Fig. 3: SubNetAct's novel operators (LayerSelect, WeightSlice, SubnetNorm) dynamically actuate SubNets by routing requests through weight-shared layers and non weight-shared components.

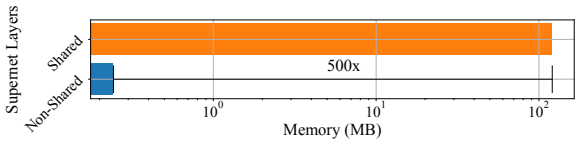


Fig. 4: SubNetAct's memory savings. The memory used by the normalization statistics is 500× smaller than the non-normalization layers. SubnetNorm decouples the normalization statistics for each SubNet and provides accurate bookeeping thus enabling high accuracy (**R1**), with minimal increase in memory consumption (**R3**).

WeightSlice enable SubNetAct to dynamically actuate the entire set of latency-accuracy options (**R1**-**R2**).

SubnetNorm is specifically implemented only for *convolution-based SuperNets*. We observe that naively introducing the LayerSelect or WeightSlice operator leads to a significant drop in SubNet accuracy (as low as 10%) in the convolution-based SuperNet. This is due to the incorrect tracking of the mean (μ) and variance (σ) in normalization layers such as BatchNorm [29]. The transformer-based SuperNet uses LayerNorm [7] which doesn't require tracking of mean and variances, and hence doesn't face this issue. To account for the discrepancy in BatchNorm layers, SubNetAct introduces the SubnetNorm operator that precomputes and stores μ and σ for each possible SubNet by performing forward pass inference on the training data. SubnetNorm takes as input a unique SubNet ID (i) and a layer ID (j) and outputs the pre-computed normalization statistics $\mu_{i,j}$ and $\sigma_{i,j}$. The layer j then uses these statistics to perform normalization of activations, effectively specializing j for each SubNet i.

Although this bookkeeping increases the memory requirements of deploying the SuperNet, Fig. 4 shows that the overhead of these non-shared normalization statistics is 500× smaller than the memory requirement of the shared layers. SubNetAct can host thousands of SubNets in memory by only keeping the statistics unique to each subnet and sharing the non-normalization weights amongst all the SubNets.

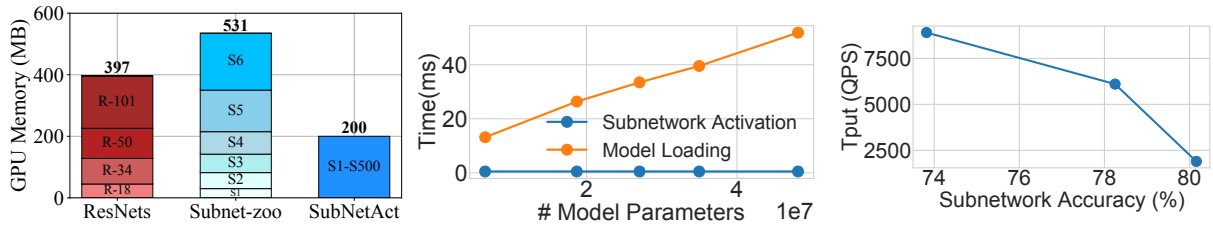
Finally, we note that the input to these control flow operators (i.e., $\mathbb{D}$, $\mathbb{W}$) remains similar to the inputs for architectural search in NAS [8]. Moreover, these inputs are independent from the request served by the actuated SubNets, and are declaratively specified by a scheduling policy (§4). Given the arrival rate, the scheduling policy chooses a specific SubNet for a request (by specifying the control tuple $\mathbb{D}$ and $\mathbb{W}$), which is then actuated by SubNetAct near-instantaneously.

3.2 Discussion: Efficacy of SubNetAct

We now highlight SubNetAct's efficacy in achieving key application requirements (**R1**-**R3**) under bursty request rates.

Reduced Memory Requirements. SubNetAct's novel operators enable SubNets to share layers in place and dynamically route requests to the appropriate SubNet based on the control tuple $(\mathbb{D}, \mathbb{W})$. Thus, SubNetAct can simultaneously serve the entire range of models spanning the latency-accuracy tradeoff space while drastically reducing memory requirements. Fig. 5a demonstrates the requirements of serving the same accuracy range by comparing: (*i*) four different hand-tuned ResNets [27] (publicly available from prior literature), (*ii*) six uniformly sampled individual SubNets [8] from a ResNet-based SuperNet, and (*iii*) SubNetAct that enables dynamic actuation of 500 SubNets. We highlight that SubNetAct reduces memory usage by up to 2.6×, while serving vastly more fine-grained latency-accuracy tradeoff points.

Near-Instantaneous Model Actuation. While switching between individual models requires loading their weights to the GPU, SubNetAct's operators enable scheduling policies to actuate any SubNet *in place* without incurring additional loading overhead. Fig. 5b compares the time taken to perform on-demand loading of individual SubNets versus in-place actuation of a SubNet in SubNetAct. We highlight that SubNetAct's model actuation is orders-of-magnitude faster than



(a) Reduced Memory Requirements. **(b)** Instantaneous Model Actuation. **(c)** High dynamic throughput range.

Fig. 5: Efficacy of SubNetAct. (a) SubNetAct requires upto 2.6× lower memory to serve a higher-range of models when compared to the ResNets from Fig. 1a and six individual SubNets extracted from SuperNet [8] (b) SubNetAct actuates different SubNets near-instantaneously (< 1ms), which is orders of magnitude faster than the model switching time. (c) SubNetAct's instantaneous actuation of models enables it to sustain higher ingest rates thus inducing a wide dynamic throughput range ($\approx 2-8k$ queries per second) within a narrow accuracy range.

on-demand loading of ML models. This allows scheduling policies that use SubNetAct to rapidly actuate lower-accuracy models under bursty conditions (**R1**) and switch to higher-accuracy models under normal load (**R2**), without coarse-grained predictions about future request rates.

Increased Throughput & Accuracy. Through its instant model actuation, SubNetAct allows scheduling policies to rapidly scale the throughput of the system, thus inducing a broad throughput range within a narrow range of accuracy to help meet SLOs (**R1**-**R2**). Fig. 5c compares the maximum sustained ingest throughput for a point-based open-loop arrival curve for serving the largest, smallest, and a median SubNet on 8 GPUs. We observe that SubNetAct can serve a wide throughput range from 2000-8000 QPS, while being able to increase accuracy between 74% to 80%.

4 Fine-Grained Scheduling Policies

SubNetAct's resource-efficient (**R3**) near-instantaneous actuation of the entire latency-accuracy tradeoff space unlocks the development of fine-grained, reactive scheduling policies. These policies can quickly scale an inference serving system's throughput upon arrival of bursty request rates to ensure that the requests meet their SLO (**R1**) with the maximum possible accuracy (**R2**). Specifically, upon a *query*'s[2] arrival to an inference serving system, it invokes the scheduling policy, which must decide the following four decision variables:

- **SubNet** ϕ from the set of all possible SubNets Φ available for actuation by SubNetAct. As discussed in §3, a SubNet $\phi \in \Phi$ is uniquely identified by the control tuple $(\mathbb{D}, \mathbb{W})$.
- **Batch** B of size $|B|$ which groups the queries that are executed together on a GPU using the SubNet ϕ.
- **GPU** n upon which the batch B is executed.
- **Time** t at which the batch B must be executed on GPU n.

In this section, we first start with the mathematical formulation of an optimal scheduling policy that decides the above parameters (§4.1). We then describe our proposed policy SlackFit that approximates the optimal policy and aims to achieve both high accuracy and SLO attainment (§4.2).

[2]We use the term *query* and *request* interchangeably in this paper.

4.1 Optimal Scheduling Policy

We formulate an optimal scheduling policy with an oracular knowledge about all queries as a Zero-One Integer Linear Program (ZILP). The policy's decision is captured by the variable $I(\phi,B,n,t) \in \{0,1\}$, which represents the decision to execute all queries $q \in B$ (from the set of all possible batches $\mathcal{B}$) on GPU n at time t with the SubNet ϕ. The SubNet ϕ has an accuracy $Acc(\phi)$ and a latency profile $l_\phi(|B|)$, which is the latency of ϕ on the batch size $|B|$. We use $a(B)$ to refer to the earliest arrival time of all the queries $q \in B$, and $d(B)$ to refer to the earliest deadline. Intuitively, the policy's goal is to maximize the accuracy of the responses (to queries) within their SLO (**R1**-**R2**). This is represented in ZILP as follows:

$$\text{maximize} \sum_t \sum_n \sum_{\phi\in\Phi} \sum_{B\in\mathcal{B}} \text{Acc}(\phi)\cdot|B|\cdot I(\phi,B,n,t) \quad (1)$$

$$\text{s.t.} \sum_t \sum_n \sum_{\phi\in\Phi} \sum_{\{B|q\in\mathcal{B}\}} I(\phi,B,n,t) \le 1, \quad \forall q \quad (1a)$$

$$\sum_{B\in\mathcal{B}} \sum_{\{t'\le t\le t'+l_\phi(|B|)\}} I(\phi,B,n,t') \le 1, \quad \forall n,t,\phi \quad (1b)$$

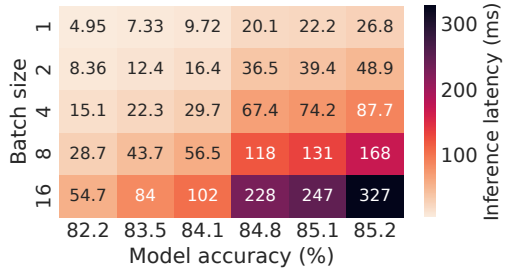
$$a(B)\cdot I(\phi,B,n,t) \le t, \quad \forall n,t,B,\phi \quad (1c)$$

$$\sum_{\phi\in\Phi} I(\phi,B,n,t) \le 1, \quad \forall n,t,B \quad (1d)$$

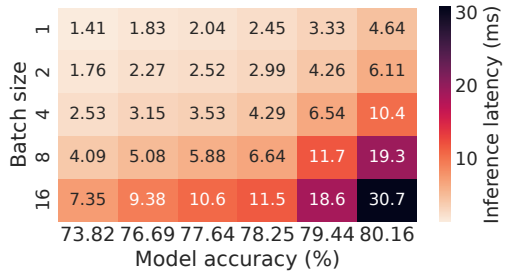
$$\sum_{\phi\in\Phi} (l_\phi(|B|)+t)\cdot I(\phi,B,n,t) \le d(B), \quad \forall n,t,B \quad (1e)$$

$$I(\phi,B,n,t) \in \{0,1\}, \quad \forall n,t,B,\phi \quad (1f)$$

The ZILP maximizes the number of queries that satisfy their latency SLOs with the highest possible accuracy across all the selected query batches i.e., $\forall(\phi,B) : I(\phi,B,n,t) = 1, Acc(\phi)\cdot|B|$ is maximized. The constraints of the ZILP denote:



(a) Transformer-Based SuperNet



(b) Convolution-Based SuperNet

Fig. 6: SlackFit control parameter space. Latencies for six different (uniformly sampled w.r.t. FLOPs) pareto-optimal SubNets in SubNetAct as a function of accuracy (x-axis) and batch size (y-axis) shown for transformer and convolution-based SuperNet. The latency increases monotonically with batch size (**P1**) and accuracy (**P2**).

(1a) A query q can be assigned to at most one batch B.

(1b) A GPU n can only execute a single SubNet ϕ on a single batch B at a particular time t.

(1c) Batch B can only execute after its arrival time $a(B)$.

(1d) Each batch B can be served with a maximum of one SubNet ϕ on a GPU n at a time t.

(1e) The batch B should complete before its deadline $d(B)$.

(1f) The choice variable $I(\phi, B, n, t)$ is a boolean indicator.

Given that solving the above ZILP is NP-Hard [43, 60] and it is impractical to expect oracular query arrival knowledge, it cannot be used to serve models online. Instead, we approximate its behavior with a heuristic, *online* scheduling policy.

4.2 SlackFit: Online Scheduling Policy

We introduce SlackFit—a simple yet effective scheduling policy that aims to maximize the accuracy (**R1**) with which the requests meet their latency SLO (**R2**). SlackFit approximates the ILP-based policy in Eq. (1) and makes the decision making tractable by operating in two phases:

Offline Phase triggered upon the registration of a SuperNet $\mathcal{M}$ modified by SubNetAct (Appendix A.1) that the inference serving system must serve. This phase consists of two stages:

1. **Profile pareto-optimal SubNets Φ_{pareto}:** To make SubNet choices in reasonable time, SlackFit makes the design decision to operate on Φ_{pareto} instead of Φ. Φ_{pareto} is the set of pareto-optimal SubNets w.r.t. latency and accuracy obtained by using the search stage of prior NAS methods [8][3]. The size of $|\Phi_{\text{pareto}}| \approx 10^3$ is orders of magnitude smaller than $|\Phi| \approx 10^{19}$, contributing to rapid scheduling decisions.
2. **Bucketize SubNet ϕ and batch size $|B|$ choices:** As discussed in §4, SlackFit must decide the SubNet $\phi \in \Phi_{\text{pareto}}$ and the batch size $|B|$ for the incoming queries. To reduce the search space for this decision, SlackFit relies on three key properties of SubNets in Φ_{pareto} (visualized in Fig. 6): **P1**: the latency increases monotonically with batch size as observed by prior works [13, 14, 25], **P2**: the latency increases monotonically with accuracy due to the choice of SubNets retrieved by NAS in Φ_{pareto}, and **P3**: lower accuracy SubNets can serve higher batch sizes at similar latencies to lower batch sizes in higher accuracy SubNets, due to the SubNet's FLOPs distribution shown in Appendix A.2.

These properties enable SlackFit to reduce the search space of choices for ϕ and $|B|$ to a single dimension – *batch latency*. Thus, SlackFit constructs evenly-spaced buckets between the minimum and maximum latency of all SubNets in Φ_{pareto} (i.e., $l_{\phi_{min}}(|B|=1)$ and $l_{\phi_{max}}(|B|=16)$ respectively, where ϕ_{min} and ϕ_{max} are the lowest and highest accuracy SubNets; using properties **P1**-**P2**). Within each bucket, SlackFit chooses the $(\phi', |B'|)$ with the highest $|B'|$ such that $l_\phi(|B'|)$ is less than the bucket's latency. By **P3**, low latency buckets contain lower accuracy ϕ, higher $|B|$ (leading to higher throughput), and higher latency buckets contain higher accuracy ϕ and lower $|B|$ (leading to lower throughput).

Online Phase of SlackFit is triggered upon the arrival of queries or the availability of a GPU n to the serving system. The key insight of the online phase is that the remaining slack of the query with the earliest deadline provides a proxy to changes in the traffic. Specifically, bursts in traffic increase queuing delays which reduces the available slack, while slack remains high under normal conditions.

Thus, SlackFit chooses a bucket $(\phi, |B|)$, where $l_\phi(|B|)$ is closest to but less than the slack of the query with the earliest deadline. It then packs $|B|$ queries with the earliest deadline into a batch B and executes it on an available GPU n at time t.

By making decisions based on the minimum remaining slack, SlackFit can automatically adjust accuracy (**R2**) and throughput of the system by choosing appropriate latency bucket on variable arrival traffic to maintain high SLO attainment (**R1**). Under normal conditions, a higher slack leads to the choice of buckets with higher $l_\phi(|B|)$, which is strongly correlated with the choice of higher accuracy models (**P2**). Conversely, bursty request arrivals lead to buckets with lower $l_\phi(|B|)$, as SlackFit operates under reduced latency slack. These buckets maximize $|B|$ (due to **P3**), thus opportunistically maximizing accuracy while satisfying SLO.

[3] It takes $\leq$ 2 minutes to perform this NAS profiling on SuperNets.

4.2.1 SlackFit's Approximation of Optimal Offline ZILP

We now provide insights on how SlackFit emulates behavior of the optimal offline ZILP. To understand the behavior of ZILP, we formulate a proxy utility function that captures the inner-term of the ZILP objective function in Eq 1, the utility function is defined for a SubNet ϕ, batch size $|B|$ and the earliest deadline d_B among all queries:

$$\mathbb{U}(\phi,|B|,d_B) = \begin{cases} \text{Acc}(\phi)\cdot|B|, & \text{if } l_\phi(|B|) < d_B \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

This utility is non-zero iff SubNet ϕ performs inference on batch size $|B|$ within the deadline d_B, and is zero otherwise. This maximizes both the number of queries processed within their SLO (**R1**) and the accuracy of their responses (**R2**).

A. ZILP and SlackFit prefer pareto-optimal SubNets. SlackFit's key design choice is to operate on pareto-optimal SubNets w.r.t. latency, accuracy (Φ_{pareto}) (§4.2). We claim that the ZILP also tends towards pareto-optimal SubNets , as these SubNets yield higher utility.

Lemma 4.1. *The utility of pareto-optimal SubNets is higher than non pareto-optimal SubNets if they have similar inference latency for a batch of queries.*

$$\mathbb{U}(\phi_p,|B|,d_B) > \mathbb{U}(\phi_q,|B|,d_B), \qquad \forall B, d_B$$
$$s.t.\ \ \phi_p \in \Phi_{pareto},\ \phi_q \in \{\Phi \setminus \Phi_{pareto}\},\ l_{\phi_p}(|B|) \approx l_{\phi_q}(|B|)$$

This validates SlackFit's design choice to operate on pareto-optimal SubNets only.

Proof By Contradiction. Assume a non-pareto optimal subnet (ϕ_q) such that it has higher utility than pareto optimal subnet (ϕ_p) for a batch B and $l_{\phi_p}(B) \approx l_{\phi_q}(B)$ i.e., $\mathbb{U}(\phi_p,B,d_B) < \mathbb{U}(\phi_q,B,d_B)$. Now, due to the pareto optimal property $Acc(\phi_p) > Acc(\phi_q)$, this implies $Acc(\phi_p)\cdot|B| > Acc(\phi_q)\cdot|B|$ which implies $\mathbb{U}(\phi_p,B,d_B) \geq \mathbb{U}(\phi_q,B,d_B)$ for any delay d_B as $l_{\phi_p}(B) \approx l_{\phi_q}(B)$. This is contradiction. Hence Proved.

B. ZILP and SlackFit prioritize lower accuracy & higher batch sizes under bursts. We make a key observation that the utility of a lower accuracy, higher batch size $(\phi_{low}, |B_{high}|)$ configuration is higher than a higher accuracy, lower batch size $(\phi_{high}, |B_{low}|)$ configuration in Φ_{pareto}.

This is because the factor difference in accuracy of SubNets in Φ_{pareto} (< 1) is less than the factor differences of batch sizes as seen in Fig. 6 i.e., $\frac{\text{Acc}(\phi_{high})}{\text{Acc}(\phi_{low})} \leq \frac{|B_{high}|}{|B_{low}|} \Rightarrow \text{Acc}(\phi_{high})\cdot|B_{low}| \leq \text{Acc}(\phi_{low})\cdot|B_{high}|$. Therefore, $\mathbb{U}(\phi_{low},|B_{high}|,d_q) \geq \mathbb{U}(\phi_{high},|B_{low}|,d_q)$ may hold true under bursts, in cases where the query q with the earliest deadline in a batch of k queries ($q \in B_k$) can be served either by: a) low accuracy model (ϕ_{min}) with batch size $|B_k|$ or b) higher accuracy model (ϕ_{max}) on a subset of queries (say m, $q \in B_m$) with remaining queries ($B_k \setminus B_m$) missing the deadline due to high load. In such cases, the optimal offline ILP will tend to option (a), similar to SlackFit.

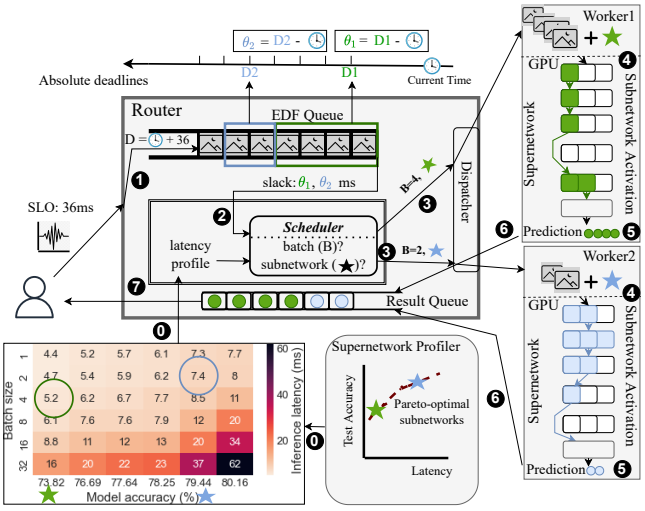


Fig. 7: SuperServe's Architecture comprises of a SuperNet profiler, a router, a fine-grained scheduler (SubNetAct), and GPU-enabled workers. Clients register SuperNets for SuperServe to serve, whose profiling and insertion of control-flow operators is done before queries arrive. Clients submit queries to the router with a specified SLO asynchronously. The query follows the critical path ❶ - ❼.

C. ZILP and SlackFit prefer higher accuracy under normal conditions. The utility of serving more requests with a high-accuracy subnet and fewer requests with a low-accuracy subnet is more than serving all the requests with a medium-accuracy subnet.

We make another observation from the latency profiles of SubNets from Φ_{pareto} in Fig. 6. For a batch size $|B|$, such that $|B| = |B_1| + |B_2|$ where $|B_1| > |B_2|$, the following holds true in many cases - $B_1\cdot\text{Acc}(\phi_{high}) + B_2\cdot\text{Acc}(\phi_{low}) > B\cdot\text{Acc}(\phi_{mid})$. Therefore, $\mathbb{U}(\phi_{high},B_1,d_q) + \mathbb{U}(\phi_{low},B_2,d(B_2)) \geq \mathbb{U}(\phi_{mid},B,d_q)$, may hold true under low load, where the query q in batch B can be served by either: a) mid accuracy model (ϕ_{mid}) with batch size B, or b) high accuracy model (ϕ_{high}) with larger partition B_1 ($q \in B_1$) with rest of the queries in batch B_2 served with the low accuracy model (ϕ_{low}) and meeting deadline $d(B_2)$. In such cases, ILP will tend to option (b) i.e., an option with higher average accuracy, simiar to SlackFit (as described in §4.2).

5 SuperServe: System Implementation

SuperServe is a system that instantiates both the SubNetAct mechanism and the SlackFit policy. SuperServe's architecture is illustrated in Fig. 7. Clients first register the SuperNet that they want SuperServe to serve, which invokes SubNetAct to automatically insert the control flow operators for dynamic actuation of SubNets (Appendix A.1). SuperServe then profiles the SuperNet to enable SubNetAct to operate on the pareto-optimal SubNets.

SuperNet Profiler. The profiler employs neural architecture search (NAS) [8] to find pareto-optimal SubNets from the Su-

perNet for each latency target. The latency of each SubNet is a function of the batch size and the environment of execution (i.e., the GPUs on the available workers). NAS and the model profiling is efficient, taking ≤ 2 minutes to complete, and providing significant benefits for the online phase of SlackFit.

Post SuperNet registration, the clients submit queries to the SuperServe router with a deadline via RPC asynchronously. These queries are enqueued to a global earliest-deadline-first (EDF) queue (❶). As soon as any worker becomes available, SuperServe's fine-grained scheduler is invoked (❷). It decides on the query-batch (B) and the subnet (ϕ) which are then dispatched to the worker (❸). Upon receiving this query-batch, the worker that instantiates the supernet instantaneously actuates the chosen subnet in-place on the GPU using SubNetAct (❹), performs inference (❺), and returns predictions for the query-batch (❻). The router redirects these predictions to the client (❼). We discuss the components of SuperServe below:

Router. The router runs the fine-grained scheduling policy and interacts with workers via RPCs. All queries are received, enqueued, and dequeued asynchronously in the router. It maintains pending queries in a global EDF queue, ordered by query deadlines (SLOs). The router invokes the scheduler whenever (a) a worker signals availability and (b) the EDF queue is not empty. It sends query-batches decided by the scheduler to workers and returns the predictions to the clients.

Fine-grained Scheduler. The scheduler's control decision is a batch-size and subnet ($\phi = (\mathbb{D}, \mathbb{W})$). The scheduler provide pluggable APIs for different policy implementations. SlackFit is one such policy implemented in the scheduler. All policies in scheduler leverage two key properties to make control decisions: (a) predictability of DNN inference latency, (b) fast actuation of SubNetAct on the query's critical path.

Worker. The DNN worker employs the SubNetAct mechanism to host a SuperNet (**R3**). SubNetAct's operators are implemented in TorchScript's intermediate representation (IR) [16]. After receiving a query-batch and subnet ($\mathbb{D}$, $\mathbb{W}$) from the router, the worker actuates the desired SubNet using SubNetAct. A forward pass on the actuated SubNet produces predictions that are returned to the router.

6 Evaluation

We assess SuperServe's end-to-end performance i.e., its ability to maximize SLO attainment (**R1**) and accuracy (**R2**) under a variety of traffic conditions, on a real-world Microsoft Azure Functions (MAF) trace (§6.2) and synthetic traces (§6.3). SuperServe is resource-efficient (**R3**) due to the use of SubNetAct mechanism, already established in §3.2. We conclude with microbenchmarks (§6.4).

6.1 Experimental Setup

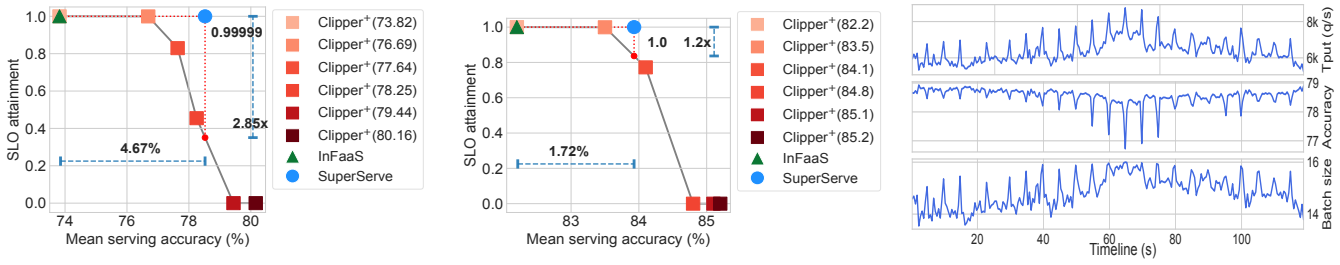
Success metrics. *SLO attainment* is the fraction of queries that finish within the deadline (**R1**). *Mean serving accuracy* is calculated for queries meeting their SLO and is the average of models' profiled accuracy used to serve them (**R2**).

Traces. We evaluate SuperServe on three sets of traces: real-world, bursty and time-varying. MAF [48] provides a real-world trace. Bursty and time-varying traces are synthetic, similar to those used in InferLine [13]. *Bursty traces* have a base traffic with mean ingest rate λ_b ($CV^2 = 0$) and a variant traffic with mean ingest rate λ_v drawing inter-arrival times from a gamma distribution (Fig. 13a). We vary λ_b, λ_v and CV^2. *Time-varying* traces differ from bursty by varying the mean ingest rate over time. We change the mean from $\mu = 1/\lambda_1$ to $\mu = 1/\lambda_2$ at rate τ q/s^2 with a fixed CV_a^2. Higher ingest acceleration τ q/s^2 denotes faster change from $\lambda_1 \rightarrow \lambda_2$.

Baselines. We compare SuperServe with the single model serving systems that don't perform accuracy trade-offs (and the models are manually selected by users, non-automated serving systems in §7). These systems are represented as Clipper$^+$ baseline and include systems like Clipper [14], Clockwork [25], and TF-serving [40]. Clipper$^+$ is manually configured to serve six different accuracy points (SubNets) that uniformly span the SuperNet's accuracy range and result in its six different versions. We also compare SuperServe with INFaaS and note that INFaaS is designed to "pick the most cost-efficient model that meets the [specified] accuracy constraint" [18,45,46]. However, in the presence of unpredictable, bursty request rates, the choice of the model accuracy to serve in order to meet the SLO requirements is unknown. Since, unlike SuperServe, INFaaS does not automatically discover the accuracy of the model to serve under unpredictable request rates and instead requires queries to be hand-annotated with accuracy thresholds, we choose to run INFaaS with no accuracy thresholds provided (§6.3.1,§6.3.2). In such a scenario, it reduces to serving the most cost-efficient model (which is the model with the minimum accuracy). We confirmed this behavior with the INFaaS authors, who agree that "[our] representation of INFaaS as a baseline that always chooses the same model is correct in the absence of an accuracy threshold, or a fixed (never changing) accuracy threshold." [18].

Subnet-Profiling. We use a ResNet-based SuperNet trained on ImageNet [15] and Transformer-based SuperNet trained on MNLI [55]. We extract pareto-optimal SubNets (Φ_{pareto}) by running NAS (publicly released by [8]) on the trained SuperNet. The pareto-optimal SubNets in the ResNet-based and Transformer-based SuperNet span accuracy ranges of $73-80\%$ and $82-85\%$ respectively. The SubNets are profiled with varied batch sizes on Nvidia RTX2080Ti.

Test bed. SuperServe is implemented in 17.5k lines of C++. gRPC [23] is used for communication between the client, the router and workers. The experiments use 8 RTX2080Ti GPUs and 24 CPU cores, with each worker assigned one GPU.



(a) Serving CNNs on MAF Trace **(b)** Serving Transformers on MAF Trace **(c)** SuperServe System Dynamics

Fig. 8: SuperServe on Real World Trace. SuperServe on Microsoft Azure Functions (MAF) [48] trace. (a, b) SuperServe achieves the best tradeoffs w.r.t. SLO attainment and accuracy when serving both Convolution and Transformer neural networks. (c) System dynamics w.r.t. batch size and subnet activation control decisions over time in response to ingest rate in (a).

6.2 End-to-End: Real Workloads

We investigate if: (a) SlackFit is capable of achieving a better trade-off between SLO attainment and mean serving accuracy on real workloads (**R1-R2**), and (b) SubNetAct contributes to serve highly unpredictable workloads at high SLO attainment.

We use the MAF trace [48] to evaluate SuperServe (similar to Clockwork [25]). The trace is collected on Microsoft's serverless platform and serves as a reasonable workload to evaluate SuperServe as serverless ML inference is an active research area [30, 57]. It consists of number of invocations made for each function per minute and contains nearly 46,000 different function workloads that are bursty, periodic, and fluctuate over time. We use 32,700 function workloads from the MAF trace. The 24-hour-long trace is shrunk to 120 seconds using *shape-preserving* transformations to match our testbed. We use the mean ingest rate of 6400 qps and 1150 qps to serve convolution and transformer networks respectively on MAF.

Result. Fig. 8a compares SuperServe with Clipper$^+$ and InFaaS when convolution and transformer networks are served on the real-world MAF trace. While serving CNNs, SuperServe achieves an SLO attainment (**R1**) of 0.99999 (five '9's). Compared to Clipper$^+$ and InFaaS , SuperServe is 4.65% more accurate (**R2**) at the same level of SLO attainment. It also achieves a 2.85x factor improvement in SLO attainment at the same mean serving accuracy. While serving transformers, SuperServe achieves a 1.2x factor improvement in SLO attainment at the same mean serving accuracy or 1.72% higher accuracy at the same SLO attainment level.

System Dynamics. Fig. 8c shows the ingest throughput (qps), serving accuracy, and batch size control decisions made by SlackFit for serving CNNs on the MAF trace. As seen in the figure, the trace contains periodic short-interval spikes that reach upto 8750 qps, demonstrating the agility of the system. SlackFit selects both smaller accuracy model and higher batch size during the load spikes to meet the deadline (**R1**). SlackFit makes such control decisions because it uses query's slack as a signal to maximize batch size. As the query slack decreases, it selects maximum batch size control parameters in the lower latency buckets. Furthermore, these control decisions increase the system throughput instantly through SubNetAct. Lastly, SlackFit serves higher accuracy models when the ingest rate is low and hence, achieves better mean serving accuracy (**R2**).

6.3 End-to-End: Synthetic

We aim to answer the following questions, whether SuperServe (a) *automatically* serves queries using appropriate models (accuracy) for different traces (**R2**), (b) achieves a better trade-off w.r.t. the success metrics (**R1-R2**), (c) withstands sharp bursts while maintaining high SLO attainment (**R1**) and (d) *instantaneously* changes system throughput. To answer these questions, we evaluate SuperServe by serving CNNs on the bursty and time-varying traces (§6.1).

6.3.1 Baseline comparison with burstiness

Fig. 9 compares SuperServe with the baselines over a range of traces increasing mean ingest rate λ_v across and CV_a^2 down. All traces are configured with 36ms SLO. Achieving high SLO attainment (**R1**) and high mean serving accuracy (**R2**) is desirable, which implies the *best trade-off is in the top-right corner of the graph*. We demonstrate that no single choice of a model is *sufficient* for different mean arrival rates and CV_a^2. For instance, the SLO attainment of Clipper$^+$(76.69) decreases as the CV_a^2 increases for $\lambda_v = 5550$ (row 3). Similarly, the SLO attainment of Clipper$^+$(78.25) decreases with increase in λ_v for $CV_a^2 = 2$ (column 1). We draw the following takeaways: **(1)** SuperServe achieves a significantly better trade-off between SLO attainment and accuracy (**R1-R2**) than the baselines (Clipper$^+$ and InFaaS). It is 4.33% more accurate than the baselines at an SLO attainment level of 0.9999 and 2.06x higher SLO attainment at the same accuracy level. SuperServe is consistently at the top-right corner in Fig. 9 across all the traces. **(2)** SlackFit *automatically* selects appropriate models for sustaining different traffic conditions. As λ_v increases, SuperServe reduces serving accuracy while maintaining high SLO attainment (columns).

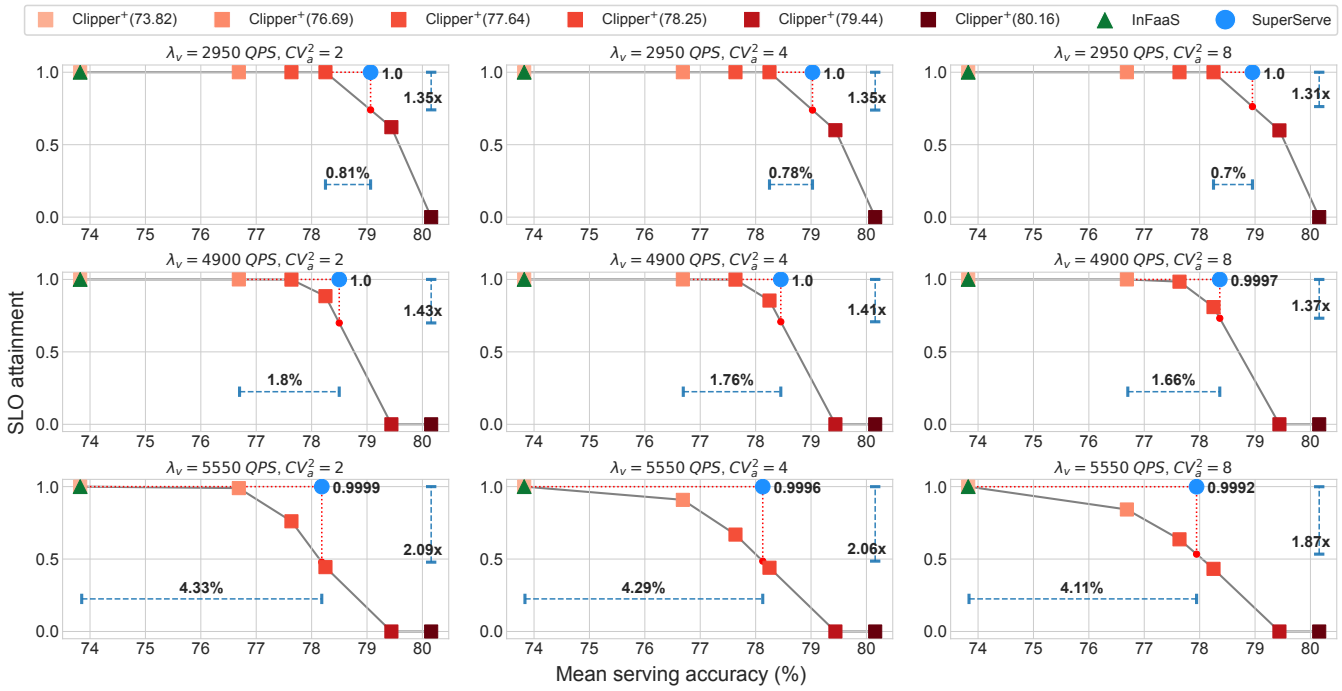


Fig. 9: SuperServe with variable burstiness. SuperServe outperforms Clipper$^+$ and INFaaS baselines by finding better tradeoffs and consistently achieving > 0.999 SLO attainment on bursty traces. Variable ingest rate $\lambda_v = \{$ 2950, 4900, 5550$\}$ q/s increases vertically (down). $CV_a^2 = \{2,4,8\}$ increases horizontally (across). SuperServe achieves a better trade-off in SLO attainment (y-axis) and mean serving accuracy (x-axis) in all cases. SuperServe consistently achieves high SLO attainment > 0.999.

Note that, across all the traces, InFaaS achieves an optimal SLO attainment but with a significantly smaller mean serving accuracy (by up to 4.33%) than SuperServe. InFaaS's policy serves the min-cost (and hence min accuracy) model for the trace without accuracy constraints. Whereas, SuperServe achieves a better trade-off between the success metrics because (a) SubNetAct allows in place activation of different subnetworks *without* affecting SLO attainment (**R1**); (b) SlackFit opportunistically selects higher accuracy models based on query's slack (**R2**). Also, the difference between SuperServe and Clipper$^+$ narrows w.r.t. accuracy as CV_a^2 increases, since SlackFit switches to lower accuracy models more frequently with burstier traffic. The system dynamics are detailed in Appendix A.3.

6.3.2 Baseline comparison with arrival acceleration

Fig. 10 evaluates SuperServe's performance at different levels of arrival rate change (i.e., arrival *acceleration*). Traces start at λ_1 and increase to λ_2 with acceleration τ. Traces fix $\lambda_1 = 2500 qps$ and $CV_a^2 = 8$ but change λ_2 and acceleration τ .

The τ and λ_2 are chosen to demonstrate that single, pre-configured model choices are inadequate to sustain different rates of arrival (mean λ) and acceleration (τ). Clipper$^+$(79.44) starts diverging as τ increases (λ_2 is 6800 qps (row 2)). Similarly, Clipper$^+$(79.44) starts diverging with increase in λ_2 ($\tau = 250\ q/s^2$ (column 1)). The key takeaways are:

- SuperServe *rapidly* scales system throughput and achieves a high SLO attainment (0.991-1.0) even with high values of τ (5000 q/s^2). The experiment demonstrates two key properties of SuperServe— (a) the *actuation delay* in SuperServe is indeed negligible, (b) the lower *actuation delay* helps achieve higher SLO attainments for time-varying traces (**R1**). SuperServe empirically demonstrates "agile elasticity" (§2), and withstands high acceleration in arrival rate (τ).
- SlackFit *dynamically* adjusts the serving accuracy over time (**R2**) and achieves a better trade-off between success metrics (**R1-R2**). When the mean ingest throughput is low (λ_1), SuperServe uses higher accuracy models. It quickly switches to lower accuracy models when mean arrival rate is high (λ_2), as evident in system dynamics Fig. 13b.

Fig. 10 experiments exhibit interesting trends. As the τ increases, the gap between SuperServe and Clipper$^+$ w.r.t. mean serving accuracy narrows. This is because SlackFit selects smaller accuracy sooner with the increase in τ. Lower τ values give enough time to SuperServe to serve intermediate mean arrival rates with higher accuracy models while gradually moving to lower accuracy models as mean ingest rate increases to λ_2 qps. Whereas, InFaaS continues to serves min accuracy model for all traces as its policy doesn't maximize accuracy by design.

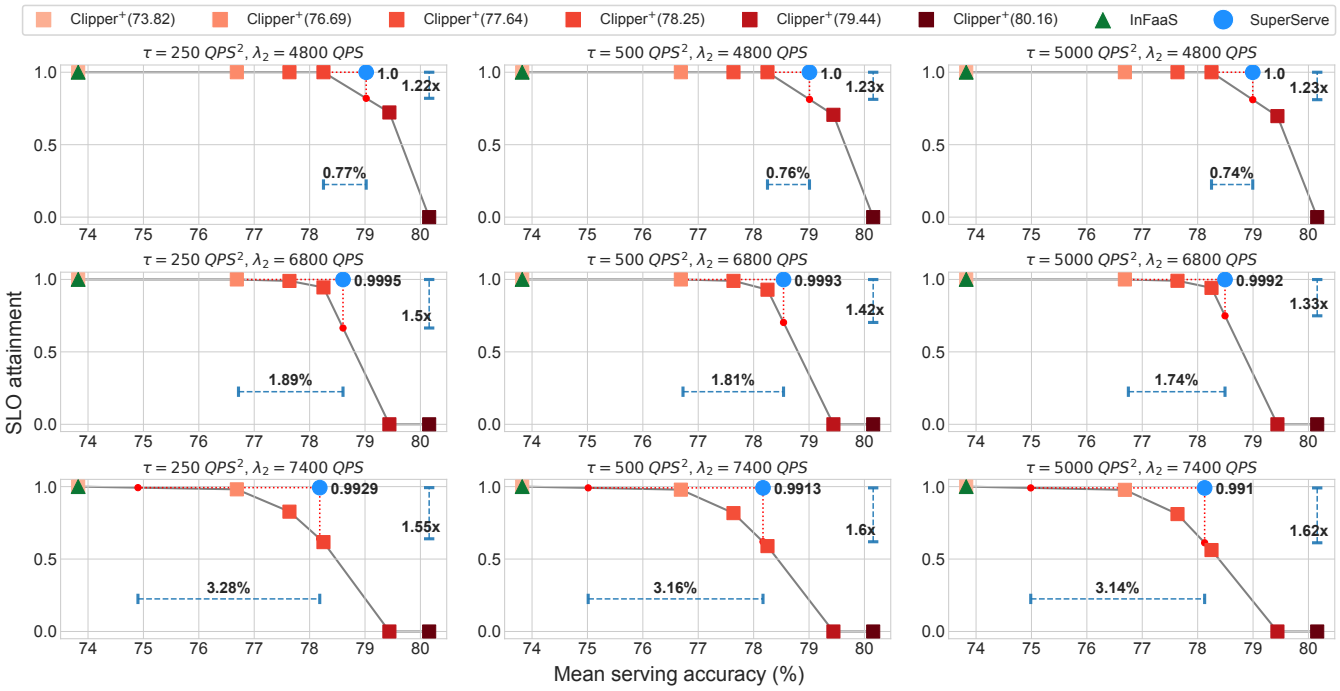
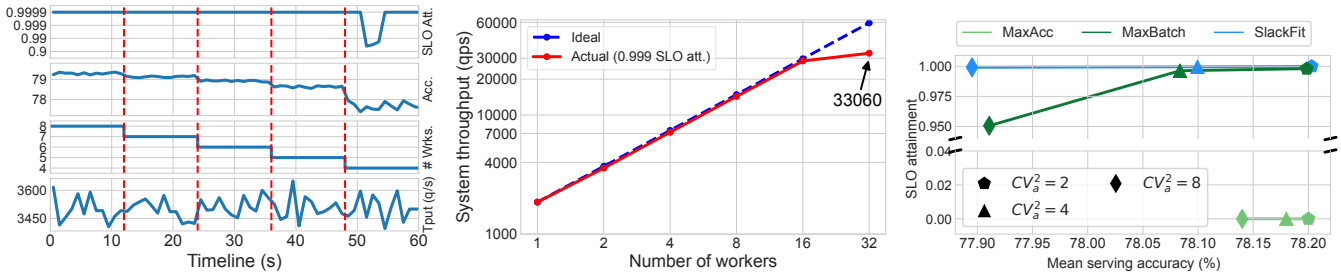


Fig. 10: SuperServe with arrival acceleration. SuperServe outperforms Clipper$^+$ and INFaaS baselines by finding better tradeoffs on time varying traces. Mean ingest rate accelerates from λ_1 to λ_2 q/s with τ q/s^2. $\tau = \{250, 500, 5000\}$ increases horizontally (across), while $\lambda_2 = \{4800, 6800, 7800\}$ increases vertically (down) with $\lambda_1 = 2500$ q/s and $CV_a^2 = 8$ staying constant. SuperServe finds a better trade-off in SLO attainment (y-axis) and mean serving accuracy (x-axis).



(a) Fault-Tolerance **(b)** Scalability **(c)** Policy-SE

Fig. 11: SuperServe's Micro-benchmarks. (a) SuperServe resiliency to faults. The system maintains high SLO attainment by dynamically adjusting served accuracy as workers drop out over time. The trace stays statistically the same ($\lambda = 3500$ qps, $CV_a^2 = 2$ (last row)). **(b)** SuperServe scales linearly with the number of workers, achieving up to 33000 qps while maintaining high .999 SLO attainment. **(c)** SlackFit finds the best tradeoff on the SLO attainment/ accuracy maximization continuum automatically (§6.4).

6.4 Microbenchmarks

Fault Tolerance. SubNetAct mechanism provides an additional advantage of transparent fault tolerance. We run SuperServe with 100% capacity (8 workers) with a bursty traffic trace ($\lambda = 3500$ qps, $CV_a^2 = 2$) for 60 seconds and gradually kill a worker every 12 seconds to simulate faults. SuperServe shows resilience to decreases in system throughput to as low as 50% by maintaining SLO attainment as high as 0.999 for the unchanging trace as it leverages subnetwork activation to serve lower accuracy models automatically. Similar methodology was used in [54]. Fig. 11a shows SLO attainment as a function of time (along with other system dynamics). As the faults occur (workers killed, dotted red lines), SuperServe *automatically* transitions to lower accuracy models to maintain high SLO attainment. We attribute SuperServe's fault tolerance to (a) a wide-dynamic throughput range afforded by SubNetAct (Fig. 5c) that allows SuperServe to serve the workload even with 50% capacity, and (b) SubNetAct's low actuation delay that provides *agility* to rapidly increase system-throughput (during faults) without SLO violations (**R1**).

Scalability. We assess if SuperServe reaches high SLO attain-

ment at scale. To show this, we scale the number of workers and observe the maximum throughput SuperServe sustains to reach SLO attainment of 0.999. We serve ResNet-18 [27] across all the workers with clients providing a batch of 8 images[4]. Scalability experiments are conducted with $CV_a^2 = 0$. Fig. 11b shows sustained ingest throughput with the increase in workers. SuperServe achieves an SLO attainment of 0.999 while reaching throughputs as high as $\approx$ 33000 qps.

Policy Space Exploration. We compare different policies implemented in SuperServe (Fig. 11c). We show that SlackFit achieves the best tradeoff w.r.t. our success metrics compared to both MaxAcc (greedily maximizes accuracy) and MaxBatch (greedily maximizes throughput) as CV_a^2 is varied. Details of the policies and the experiment are in §A.5.

7 Related Work

Training SuperNets was first proposed by OFA [8]. Recent works such as CompOFA [47] and BigNAS [58] propose improvements to the SuperNet training. CompOFA makes the training of SuperNets faster and more accurate by training a fewer number of SubNets simultaneously. On the other hand, BigNAS trains the SuperNet in one-shot with a wider range of SubNets. DynaBERT [28] trains a SuperNet based on the Transformer architecture for text datasets. Similarly, AutoFormer [12] trains SuperNets derived from vision transformers. NasViT [22] trains the SuperNet for semantic segmentation tasks and achieves a better trade-off between accuracy and latency. SuperServe provides system support for serving Supernets trained using any existing technique. We believe that SuperNets are an emerging phenomenon, and system support for serving them is the need of the hour.

Model serving systems can be divided into two categorizes — a) Non-Automated, and b) Automated. *Non-automated serving system* expect developers to provide the prediction models and make explicit choices in the accuracy-latency trade-off space. TensorFlow Serving [40] serves the models trained in TensorFlow framework while Clipper [14] and Triton [39] support models trained from multiple frameworks. Clockwork [25] guarantees predictable tail latency for DNN inference by making cross-stack design decisions explicitly for worst case predictability. Inferline [13] provides support for provisioning inference pipelines that consist of multiple models, but the models are still hard coded in the pipeline vertices. Prior works in this category are complementary to SuperServe. For instance, SuperServe's workers can be made more predictable by consolidating choices like Clockwork. Inferline's autoscaling policy can be used on top of SuperServe. Triton's model optimizations can be done to the SuperNet itself.

In contrast, *automated serving systems* [3, 4, 45, 61] automate the navigation of the accuracy-latency trade-off space with a policy, resulting in automatic DNN selection at runtime. However, both [45] and [61] use state-of-the-art DNNs (e.g., ResNets, MobileNets) and rely on model loading mechanisms instead of SuperNets, which offers better pareto-optimality and orders of magnitude faster model switching enabled via proposed SubNetAct. More importantly, these mechanisms implicitly bias their policies to avoid model switching, which limits their ability to respond to bursty request rates in an agile fashion. Specifically, InFaaS's DNN switching policy is biased towards selecting the least accurate DNNs that satisfy accuracy constraints, as the goal of the stated goal of the system is to satisfy constraints instead of treating accuracy as an optimization objective. Proteus [3] formulates the accuracy scaling problem as an MILP. However, the accuracy scaling action remains coarse-grained as MILP solver is invoked every 30 seconds in Proteus. Such coarse-grained decision-making has limitations like InFaaS: SLO violations with high actuation delays. Moreover, Proteus's batching technique is complementary to SlackFit. It introduces a delay in scheduling to account for heterogeneous resources. Such delays can be introduced in SlackFit to extend it to heterogeneous resources. Loki [4] tackles accuracy and hardware selection for ML inference pipelines. Extending fine-grained decision-making to ML inference pipelines is future work. SuperServe supports model serving via SubNet activation, thus addressing the model switching overhead and enabling fine-grained decision-making for bursty workloads.

8 Conclusion

We describe a novel mechanism SubNetAct that carefully inserts specialized control flow operators into SuperNets to enable a resource-efficient, fine-grained navigation of the latency-accuracy tradeoff space. SubNetAct unlocks the design space of reactive scheduling policies. We design a simple, yet effective greedy heuristic-based scheduling policy SlackFit. SuperServe, which uses SubNetAct and SlackFit, achieves 4.67% better accuracy at the same level of SLO attainment or 2.85x better SLO attainment at the same level of accuracy compared to state-of-the-art inference serving systems.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation (NSF) under Grant Number CNS-2420977 as well as a sponsored research award from Cisco Research. We would also like to sincerely thank the NSDI reviewers and, especially, our shepherd, Seo Jin Park, for their insightful comments that significantly improved the quality of this paper.

Disclaimer: Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and don't necessarily reflect the views of the NSF.

[4] We don't perform adaptive batching for this experiment

References

[1] Accelerate ai development with google cloud tpus. https://cloud.google.com/tpu.

[2] Aws inferentia. https://aws.amazon.com/machine-learning/inferentia/.

[3] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 318–334, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Sohaib Ahmad, Hui Guan, and Ramesh K. Sitaraman. Loki: A system for serving ml inference pipelines with hardware and accuracy scaling. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, volume 1 of *HPDC '24*, page 267–280. ACM, June 2024.

[5] Waleed Ali, Sherif Abdelkarim, Mahmoud Zidan, Mohamed Zahran, and Ahmad El Sallab. Yolo3d: End-to-end real-time 3d oriented object bounding box detection from lidar point cloud. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 0–0, 2018.

[6] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *computer*, 50(10):58–67, 2017.

[7] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[8] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020.

[9] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Ofa pretained models. https://drive.google.com/drive/folders/10leLmIiMtaRu4J46KwrBaMydvQt0qFuI?usp=sharing, 2022.

[10] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[11] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[12] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 12270–12280, 2021.

[13] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: ML inference pipeline composition framework. *CoRR*, abs/1812.01776, 2018.

[14] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[16] Facebook. Torchscript. https://pytorch.org/docs/stable/jit.html, 2022.

[17] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *International Conference on Learning Representations*, 2020.

[18] Romero Francisco, Qian Li, and Christos Kozyrakis. Personal Communication, December 2022.

[19] Shen Gao, Xiuying Chen, Piji Li, Zhaochun Ren, Lidong Bing, Dongyan Zhao, and Rui Yan. Abstractive text summarization by incorporating reader comments. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6399–6406, 2019.

[20] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E Gonzalez, and Ion Stoica. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 453–471, 2022.

[21] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A Wright, Joseph E Gonzalez, and Ion Stoica. Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8806–8813. IEEE, 2021.

[22] Chengyue Gong, Dilin Wang, Meng Li, Xinlei Chen, Zhicheng Yan, Yuandong Tian, Vikas Chandra, et al. Nasvit: Neural architecture search for efficient vision transformers with gradient conflict aware supernet training. In *International Conference on Learning Representations*, 2021.

[23] Google. gRPC. https://grpc.io/docs/languages/cpp/quickstart/, 2022.

[24] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. Swayam: Distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 109–120, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[26] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[28] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9782–9793. Curran Associates, Inc., 2020.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[30] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. *CoRR*, abs/1710.08460, 2017.

[31] Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. SMART: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2177–2190, Online, July 2020. Association for Computational Linguistics.

[32] Ameet V Joshi. Amazon's machine learning toolkit: Sagemaker. In *Machine Learning and Artificial Intelligence*, pages 233–243. Springer, 2020.

[33] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, 2016.

[34] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[35] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[36] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[37] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[38] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11976–11986, 2022.

[39] Nvidia. Triton inference system. https://github.com/triton-inference-server/server/tree/v2.21.0, 2020.

[40] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[41] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using

specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

[42] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for memory-efficient, real-time video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, 2023.

[43] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[44] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[45] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[46] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. InFaaS policy's decisions with respect to accuracy constraints. https://github.com/stanford-mast/INFaaS/blob/02ad2b86bcbc66d55b17baaef608c4e864b03918/src/master/queryfe_server.cc#L342-L356, 2022.

[47] Manas Sahni, Shreya Varshini, Alind Khare, and Alexey Tumanov. CompOFA – Compound Once-For-All Networks for Faster Multi-Platform Deployment. In *International Conference on Learning Representations*, 2021.

[48] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[49] Jonathan Soifer, Jason Li, Mingqin Li, Jeffrey Zhu, Yingnan Li, Yuxiong He, Elton Zheng, Adi Oltean, Maya Mosyak, Chris Barnes, et al. Deep learning inference service at microsoft. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, pages 15–17, 2019.

[50] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[51] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[53] Prahlad Venkatapuram, Zhao Wang, and Chandra Mallipedi. Custom silicon at facebook: A datacenter infrastructure perspective on video transcoding and machine learning. In *2020 IEEE International Electron Devices Meeting (IEDM)*, pages 9–7. IEEE, 2020.

[54] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 338–352, 2019.

[55] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[56] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at Facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[57] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 768–781, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang,

Xiaodan Song, Ruoming Pang, and Quoc Le. BigNAS: Scaling up neural architecture search with big single-stage models. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 702–717, Cham, 2020. Springer International Publishing.

[59] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[60] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, 2023.

[61] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[62] Minjia Zhang, Samyam Rajbandari, Wenhan Wang, Elton Zheng, Olatunji Ruwase, Jeff Rasley, Jason Li, Junhua Wang, and Yuxiong He. Accelerating large scale deep learning inference through DeepCPU at microsoft. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, pages 5–7, 2019.

[63] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[64] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

A Appendix

A.1 SubNetAct: Automatic Operator Insertion

We introduce SubNetAct's control flow operators automatically. Specifically, the LayerSelect operator is introduced at every stage of $\mathcal{M}$, and each block within the stage (such as Bottleneck in OFAResNets [8] or TransformerBlock in Dynabert [28]) is converted to a boolean module whose boolean handle is tracked by the LayerSelect operator. Each convolution or attention layer of $\mathcal{M}$ is modified by wrapping it with the WeightSlice operator. Finally, all the batchnorm layers in $\mathcal{M}$ are converted to the SubnetNorm operator with additional information provided to it about each subnet's tracked statistics. The algorithm to automatically enable control flow operations in M is provided in Alg. 1.

Input: Supernet Arch. $\mathcal{M}$, Supernet Weights W, Tracked Mean and Variances $TrackedStats$

```
newOperators = {}
for s ∈ STAGES(M) do
    // layerSelect operator selects layers within each stage
    ls = LAYERSELECT()
    newOperators[s] = {}
    for m ∈ GETMODULES(M, s) do
        if m.type == Bottleneck || m.type ==
          TransformerLayer then
            bool select_m // boolean switch for layer
            m_new = TOBOOLMODULE(m, select_m)
            // layerSelect controls boolean of stage's layers
            ls.REGISTERBOOL(select_m)
        end
        else if m.type == Attention || m.type == Conv then
            // WeightSlice applied to attn or conv layers
            m_new = WEIGHTSLICE(m.type, W[m.id])
            newOperators[s][m.id] = m_new
        end
        else if m.type == BatchNorm then
            // SubnetNorm only applied to BatchNorm
            m_new=SUBNETNORM(W[m.id], TrackedStats)
            newOperators[s][m.id] = m_new
        end
        MODIFYMODULE(M, m, m_new)
        newOperators[s]["layerSelect"] = ls
    end
end
REGISTERCONTROLFLOWOPS(M, newOperators)
```

Algorithm 1: Introducing SubNetAct Operators in Supernets. The algorithm introduces control-flow operates to enable SubNetAct for latency/accuracy navigation. The pre-requisites to enable SubNetAct are trained weights and architecture of the supernet that are obtained from existing NAS approaches [8, 28, 47, 58].

A.2 SubNet Profiled GFLOPs

As discussed in §4.2, SlackFit bucketizes SubNet $\phi \in \Phi_{\text{pareto}}$ and batch size $|B|$ using three properties. While Fig. 6 showed that inference latency increases monotonically with both batch size (**P1**) and accuracy (**P2**), the analytical basis for it is shown in Fig. 12. This heatmap of GFLOPs (Giga Floating Point Operations) quantifies the computational demand for each pair of neural network model architecture (SubNet ϕ) and batch size ($|B|$). Fig. 12a and Fig. 12b demonstrate that the computational demand for any SubNet ϕ increases as its batch size increases. Moreover, SubNets with higher accuracy require more FLOPs in both Transformer-based and Convolution-based SuperNets. Thus, given a specific hardware (in our case NVIDIA RTX2080Ti GPU), the inference latency increases as computational demand from (SubNet ϕ, $|B|$) increases, validating properties **P1** and **P2**.

Finally, the FLOPs distribution also shows multiple instances where the FLOP count for lower accuracy SubNet with higher batch size is similar to or less than the FLOP for a higher accuracy SubNet with lower batch size. For e.g., in Fig. 12b, (73.82, BS16) requires FLOPs slightly less than (80.16, BS2). This reduced GFLOP demand influences the inference latency, and allows lower accuracy SubNets to serve higher batch sizes at similar latencies as higher accuracy SubNets with lower batch sizes, validating **P3**.

A.3 System Dynamics: Synthetic Traces

We also derive key observations from the dynamics to understand how SuperServe achieves high SLO attainment and better trade-offs (**R1-R2**) for synthetic traces.

Fig. 13 shows the system dynamics of SuperServe for both bursty and time-varying traces. The mean ingest rate of the bursty traces is 7000 qps and they vary in $CV_a^2 = \{2, 8\}$. Similarly, in case of the time-varying traces, the ingest rate is increased from λ_1 qps to λ_2 qps at varying accelerations $\tau = \{250\ q/s^2, 5000\ q/s^2\}$. The control decisions made by SlackFit (subnetwork (accuracy) and batch size) are shown over time.

Fig. 13a shows system dynamics for the bursty traces. The trace with $CV^2 = 8$ (blue line) has higher spikes than the trace with $CV^2 = 2$ (orange line). First, note that SuperServe operates at an accuracy range of $76 - 78\%$ and never selects a higher accuracy subnetwork such as the subnetwork of 80.16% accuracy. This is because the subnetwork of 80.16% accuracy diverges at the mean ingest rate of 7000 qps (also seen in Fig. 9 last row). Hence, SuperServe *automatically* selects appropriate subnetworks for different mean ingest rates. Moreover, SuperServe uses lower accuracy models more frequently with increasing CV_a^2. This is because increased jitter reduces query slack, causing SlackFit to pick lower latency buckets more often. This corroborates the trend seen in Fig. 9 where the mean serving accuracy of SuperServe monotonically decreases as CV_a^2 increases. Lastly, during the load spikes, SlackFit usually selects control parameters with high batch size and smaller subnetwork (§4.2.1). This control decision allows SuperServe to drain the queue faster, resulting in a high SLO attainment on the traces (**R1**).

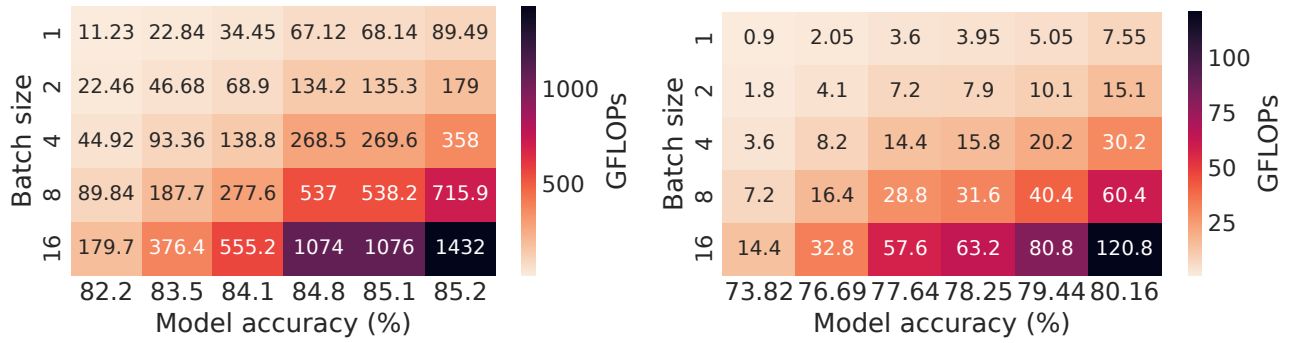
Fig. 13b shows the system dynamics for the time-varying traces. $\tau = 5000\ q/s^2$ (blue line) increases the ingest rate from 2500 qps to 7400 qps faster than $\tau = 250\ q/s^2$. For both the traces, SuperServe dynamically changes the accuracy from ≈ 79.2 to ≈ 77.5 as mean ingest rate increases. SuperServe's ability to dynamically adjust accuracy helps it achieve a higher mean serving accuracy (**R2**) compared to serving a single model statistically. Moreover, for $\tau = 5000\ q/s^2$, SuperServe jumps to lower accuracy and higher batch size control parameters quickly. While, for for $\tau = 250\ q/s^2$, SuperServe uses intermediate models to serve the intermediate ingest rate during $\approx 60 - 80$ seconds. A higher τ value forces query's slack to reduce drastically. Hence, SlackFit rapidly switches to selecting control parameters of smaller subnetwork and higher batch size from the low latency buckets (§4.2) to satisfy deadlines (**R1**). Therefore, increase in τ decreases mean serving accuracy (a trend observed in Fig. 10 across the rows).

A.4 Scheduling policies

The core functionality of the SuperServe's scheduler is to maximize (a) SLO attainment and (b) prediction accuracy for any arrival trace dynamics. The scheduler offers a pluggable policy framework to support any application sensitivity to these metrics by allowing arbitrarily different trade-offs between them. SuperServe policy interface dictates that control decisions are made w.r.t. the batch size and subnetwork to activate. Both of these control parameters affect SLO attainment and serving accuracy. This is because the scheduler a) perpetually operates under a latency constraint and b) these control decisions have a cumulative effect over time (e.g., higher accuracy affects queue build-up later). Finding a globally optimal set of batch size and subnetwork control tuples over time is NP-hard. As our control decisions must be made on the critical path of queries' end-to-end latency, quick sub-millisecond control decision making is a key performance requirement. Thus, to meet the real time requirements, we primarily consider scheduling policies that are greedy w.r.t. time. The policies decide the batch size and subnetwork based on the remaining slack of the most urgent query. The slack is calculated using a fast (sub-ms) $O(1)$ EDF queue lookup operation.

A.5 Policy Design Space

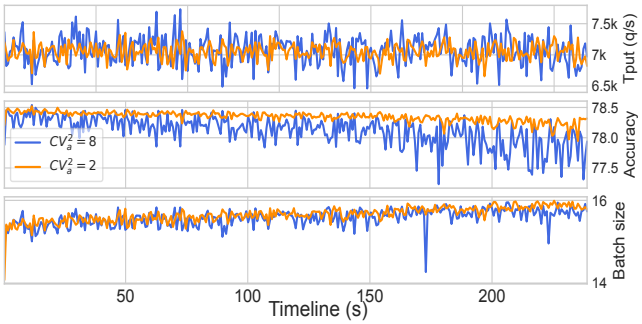
MaxBatch Policy. This policy first maximizes the batch size and then the accuracy. It greedily finds a maximal batch size (b) for the smallest accuracy subnetwork that fits within latency slack θ.Within the chosen batch size MaxBatch finds



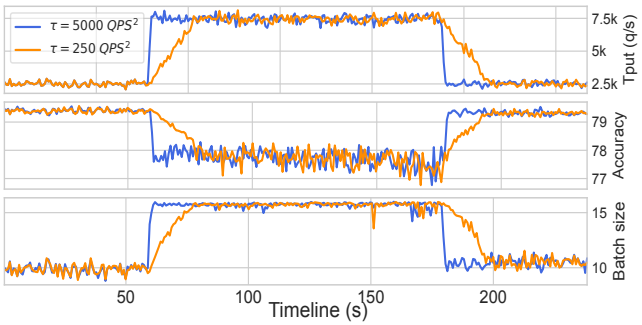
(a) Transformer-Based SuperNet

(b) Convolution-Based SuperNet

Fig. 12: SlackFit control parameter space. FLOPs for six different pareto-optimal SubNets in SubNetAct as a function of accuracy (x-axis) and batch size (y-axis) shown for both transformer and convolution-based supernet. The FLOPs are *monotonic* with batch size and accuracy. This trend in FLOPs forms the analytical basis of the trend in the inference latency of these models (as shown in Fig. 6).



(a) Dynamic accuracy and batch size control: bursty traces



(b) Dynamic accuracy and batch size control: time-varying

Fig. 13: System Dynamics on Synthetic Traces. Accuracy and batch size control decisions shown over time in response to ingest throughput (q/s). (a) bursty traces $\lambda = 7000 = (\lambda_b = 1500) + (\lambda_v = 5500)$ with burstiness of $CV_a^2 = 2$ (orange) and $CV_a^2 = 8$ (blue). (b) time varying traces accelerate from $\lambda_1 = 2500$ q/s to $\lambda_2 = 7400$ q/s with acceleration $\tau = 250q/s^2$ (orange) and $\tau = 5000q/s^2$ (blue). Batch size and subnetwork activation control choices over time show how SuperServe reacts to each of the four plotted traces in real time. This illustrates dynamic latency/accuracy space navigation.

the maximum accuracy subnetwork (s) such that the profiled latency $L(b,s) < \theta$. It returns the control choice (b,s). This policy leverages insights (**I1**) and (**I2**). It takes O($log(B)$) operations to find b and O($log(S)$) operations to find s (binary search on monotonically increasing latency w.r.t. batch size and accuracy). As a result, this lightweight policy scales well with the profile table, taking only O($log(B) + log(S)$) operations to make control decisions.

MaxAcc Policy. MaxAcc first maximizes the accuracy and then the batch size. Mirroring MaxBatch, MaxAcc performs a binary search for the largest accuracy (s') with $L(1,s') < \theta$ first. Then, it finds the maximal batch size (b') keeping the subnetwork choice fixed to the chosen s', such that $L(b',s') < \theta$ ms. Similarly to MaxBatch policy, it leverages insights (**I1**) and (**I2**) and takes O($log(B) + log(S)$) operations to return the control choice (b',s').

The proposed SlackFit Policy. This is our best performing policy. At a high level, SlackFit partitions the set of feasible profiled latencies into evenly sized latency buckets. Each bucket consists of control tuples (b,s) with $L(b,s)$ within the range of bucket width. Then the policy chooses a bucket with latency $\leq \theta$. Finally, from the choices within the selected bucket, it picks the control choice that maximizes batch size. Intuitively, selecting control parameters closest to slack θ configures the system to operate as close to capacity as possible.In other words, choices with latency less than that either reduce the throughput capacity or the serving accuracy, eventually lowering system's SLO attainment and accuracy. This draws on the monotonicity insights (**I1**) and (**I2**). SlackFit's novelty is in insight (**I3**). We observe that SlackFit *dynamically* detects and adapts to the runtime difficulty of the trace. A well-behaved trace (e.g., low ingest rate, variation, acceleration) results in higher θ. Higher θ leads to the choice of higher latency buckets. And higher latency buckets are correlated strongly with fewer control tuple choices (Fig. 6), maximizing the probability of choosing higher accuracy models. Conversely, mal-behaved traces (higher ingest rate, variation, acceleration) lead to lower latency bucket choices, as the scheduler is operating under much lower θ conditions. There

are more control choices in lower latency buckets, which leads to control tuples within those buckets to favor higher batch sizes. This leads to processing the queue faster.

Experiment Result. In Fig. 11c we show that SlackFit achieves the best tradeoff w.r.t. our success metrics compared to both MaxAcc – a policy that greedily maximizes accuracy and MaxBatch — a policy that greedily maximizes batches. The traces used mean $\lambda = 7000$ qps $((\lambda_b = 1500) + (\lambda_v = 5550))$ and $CV_a^2 \in \{2, 4, 8\}$. SlackFit reaches the highest SLO attainment(0.999) for all CV_a^2. MaxBatch starts under performing w.r.t. SLO attainment with CV_a^2 increase. The SlackFit and MaxBatch difference is most pronounced at the highest CV_a^2, eventually causing a significant 5% drop in the SLO attainment. Both policies maximize the batch size within latency slack θ when operating under small θ. When θ increases, however, MaxBatch continues to maximize the batch size unconditionally—a greedy choice that leads to packing larger batches. This greedy decision causes more time to be spent in a worker compared to SlackFit, which adaptively shifts to higher accuracy models under larger θ conditions with compound effect on queued queries, eventually missing their SLOs. maxAcc is unable to keep up with this trace. It never switches to policy decisions that process the queue faster. This policy comparison shows a continuum between faster queue processing and serving higher accuracy, with SlackFit automatically finding the best point in this continuum.

Pineapple: Unifying Multi-Paxos and Atomic Shared Registers*

Tigran Bantikyan
Northwestern

Jonathan Zarnstorff
Unaffiliated

Te-Yen Chou
CMU

Lewis Tseng
UMass Lowell

Roberto Palmieri
Lehigh University

Abstract

Linearizable storage systems reduce the complexity of developing correct large-scale customer-facing applications, in the presence of concurrent operations and failures. A common approach for providing linearizability is to use consensus to order operations invoked by applications. This paper explores designs that offload operations (from the consensus component) to improve overall performance.

This paper presents Pineapple, which uses logical timestamps to unify Multi-Paxos and atomic shared registers so that any node in the system can serve read and write operations. Compared to Multi-Paxos (or leader-based consensus), Pineapple reduces bottlenecks at the leader. Compared to Gryff, which unifies EPaxos and atomic shared registers, Pineapple has better performance because Pineapple has "*non-blocking* operation execution."

Our evaluation shows that Pineapple improves both throughput and tail latency, compared to state-of-the-art systems (e.g., Gryff, Multi-Paxos, EPaxos), in both wide-area networks and local-area networks. We also integrate Pineapple with etcd. In a balanced workload, Pineapple reduces median latency by more than 50%, compared to the original system that uses an optimized version of Raft.

1 Introduction

Large-scale customer-facing applications increasingly rely on linearizable storage systems. These systems ensure that a service is highly available and consistent in the presence of concurrent operations and node crashes. Modern applications prefer strong consistency over weaker consistency, because it simplifies application logic, which then significantly lowers the effort in development and maintenance [11, 15, 17, 19].

One common solution for developing linearizable storages is using consensus. Multi-Paxos [36, 39] and Raft [53], two popular leader-based consensus algorithms, have been adopted in many production storage systems, e.g., Google Spanner [17], Microsoft Azure Storage [11], Redis [2], RethinkDB [4], CockroachDB [3], and etcd [1]. To improve performance, this paper explores approaches that offload reads and writes from the leader to followers (or non-leaders).

Targeted Workloads. We focus on linearizable storages designed for large-scale web applications. For scalability and availability, these applications often rely on variations of key-value storages, NoSQLs, or NewSQLs, whose performance could be improved using our design.[1] Most modern storages support reads and writes on a single key-value pair, e.g., Cassandra [35], etcd [1], Redis [2], and Memcached [7, 52].

Example large-scale web applications that heavily use such single-key reads or writes include Facebook inbox search [35], WeChat's user account management [73], and Microsoft's OneDrive [15]. In these applications, the program logic and data models allow the developers to use single-key operations efficiently. For example, WeChat creates table(s) as per user account [73], and Microsoft uses a versioned object storage as the backend for OneDrive [15].

In fact, the need to optimize read and write performance can be found in many real-world systems. For example, WeChat designs PaxosStore [73] that uses a combination of quorum and leader lease to reduce latency for reads and writes. Google's Spanner [17] and Microsoft's Giza [15] use variations of techniques to achieve similar goals (e.g., Spanner's TrueTime and Giza's adoption of Fast Paxos [38]).

Performance studies from Facebook [7, 9] also support the goal of improving read and write performance: (i) Only 0.2% of operations involve a write in their distributed data store for social graph TAO; and (ii) 97% of operations are reads, writes, or deletes in Facebook's usage of RocksDB, a key-value store.

Pineapple. Motivated by these observations, we propose Pineapple, a high-performance *linearizable key-value storage*

*The work was partially done when the first three authors were with Clark University. First two authors have equal contribution. This material is based upon work supported by the National Science Foundation under Grant No. CNS-2045976 and CNS-2449640.

[1] In typical NoSQL databases, an entire row can be treated as a single key-value pair. Hence, our design could be compatible with data models in similar systems. For conciseness, the paper is focused on key-value storages.

that offloads reads and writes from the leader of Multi-Paxos to followers. As in [37,43,46,61], we assume that an operation touches a single key or a set of keys, where each key uniquely identifies an object (or a piece of data). With a slight abuse of terminology, we use "key" and "object" interchangeably. Pineapple supports three operations (definitions in §2.1):

(i) reading the value from a single key,
(ii) writing a value to a single key, and
(iii) performing a one-shot transaction [34], which either accesses or updates the values of a set of keys, or provides a stronger synchronization (e.g., read-modify-write, fetch-and-add, compare-and-swap), or both.

Pineapple: Blind Writes and One-shot Transactions. The single-key write in Pineapple is in fact a "blind write," which overwrites an existing key-value pair. While not as powerful as conditional writes, blind writes are still applicable to many web applications. For instance, many storage systems, such as Cassandra, Spanner, HBase, RocksDB, Redis, and etcd, support blind writes. Moreover, useful features for web applications can be implemented using blind writes. Examples include versioned objects for Microsoft Giza [15] and Amazon S3, and data structures with associative and commutative properties (e.g., data structures with statistical or probabilistic properties like Histogram and Hyperloglog) [12].

Pineapple supports one-shot transactions, which can be represented as *stored procedures* [34,48,49]. In these transactions, clients (or applications) send the transaction logic to servers, and the servers execute them *atomically*, without client interaction during the process. Additionally, the transactions' input and output key sets are known a priori. Therefore, servers can execute the transactions (once they are ordered) without further interaction with the clients.

While less powerful than general transactions, one-shot transactions are sufficient for many web applications [10,49]. For example, one-shot transactions can automatically update multiple users' accounts (e.g., money transfers) on payment platforms. They can also enable a scan or range query for files on cloud object-based storage. A key-value store, which provides only single-key reads/writes, and Gryff [10], which provides only single-key reads/writes/read-modify-writes, are not sufficient for supporting these operations. An application either has to sacrifice guarantees (e.g., relaxed consistency when doing a range query) or use general transactional storage (potentially with lower performance on single-key operations).

Pineapple: Key Techniques. Pineapple uses a quorum-based leaderless approach (ABD [8]) to serve reads and writes on a single key-value pair and relies on a leader-based consensus (Multi-Paxos) to agree on the order of one-shot transactions. Even though it is *not* as intuitive to combine a leaderless design with a leader-based one, such unification not only simplifies the overall design, but also improves performance.

Interestingly, we do not need a completely new design to unify the two components. In particular, we use a mature technique – logical timestamps – to unify Multi-Paxos and ABD. The main technical contribution is our insight to combine them together so that they can work seamlessly together, even in the period of leader change or leader failure.

Pineapple: Highlights. Pineapple provides the following features that would favor widespread adoption.

- Pineapple offloads single-key reads and writes to follower nodes. This helps reduce tail latency for reads and writes and increase overall throughput.
- In a 3-node system, Pineapple guarantees that single-key reads complete in 1 round-trip time (RTT). With more nodes, reads can complete in 1 RTT if there is no concurrent one-shot transaction or write on the same key.
- Pineapple supports one-shot transactions that can access or update the values of a set of keys and provide stronger synchronization power.
- All operations supported by Pineapple have "*non-blocking* execution," i.e., a node can execute any operation, after communicating with the leader or a quorum.

These features jointly allow Pineapple to improve performance over state-of-the-art systems, e.g., Gryff [10], Multi-Paxos, PQR [13], EPaxos [46], in both wide-area networks and local-area networks over a wide spectrum of workloads.

Non-blocking operation execution is a natural feature in consensus algorithms like Multi-Paxos and Raft. To improve performance, recent systems often choose to separate ordering and execution phases, causing the executions of some operations *blocking*. For example, EPaxos and Gryff need to resolve dependency before execution. PQR needs to wait for committed log entries before executing reads.

As we will see in §4, such blocking operation executions cause longer tail latency and reduced throughput. Moreover, the blocking nature is not ideal for practical deployment, because a slow node (e.g., due to intermittent connection issues, misconfigurations or garbage collection events [23,24,50]) could impact latency and throughput unexpectedly.

Pineapple also has one appealing feature for practitioners. Pineapple is built upon mature techniques that are well understood and tested, e.g., Mutli-Paxos, ABD (or quorum-based approach), and logical timestamps. Fundamentally, Pineapple adds one more round of communication into Multi-Paxos so that it could interact with ABD atomic registers. Therefore, we believe that our design of Pineapple could potentially be more straightforward to understand and integrate with production systems that use leader-based consensus.

Existing Solutions. There are roughly four categories of solutions on improving the performance of consensus-based linearizable storages: (i) a completely new consensus algorithm that relies on different assumptions and models, e.g., [6,23,24,37,44,46,54,55], (ii) a variant of existing consensus algorithm that better utilizes new hardware such as RDMA or SDN, e.g., [5,40,45], (iii) an engineering or software solution

	Latency			(Fast) Quorum Size	Non-blocking Exe.	Limitations
	Read	Write	Other			
Multi-Paxos [36]	1/2			$\lfloor n/2 \rfloor + 1$	all ops	leader bottleneck
PQR [13]	1/1	1/2	1/2	$\lfloor n/2 \rfloor + 1$	non-read ops	leader bottleneck & read tail latency
EPaxos [46]	1/2			$f + \lfloor (f+1)/2 \rfloor$	-	long tail latency
Gryff [10]	1/2	2/2	2/3	$f + \lfloor (f+1)/2 \rfloor$	reads and writes	lower throughput & no support of multi-key ops
Pineapple (this work)	1/2	2/2	2/3	$\lfloor n/2 \rfloor + 1$	all ops	3-RTT one-shot transactions

Table 1: **Pineapple vs. closely related systems.** • All of the systems tolerate up to f crashed nodes with $n = 2f+1$ nodes.
• The first number in Latency columns indicates the RTTs in the case of no conflict (for EPaxos, Gryff, and Pineapple's read) or the RTT in the case when the operation is served from a leader node (for Multi-Paxos, PQR and Pineapple's write and one-shot transactions, and the second is the RTT with conflict (for EPaxos, Gryff, and Pineapple's read) or the RTT in the case when the request is served from a follower node (for Multi-Paxos, PQR's writes and Pineapple's one-shot transactions. "Other" represents RMW for Gryff and one-shot transactions for other systems.
• The second column specifies the quorum size for Multi-Paxos, PQR, and Pineapple and the fast quorum size for EPaxos and Gryff.
• An operation has "*non-blocking* execution" if it can be executed immediately upon learning a decision of its ordering.

to scale an existing consensus algorithm, e.g., [14, 61, 64, 69], and (iv) a new algorithmic component that is integrated with an existing consensus algorithm, e.g., [10, 13, 31, 37, 38].

Table 1 compares key features between Pineapple and closely related systems. (More discussion on other related work in §5.) Pineapple belongs to category (iv). Two notable recent systems in this category are PQR [13] and Gryff [10]. PQR improves Multi-Paxos's performance by offloading reads to followers. Gryff allows any node to perform reads, writes and Read-Modify-Writes (RMWs) by unifying EPaxos [46] and atomic shared registers [8, 30]. Both demonstrate improved latency or throughput over prior solutions such as Multi-Paxos and EPaxos under workloads for web applications. Compared to Pineapple, there are a few limitations in PQR and Gryff:

- In PQR, only reads are offloaded to followers; hence, other operations can still saturate the leader.
- In PQR, reads have "*blocking* execution," i.e., reads may not be executed immediately and have to wait for a committed entry from the leader. The effect of blocking executions, which leads to longer latency, can be observed even during a light workload without leader change.
- Gryff's throughput is generally lower than *both* EPaxos and Multi-Paxos due to its more complicated execution logic and extra message overhead.
- Gryff's operations can *only* access or update a *single* key. In other words, it doesn't support one-shot transactions. Many operations used by modern web applications are not supported by Gryff. (More details in §2.3.)
- Gryff's RMWs have "blocking execution," i.e., the execution of RMW needs to wait until all the prior dependency have been resolved and executed. This is because Gryff uses a variant of EPaxos to handle RMWs. Our evaluation shows that such blocking execution affects both tail latency and throughput.

Pineapple: Performance. We evaluate Pineapple in both local-area networks (LAN) and wide-area networks (WAN). Our evaluation shows that Pineapple improves performance in both cases. In both WAN and LAN, Pineapple has roughly 10% to 20% of improvement over the closest competitor (i.e., EPaxos and PQR) in throughput when RMW ratio is less than 20%. As for tail-latency in WAN, Pineapple generally has better read tail latency than Multi-Paxos and PQR with low-conflict workloads, in 3-node and 5-node deployments.

We also integrate Pineapple with etcd [1], a backend system used by many large-scale production systems such as Kubernetes. We replace its consensus layer (powered by Raft [53]) by Pineapple. Pineapple improves median latency by more than 50% in LAN, under a YCSB balanced workload [16]. This demonstrates that our design effectively improves latency in an optimized production system.

Outline. We first discuss the preliminaries in §2. We then present our design of Pineapple and its analysis in §3. We evaluate Pineapple in §4. Related work is presented in §5.

2 Preliminaries and Background

2.1 System Model and Shared Objects

This paper is focused on systems consisting of n nodes (or servers) in a point-to-point communication network. Up to f nodes may fail by crashing, e.g., stopping execution at any point of time. We assume $n = 2f+1$.

The system is assumed to be *asynchronous*. That is, there is no upper bound on the time a message takes to be delivered, and there is no upper bound on the relative speed at which nodes proceed. Due to the asynchrony assumption, a failure is not detectable by others. Pineapple ensures safety with asynchrony. Reads and writes can be completed under asynchrony, due to the adoption of ABD registers [8]. One-shot transactions satisfy liveness under *partial synchrony* [22] – when there is a stable leader that can communicate with a quorum, then one-shot transactions are guaranteed to be completed.

Shared Objects. The system consists of m shared objects (i.e., key-value pairs) that can be accessed and updated concurrently.[2] Each object is uniquely identified by a key (or an

[2]The assumption of m is only for easiness of presentation. In practical

identifier). Pineapple provides the following three types of operations. Any client connects to a Pineapple node can use the interface to access or update any of the m shared objects:

- Read(k): it returns the value of key k;
- Write(k, v): it updates the value of key k to value v; and
- OT($f(\cdot), K_{input}, K_{output}$): it takes three parameters – a function $f(\cdot)$ and two sets of keys – and it atomically performs the following three steps:
  (i) read the values of input key set, i.e., $V_{input} \leftarrow$ values of key set K_{input}
  (ii) compute the value of output key set, i.e., $V_{output} \leftarrow f(V_{input})$
  (iii) update the values of key set K_{output} to V_{output}.

By definition, $|V_{input}| = |K_{input}|$ and $|V_{output}| = |K_{output}|$, where $|S|$ denotes the cardinality of a set S. K_{input} and K_{output} may have different sizes and may not have any overlap.

This definition of one-shot transactions captures a wide range of "transactions" used in modern large-scale applications. For example, $y \leftarrow x + 1$ can be represented as OT($f(\cdot), \{x\}, \{y\}$), where $f(v) = v + 1$.

Another common usage is "conditional write." For example, the following function can be used for a one-shot transaction that implements an atomic counter that starts with a default value 0 and increments by 1 when invoked:

$$f(v) = \begin{cases} v+1, & \text{if } v \text{ is non-empty} \\ 0, & \text{otherwise} \end{cases}$$

Here we assume that a returned value is empty (or null), if an object is not initialized yet.

The function $f(\cdot)$ can take multiple inputs, which is useful to support richer semantics like compare-and-swap. (An example can be found in §2.3.) An alternative view is that one-shot transactions can perform the If/Then/Else logic over any subset of the m objects "*atomically*."

Linearizability. Pineapple achieves linearizability [30], which requires that concurrent operations appear to execute in some total order O such that (i) O is consistent with the interface (i.e., it is legal with respect to the semantics), and (ii) O is consistent with the real-time ordering constraint. That is, for any operation o_1 completes before the invocation of operation o_2, o_1 must appear before o_2 in the total order O.

2.2 Ordering Read/Writes w/o Consensus

Herlihy's wait-free hierarchy [29] defines different synchronization power of shared objects in asynchronous systems. For example, atomic shared registers that provide read and write operations on a single key have the weakest synchronization power, whereas consensus objects provide the strongest power. This hierarchy explains why there exists a non-blocking linearizable solution for ordering reads and writes for atomic registers, but not for consensus objects.

systems, objects can be dynamically created or deleted.

The difference between atomic register and other objects in the wait-free hierarchy is that each write completely defines the state of the single object *without* any reliance on a previous state. As a result, the total ordering does not have to be "stable." In other words, the ordering does *not* need to be observed by all nodes. Only the operations that are required by real-time ordering constraints need to be observed by all nodes.

An implication of such an observation is that a node i can safely apply a write, say o_{24}, which comes with the largest logical timestamp that i has seen so far. Node i can apply o_{24}, even if it does not know any prior writes. Suppose at some later time, i receives an earlier write, say o_8, it can safely ignore that write because it already has a more recent state (as defined by o_{24}). This is why the usage of logical timestamp is sufficient for atomic shared registers. Note that because of asynchrony, operations may be received in an arbitrary order.

Multi-Writer ABD. We briefly discuss the multi-writer variant [42] of ABD [8], which allows multiple concurrent writer clients to read or write a single key. It associates a tag $t = (ts, id)$ with each key where ts is a logical timestamp and id is the identifier of the writer proxy, i.e., the ID of the proxy node that handles the write for the client. Writes and reads can be ordered lexicographically using the associated tags, to obtain the desired total ordering for linearizability.

For each key, a node stores a value v and the tag t where t is the largest tag a node has observed so far. Both reads and writes have two phases – get and put phase. In the get phase, reads learn the value to be returned, whereas writes learn the largest tag from a quorum. In the put phase, both operations need to ensure that a quorum has obtained the information – this phase ensures real-time ordering.

A read begins with reading the tag and value from a quorum. Upon receiving responses, the reader proxy then retrieves the value v^{max} with the largest tag t^{max} from them. Then the proxy transmits (v^{max}, t^{max}) to a quorum and waits for acknowledgments. Once a quorum of acknowledgements is collected, the proxy returns v^{max} to the reader client.

A writer proxy p follows the same two-phase structure. The difference is that in the put phase, the proxy transmits the value to be written with a new tag $t = (t^{max}.ts + 1, p)$. Upon receiving a quorum of acknowledgement of the second phase, the proxy lets the writer client knows that the write has been completed. Since there is no guarantee that all the writes will be observed in the same order at each node, the ordering observed be each node at any point of time is "*non-stable*." That is, the order might change in the future.

Consensus and SMR. On the contrary, consensus or state-machine replication (SMR) [59] requires a "stable" ordering. More concretely, the nodes have to agree on the same sequence of operations, so the state machine at each node can apply these operations one by one starting from the beginning of the sequence. Since the operations are deterministic, nodes are guaranteed to obtain the same state upon the execution

of the j-th operation in the sequence. Here, the sequence of operations is exactly the total ordering O required by the linearizability correctness property [30].

The reason that SMR's ordering is "*stable*" is that the position of an operation in the sequence will not change; otherwise, the linearizability guarantee could be violated. In comparison, the ordering of shared registers may change due to asynchrony. In the example above, node i may initially think that o_{24} is the first write. It is only until that i receives o_8, it learns that o_{24} should occur later in the ordering. However, because of the semantic of shared registers, such a non-stable ordering still suffices for linearizability.

The difference between stable and non-stable ordering also makes it difficult to use atomic shared registers to implement an SMR system.

2.3 Challenges

RMW and Gryff's solution. Gryff is focused on one-shot transactions that access only a single key, namely RMWs. RMWs require a stable ordering because its outcome depends on a previous state. Using the terminology introduced in §2.1, a previous state is the values of the keys specified in the second parameter K_{input}. Without a stable ordering of operations, the outcome of applying RMWs may lead to inconsistency.

More formally, suppose $K_{input} = K_{output} = \{k\}$ and $f(\cdot)$ defines a compare-and-swap (CAS):

$$f(old, new) = \begin{cases} new, & \text{if value of } k = \text{old} \\ \text{value of } k, & \text{otherwise} \end{cases}$$

Consider the case of a write o_8 that updates the value of key k to 8, and a RMW o_{24} with $f(\text{empty value}, 24)$. Now the order of how nodes apply these two operations matters. If the ordering is not stable, it is possible that node i applies o_8 before o_{24} and node j applies o_{24} before o_8. As a result, two nodes will have divergent states, even if they eventually receive and apply both operations.

Gryff [10] uses a novel design, called Consensus-After-Register timestamps (or carstamps), to enforce the stable ordering of RMWs, while allowing writes and reads to order in a non-stable manner. Gryff uses ABD [8] to order reads and writes (the "register" part), and uses EPaxos to order RMWs (the "consensus" part). The RMWs are ordered after some writes (the "after" part).

With carstamp, Gryff then uses EPaxos to keep track of dependency and execution order. In the prior example, assume o_{24} has observed o_8. By design, every node that applies o_{24} must wait until it has applied o_8. This ensures linearizability, but results into *blocking* execution in Gryff's RMWs.

Multi-Key Operations. Next, we discuss why it is non-trivial to extend ABD and Gryff to handle operations that may access or update multiple keys, namely multi-key operations. Figure 1 presents an example that shows why reading and writing each key separately using ABD is insufficient to support scan operations, i.e., an operation that returns the value of multiple keys. In the discussion below, let $\text{scan}(x, y)$ represent a scan operation that returns the values for keys x and y atomically.

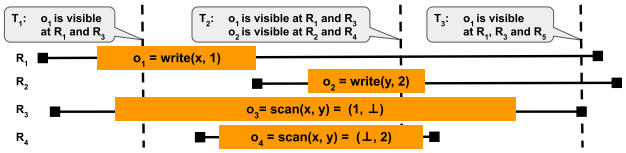


Figure 1: **An example of four operations in a five-node system.** The left dot and right dot represent the invocation time and completion time, respectively.

In this strawman example, a $\text{scan}(x, y)$ performs two ABD reads in parallel – one reading key x and the other reading key y. Suppose there are five replicas (R_5 is not shown for brevity) and four operations o_1 to o_4 such that:

- At time T_1: the write o_1 is visible only at R_1 and R_3;
- At time T_2: the write o_1 is visible only at R_1 and R_3, and the write o_2 is visible only at R_2 and R_4;
- At time T_3: the write o_1 is visible at R_1, R_3 and R_5; This is because the put phase of the ABD reads of o_3 helps propagate the value to a quorum, even though the operation o_1 has not completed at this point.

Now, consider the two scans:

- Scan o_3 returns $(x, y) = (1, \perp)$. This is possible if o_3 completes the get phase of two parallel ABD reads at T_1 with quorum R_1, R_3, R_5.
- Scan o_4 returns $(x, y) = (\perp, 2)$. This is possible if o_4 completes the get phase of the two parallel ABD reads at T_2 with quorum R_2, R_4, R_5. These nodes only know o_2 but not o_1, as at T_2, only R_1 and R_3 know o_1.

Consequently, we have "*incomparable*" operations. That is, in the total ordering that one needs to construct for linearizability, o_3 cannot be placed before nor after o_4. This is because such an incomparable scenario is impossible on a single machine.

This example can be extended for any algorithm, including Gryff, that does not perform coordination between different keys. Pineapple addresses this issue by relying on the leader to decide the order of multi-key operations. For example, if R_3 is the (Multi-Paxos) leader which orders o_3 before o_4, then o_4 will return $(1, 2)$, making the two operations comparable.

3 Pineapple: Design and Analysis

This section presents the design and analysis of Pineapple. We focus on the operations on a single key k and the common case when there is no leader change. That is, the one-shot transactions defined in §2.1 have $K_{input} = K_{output} = \{k\}$. Note that in this case, Pineapple provides the same semantic as Gryff. Hence, we will use "RMWs" and "one-shot transactions" interchangeably in the following presentation. Appendix A presents the generalization to multiple keys and how Pineapple handles the case when leader changes.

Algorithm 1 Pineapple: ABD Component at Node N_i

Write($key, value$) invoked by Writer Proxy N_i:
1: *// Get phase: obtain tag*
2: **send** $\langle$GET-TAG$, key\rangle$ **to all**
3: **wait until** receiving $\langle$ACK-TAG$, key, ps_j\rangle$ **from quorum** Q
4: $ps^{max} = (tag^{max}, slot^{max})$ is largest pstamp received from all $j \in Q$
5: $ps^{new} \leftarrow ((tag^{max}.ts + 1, i), slot^{max})$
6: *// Put phase: propagate write request*
7: **send** $\langle$PUT$, key, ps^{new}, value\rangle$ **to all**
8: **wait until** receiving $\langle$ACK-PUT$, key\rangle$ **from quorum**

Read() invoked by Reader proxy N_i:
9: *// Get phase: obtain value and tag*
10: **send** $\langle$GET-VALUE$, key\rangle$ **to all**
11: **wait until** receiving $\langle$ACK-VALUE$, key, value_j, ps_j\rangle$ **from quorum** Q
12: $ps^{max} = (tag^{max}, slot^{max}) \leftarrow$ largest pstamp received from all $j \in Q$
13: $v^{max} \leftarrow$ value associated with ps^{max}
14: *// Put phase: propagate learned value*
15: **send** $\langle$PUT$, key, v^{max}, ps^{max}\rangle$ **to all**
16: **wait until** receiving $\langle$ACK-PUT$, key\rangle$ **from quorum**
17: return v^{max}

Background Handlers at node N_i:
18: **Upon receiving** $(\text{GET-TAG}, key)$ **from** N_j:
19: **send** $\langle$ACK-TAG$, key, Storage_i[key].ps\rangle$ **to** N_j
20: **Upon receiving** $(\text{PUT}, key, ps_j, value_j)$ **from** N_j:
21: **if** $Storage_i[key].ps < ps_j$ **then**
22: $\quad Storage_i[key].ps \leftarrow ps_j$
23: $\quad Storage_i[key].value \leftarrow value_j$
24: **send** $\langle$ACK-PUT$, key\rangle$ **to** N_j
25: **Upon receiving** $(\text{GET-VALUE}, key)$ **from** N_j:
26: **send** $\langle$ACK-VALUE$, key, Storage_i[key].value, Storage_i[key].ps\rangle$ **to** N_j

Algorithm 2 Pineapple: Multi-Paxos Component at Leader N_l

RMW($f(\cdot), \{key\}, \{key\}$) invoked by Pineapple Leader N_l:
1: *// Get phase: obtain most recent values*
2: **send** $\langle$GET-VALUE$, key\rangle$ **to all**
3: **wait until** receiving $\langle$ACK-VALUE$, key, value_j, ps_j\rangle$ **from quorum** Q
4: $v^{max} \leftarrow$ value w/ largest ps^{max} received from all $j \in Q$ ▷$l \in Q$
5: $v^{new} \leftarrow f(v^{max})$ ▷`Apply RMW`
6: $ps^{new} \leftarrow (ps^{max}.tag, slot_i + 1)$ ▷`Increment slot index`
7: *// Put phase: propagate output*
8: **send** $\langle$PUT$, key, v^{new}, ps^{new}\rangle$ **to all**
9: **wait until** receiving $\langle$ACK-PUT$, key\rangle$ **from quorum**

3.1 Pineapple: Design

Pstamps: Timestamp for Pineapple. Similar to how Gryff uses carstamps, Pineapple uses a variation of timestamp – pstamps (which stands for Pineapple timeSTAMPs). Pstamps is a composition of tag used by ABD and the slot index uses by Multi-Paxos. Formally, pstamp p is a tuple $(tag, slot)$, where

- tag is an ABD tag – a tuple (ts, id) such that ts is the logical timestamp and id is the ID of the writer proxy;
- $slot$ is a positive integer that represents the slot index of the Multi-Paxos component.[3]

As in [8, 10, 42, 63], each write or RMW is associated with a unique pstamp, and Pineapple uses pstamps to determine the order of operations. These pstamps can be ordered lexicographically. That is, for two pstamps $p = (tag_p, slot_p)$ and $p' = (tag_{p'}, slot_{p'})$, we say that $p > p'$ if and only if:

- $tag_p.ts > tag_{p'}.ts$;
- $tag_p.ts = tag_{p'}.ts$ and $tag_p.id > tag_{p'}.id$; or
- $tag_p.ts = tag_{p'}.ts$ and $tag_p.id = tag_{p'}.id$ and $slot_p > slot'_p$

Interaction between ABD and Multi-Paxos. In Pineapple, ABD handles reads and writes, whereas Multi-Paxos takes care of RMWs. These two components use pstamps to interact with each other – both use pstamps to determine which operation is more recent.

ABD runs as usual by using the tag part in pstamps. Writes only updates the tag part, whereas reads need to use the ordering of pstamps to decide which value to return. The value returned could be a value generated by a write or a RMW.

Multi-Paxos in Pineapple needs to do one more round of communication, compared to the original Multi-Paxos. This is because the leader might not be in the quorum for an ABD write. More concretely, the leader performs the following steps to complete a RMW with $f(\cdot)$:

- Contact a quorum Q, which includes the leader itself, to retrieve the value v^{max} that has the largest pstamp ps^{max};
- Apply RMW by computing $f(v^{max})$. Let the output be v^{new}; and
- Compute new pstamp ps^{new} as $(ps^{max}.tag, slot_i + 1)$.
- Transmit (v^{new}, ps^{new}) to all and wait for an acknowledgement from a quorum.

The comparison of pstamps is defined in §3.1. Moreover, the quorum Q must include the leader itself, as in the common case with a stable leader, the leader has the most recent information about prior RMWs. This is also the reason that in the third step, the leader can update the pstamp by incrementing the local slot index $slot_i$ directly.

In the case when leader fails or changes, the leader N_l will use the meta-data of Multi-Paxos, e.g., ballots (not presented for conciseness), to detect such a scenario. Then Pineapple will need to participate in Phase 1 (the prepare phase) of Multi-Paxos to figure out the next leader. In this period, some operations might not be able to make progress. Same as other consensus algorithms, e.g., Multi-Paxos and Raft, Pineapple requires partial synchrony to elect a new stable leader and ensure liveness. Appendix A presents more details.

Furthermore, ABD reads in Algorithm 1 needs to be modified slightly so that they do not return inconsistent results when a leader change. Appendix A.2 discusses the extension.

Pineapple: Pseudo-code and Correctness. Algorithm 1 presents the ABD component of Pineapple. The logic is iden-

[3] In the full version of Pineapple, the slot index also needs to include necessary meta-data for Multi-Paxos, e.g., the ballot and leader ID. Such information is needed for handling leader changes. To keep our discussion concise, the second field is a single integer.

tical to the one described in §2.2, except that Pineapple uses pstamps except for tags. For each key *key*, $Storage_i[key]$ keeps two fields:

- $Storage_i[key].value$: the current value for key *key*, and
- $Storage_i[key].ps$: the associated pstamp.

Pineapple also adopts various optimizations mentioned in [10, 25, 42]. As a result, Pineapple is able to always complete reads in 1 RTT when $n = 3$.

Algorithm 2 presents the Multi-Paxos component, assuming that there is a stable leader N_l. Compared to the original Multi-Paxos, Pineapple needs to do the Get phase to learn the most recent value for the $K_{input} = \{key\}$. The Put phase is basically the same as Phase 2 (the propose phase) in Multi-Paxos, which updates a quorum (of acceptors) with the value generated by the RMW.

Analysis. Since both ABD and Multi-Paxos use a majority quorum, the quorum size is $\lfloor n/2 \rfloor + 1$. As in prior optimizations, ABD can complete on the fast path when the quorum returns the same response. RMW takes 3-RTT if the client is not co-located with the leader. All the operations have non-blocking execution, because they can be executed upon learning a decision either from a quorum or a leader.

Correctness. On a high-level, the correctness follows from three observations: (i) the correctness of ABD to order reads and writes; (ii) the correctness of Multi-Paxos to order RMWs; and (iii) the usage of pstamps to ensure that if a RMW o is visible to any read, then any future operation must also observe o or any operation that has pstamps larger than o's. Appendix B presents the full proof.

We use examples from §2 to provide an intuition on the correctness. First, RMWs go through the leader; hence the ordering is stable. In particular, if the leader sees o_8 when applying RMW o_{24} in the Get phase (Line 1 in Algorithm 2), then other nodes will order o_8 before o_{24}. The reasons that this ordering is stable are: (i) the leader decides the order, which holds true by the design of Multi-Paxos, even when there is a leader change; and (ii) the pstamps correctly capture the fact that o_{24} happens after o_8 – which is due to the design that the leader increments slot index (Line 6 in Algorithm 2).

Second, multi-key operations, such as scans and MSETs, are also ordered by the leader. Therefore, inconsistency (or incomparability) in Figure 1 is impossible. This is intuitively because that all nodes will observe the sequence (of the slots) following the stable order determined by the leader.

In Figure 1, both scans o_3 and o_4 are served by the leader N_l. Suppose N_l observes o_1, but not o_2 in the Get phase, so it returns $(1, \perp)$ for o_3. When serving o_4, N_l can only return $(1, \perp)$ or $(1, 2)$. This is because the quorum Q at Line 3 in Algorithm 2 must contain N_l itself. In other words, $(\perp, 2)$, an incomparable result in the example from §2, can never be returned by N_l, as it has already observed o_3.

Finally, the reason that writes can be ordered in a non-stable manner is because of the tag part of the pstamps and its semantic. If a node i receives a write with a pstamp larger than its local value, then i can safely overwrite its local value; otherwise, the write can safely be discarded. Furthermore, even though the ordering of concurrent writes is non-stable in general, whenever the leader observes them, the relative ordering between writes and RMWs becomes stable.

More concretely, consider a write w concurrent with a RMW r. We have two cases:

Case I: If the leader observes w during the Get phase, then r must have a larger pstamp, which allows all nodes to ignore w, after applying r. In other words, w *cannot* overwrite r.

Case II: If the leader does not observe w, then there are two subcases: (i) w's pstamp is larger than o's, or (ii) there is another write w' whose pstamp is larger than w and has the same tag part as in r's pstamp.

- In (i), the result of r will be overwritten by w. (The total ordering is r, w.)
- In (ii), w will be discarded by nodes that have applied r. (The total ordering is w, w', r)

In both subcases, linearizability and the semantics are satisfied, because the leader's observation in the Get phase makes the relative ordering among w, w' and r stable.

4 Evaluation

We aim to compare Pineapple with other systems that have similar goals and target general environments that are appropriate for Multi-Paxos. In other words, we target systems that do not make specific assumptions on the hardware or networks. Recall that §1 mentions four categories of solutions that can be used to improve consensus-based linearizable storages. The purpose of our evaluation is focused on category (iv) – understanding the performance of new algorithmic components that are integrated with existing consensus algorithms, since systems in other categories typically adopt different architectures and/or assumptions (more discussion in §5).

We compare Pineapple with five other systems – Multi-Paxos, Multi-Paxos with leader lease, PQR [13], EPaxos [46], and Gryff [10]. We evaluate throughput and latency in both local-area network (LAN) and wide-area network (WAN), with the focus on geo-replicated, large-scale web applications. For these evaluations, we focus on tail latency, because user-perceived latency in these applications is highly correlated with the tail latency of the underlying storage systems [7, 19, 52, 58]. Finally, we compare Pineapple with the optimized Raft in a production system etcd.

4.1 Implementation and Experiment Setup

Implementation. We implement Pineapple, PQR [13], and Multi-Paxos with leader lease in Go using the framework of EPaxos, Gryff and Multi-Paxos from [10, 46] to ensure a fair comparison. In the discussion below, we will use "Paxos

	CA	VA	IR	OR	JP
CA	0.2				
VA	72	0.2			
IR	151	88	0.2		
OR	59	93	145	0.2	
JP	113	162	220	121	0.2

Table 2: **RTT (in ms) between VMs in emulated geographic regions** [10]. For $n = 3$, we use region CA, VA, and IR.

systems" to refer to Multi-Paxos, Multi-Paxos with lease and PQR, as they have similar performance pattern in most cases.

Leases [27, 36, 65, 67, 73] are practical optimizations to reduce communication overhead, by requiring *additional assumptions* on timing (or synchrony). We implement the version of leader lease from [66, 67], which assumes a known bound on clock drift so that only one node holds a lease at a time. In the implementation, the leader can serve reads *without* communicating with another node, while holding the lease. Since we focus on the common case, we assume that the leader has the lease at all time during our evaluation.

We do *not* enable thrifty optimization *nor* batching, because these optimizations generally increase tail latency due to a higher chance of conflicts [10,46,61]. Thrifty operations send some messages to a necessary quorum only; however, it makes EPaxos easier to build a long chain of dependency, and makes nodes wait longer before an operation can be execugted. Since we are targeting large-scale customer-facing applications, we focus on configurations that favor low tail latency.

We also implement Pineapple in etcd by replacing the Raft algorithm. This allows us to study how Pineapple interacts with the other layers of a production key-value system. In addition, etcd persists the states to disk; hence, etcd experiments also allow us to investigate the impacts of persistence.

Testbed. We run our experiments on CloudLab [21] using c6525-25g (AMD 7302P at 3.00GHz, 16 cores, 128 GM RAM) for both node and client VMs. For LAN, the ping latency is around 0.15ms and the configured link bandwidth is 10 Gbps. For WAN, we adopt the same latency profile used in [10,63] – (i) $n = 3$: nodes in California (CA), Virginia (VA), and Ireland (IR); and (ii) $n = 5$: two more nodes in Oregon (OR) and Japan (JP). The WAN latencies are emulated using Linux's Traffic Control (tc) by adding delays to packets on all VMs with filters on different IPs. Table 2 shows the configured RTT between nodes and clients in different regions. These numbers were chosen to represent typical RTT between the corresponding Amazon EC2 availability regions [10, 63].

Experiment Setup. The systems are configured to commit and execute the operations before responding to client requests (close-loop clients). Since as observed in [10, 61, 63], most web applications depend on the confirmation returned by an operation. For example, both Redis [2] and ZooKeeper [32] return results for both reads and writes.

Each experiment is run for 180 seconds, and we collect statistics in the middle of 150 seconds. The statistics collected in this period are quite stable, after the removal of fluctuating numbers during warm-up and cool-down. Following prior setups [10, 46, 61, 63], each object is of 16B. This is because 16B gives the best performance for EPaxos and Gryff. Furthermore, this also reflects real-world observation. For example, the report from Facebook [7] suggests that 40% of the data in their Memcached workload is less than 11B. In Facebook's production TAO system [9, 52], 50% of its requests have the value field smaller than 16B. When evaluating Pineapple in etcd, we test it with a larger object size at 100B.

Following [10, 13, 46, 61, 63], all systems we test store data in memory, except for the etcd experiments. This strategy is reliable as long as the number of *concurrent* node failures is bounded, which is a common case with $n = 2f + 1$ [15, 17, 24]. Moreover, recent hardware solutions are being developed and adopted to prevent data loss from crashed machines, e.g., persistent memory or disaggregated memory [18, 62, 70, 72].

Two operations are said to be "*conflicting*" with each other if they are targeting the same object (or same key) [10, 15, 46, 61, 63]. A conflict rate p means that (i) with a probability p, a client chooses the same key that is defined in advance, and (ii) with probability $1 - p$, a client chooses a unique key.

Our evaluation does not use Zipfian distribution [16, 61], except when using YCSB for etcd experiments, because we want to pick favorable configurations for EPaxos and Gryff. Due to their execution logic, they suffer performance degrade under Zipfian workloads [10, 61].

4.2 Summary of Our Findings

We aim at answering the following questions:

- Does Pineapple improve maximum attainable throughput in both LAN and WAN? (§4.3)
- How do the read/write/RMW latency of Pineapple compare against state-of-the-art systems? (§4.4)
- How does Pineapple perform as the consensus layer in a production system etcd? (§4.5)

To summarize our findings,

- In LAN, Pineapple improves throughput by 10% – 20%, over the closest competitor (EPaxos) when the RMW ratio is less than 20%. In WAN, Pineapple has around 3.5x throughput, over the closest competitor (PQR and Multi-Paxos with lease) in a balanced workload.
- In WAN, Pineapple generally has better read tail latency than Paxos systems in 3- and 5-node deployments under web application workloads [10, 16]. Pineapple has better RMW tail latency than Gryff and EPaxos in a 3-node deployment under workloads with 25% conflict rate.
- We compare the performance of Pineapple and an optimized Raft in etcd [1]. Our evaluation using YCSB [16] shows that Pineapple reduces median latency by 20% to 50% in a balanced workload in LAN.

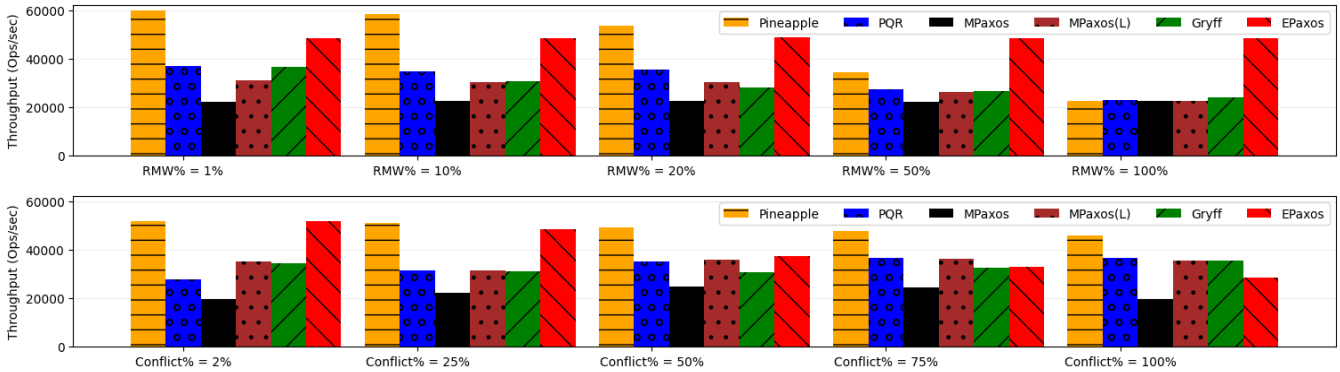


Figure 2: **Maximum attainable throughput in 5-node LAN** ($n = 5$, 49.5% reads, 49.5% writes). Top row varies RMW ratios with conflict rate 25%. Bottom row varies conflict rates with RMW ratio = 20%. MPaxos is the label for Multi-Paxos, whereas MPaxos(L) is the label for Multi-Paxos with leader lease.

4.3 Maximum Attainable Throughput

To achieve maximum throughput, we load each system up to saturation and we collect the median throughput during the period around saturation.

Throughput in LAN. Figure 2 shows the maximum throughput with 1% to 100% RMWs ratio. The remaining operations consist of an equal number of reads and writes. This reflects the balanced workload of YCSB-A [16]. In addition, the numbers help estimate the impact of RMWs and the bottleneck of the leader in Paxos systems and Pineapple.

In this experiment, $n = 5$. Each system requires different number of close-loop clients to saturate (ranging from 150 to 300 clients). In Pineapple, PQR, EPaxos, and Gryff, the clients are evenly distributed so that each node handles the same number of clients. In other two systems, the clients only communicate with the leader node. In PQR and Pineapple, some clients need to communicate with both the leader and one follower node. Top row has conflict rate = 25% so that we capture the effect of conflicting operations.

With RMW ratio $\leq 20\%$, Pineapple has 10% – 20% improvement over EPaxos, the closest competitor. When RMW ratio is 100%, Pineapple is slightly slower than Paxos systems, because RMWs go through the leader and Pineapple's leader needs to process more messages for RMWs.

EPaxos, on the other hand, remains stable throughout different ratios. It has best performance with higher RMW ratios, at around 2.2x over other systems. Compared to Pineapple and Paxos systems, EPaxos's leaderless design improves throughput. Compared to Gryff, it has simpler execution logic.

Bottom row demonstrates that all systems under test, except for EPaxos, have stable throughput under different conflict rates. This is because with higher conflict rates, EPaxos's throughput is affected by its growing dependency graph. Gryff is less affected because only RMWs need to resolve dependency. Multi-Paxos has the worst throughput, across all conflict rates, because of the saturated leader.

Throughput in WAN. In WAN, Pineapple has 4x to 10x throughput, compared to EPaxos, in read-heavy workloads (a variant of YCSB-B), and around 3.5x throughput in balanced workloads (a variant of YCSB-A). This is because as shown in [10, 61], EPaxos's dependency graph is easier to grow in size in WAN, which significantly affects its throughput. Due to page limits, we present details in Appendix C.

In read-heavy workloads, "Multi-Paxos with lease" has best performance, with 1.25x to 1.3x throughput, compared to Pineapple's. This is because with lease, the leader does not need to process as many messages as Pineapple does. In balanced workloads, Pineapple has best performance, which has 3x to 4x throughput improvement over the next closest competitor (PQR and "Multi-Paxos with lease"). This is because the leader in Paxos systems is saturated by writes and RMWs, whereas writes are evenly distributed in Pineapple.

4.4 Latency

4.4.1 Latency-Throughput in LAN

Figure 3 plots the throughput against median (p50) and 90-percentile (p90) latency for all operations by increasing the number of concurrent closed-loop clients (3 to 360) in LAN with $n = 5$. We use a variation of YCSB-A balanced workload with 1% RMWs from [10] and 25% conflicts to measure the impact of RMWs and concurrent operations. Pineapple has around 20% improvement in throughput at saturation. Pineapple in general has smallest p50 and p90 latency across all loads. Gryff has low latency but has limited throughput. This demonstrates that compared to Gryff, the usage of Multi-Paxos in Pineapple effectively improves performance

4.4.2 Latency in WAN

Read-heavy Workload with $n = 3$**.** Same as in [10], we use a variant of YCSB-B [16] workload that contains 94.5% reads,

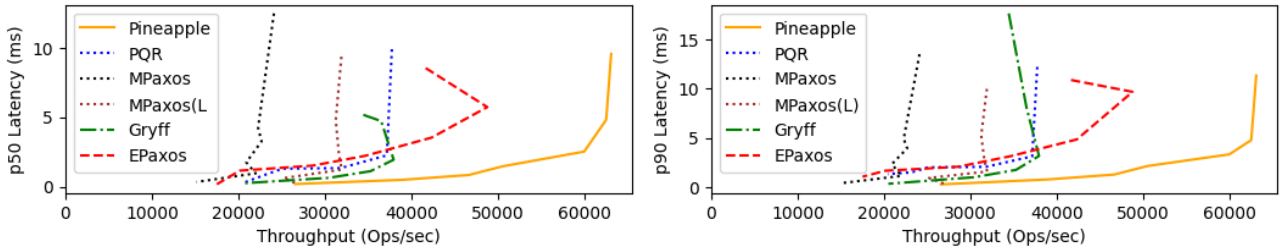
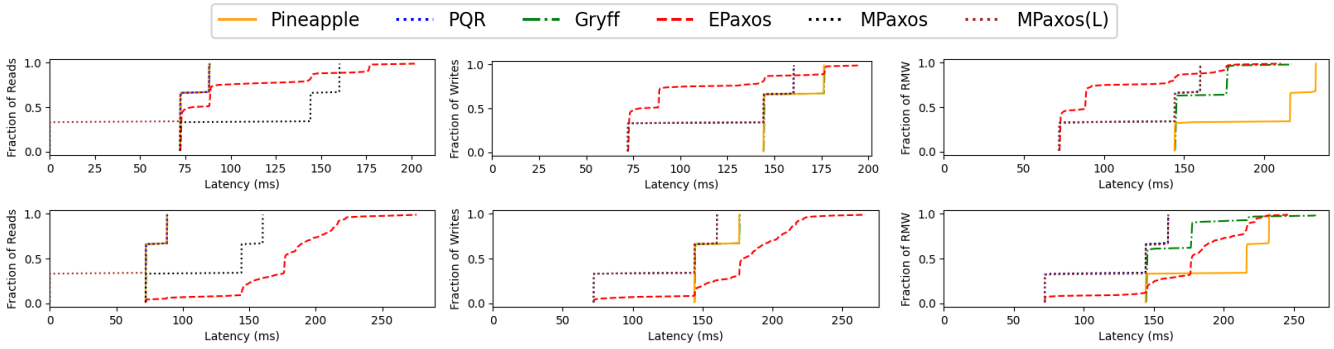


Figure 3: **Latency vs. Throughput in LAN** ($n = 5$, 49.5% reads, 49.5% writes, 1% RMWs, 25% conflict rate). Left figure presents median (p50) latency, whereas right figure presents 90-percentile (p90) latency.



(a) **Reads** (b) **Writes** (c) **RMWs**

Figure 4: **Latency CDF** ($n = 3$, 94.5% reads, 4.5% writes, 1% RMWs). Top row has 2% conflict rate, whereas bottom row has 25%. Both Gryff and Pineapple complete reads in 1 RTT when $n = 3$. Leader of Paxos systems and Pineapple is in VA region.

4.5% writes, and 1.0% RMWs. The focus is on a read-heavy workload to reflect the pattern of web applications [9, 10, 16, 17]. In this set of experiments, the leader node is located in the VA region (c.f. Table 2). This setup is favorable to Paxos systems, as the leader is in the *nearest* quorum for both CA and IR. Following [10], each region has 16 concurrent clients. Figure 4 shows the latency CDF (cumulative distribution function) of all three operations with $n = 3$.

1-RTT Reads in Gryff and Pineapple. For $n = 3$, both Gryff and Pineapple always complete reads in 1 RTT. Figure 4a shows that an operation is completed after nodes spend 1 RTT to the nearest quorum, regardless of the conflict rate.

As in [10, 46, 61], clients and nodes are co-located; hence, RTTs between them is negligible. Clients in VA are closest to the nodes in VA and CA and vice versa for clients in CA. This results in around 66% of the reads completing in the RTT between VA and CA (72 ms). Clients in IR take 88 ms – these are the tail latency for reads.

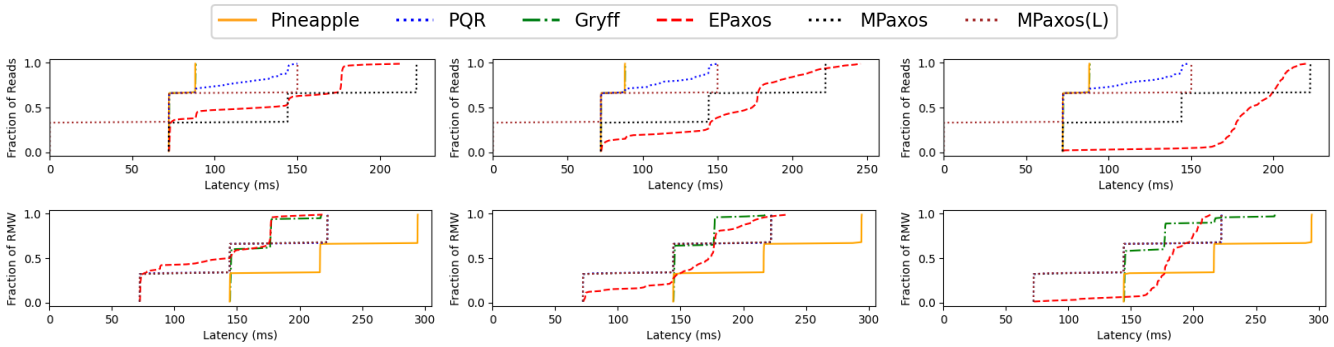
Stable Performance in Paxos Systems. The leader is in the VA region; hence, the blocking execution of PQR's reads is negligible. This is because the leader is always in the closest quorum. For both writes and RMWs, PQR takes 1 to 2 RTTs to complete. The tail latency is at 160 ms, which occurs when the clients in IR contact the leader in VA (88 ms), which then contacts the nearest quorum (72 ms). $1/3$ of the reads in "Multi-Paxos with lease" have latency close to 0, because the leader can serve them immediately.

Tail latency of RMWs. Pineapple's RMWs take 3 RTT at follower nodes. Thus, the tail latency in Figure 4c is $72 * 2 + 88 = 232$, where the first part is the time from VA to the nearest quorum (CA) and the second part is the time the IR clients take for contacting the leader in VA.

With 2% conflicts and low percentage of non-read operations, Gryff has relatively stable RMW performance. With 25% conflicts, Pineapple has roughly 30 ms improvement at p99 latency relative to Gryff, because the execution of RMWs in Gryff is blocking. EPaxos in general has worse tail latency, due to its dependency tracking.

Blocking Operation Execution. Figure 5 presents the latency CDF for reads and writes under the same workload. To better demonstrate the blocking effect, the leaders are located in CA. PQR's reads cannot complete in 1 RTT, because the leader is not in IR's closest quorum. As a result, the tail latency of PQR is around 70 ms longer than Gryff's and Pineapple's.

Gryff's RMWs has longer tail latency under higher conflicts, because of longer dependency. In this case, due to Pineapple's 3-RTT RMWs, its tail latency is still the largest under low workload. Figure 6 uses exactly the same configuration as in the bottom row of Figure 5c, except with a higher number of clients at 250 clients per region. More concurrent



(a) 2% **conflict rate** (b) 10% **conflict rate** (c) 25% **conflict rate**

Figure 5: **Latency CDF** ($n = 3$, 94.5% reads, 4.5% writes, 1% RMWs). Top row plots the latency CDF for reads, whereas bottom row plots the latency CDF for RMWs. Leader of Paxos systems and Pineapple is in CA region.

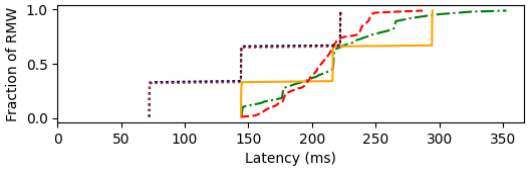


Figure 6: **Latency CDF** ($n = 3$, 94.5% reads, 4.5% writes, 1% RMWs, 250 clients)

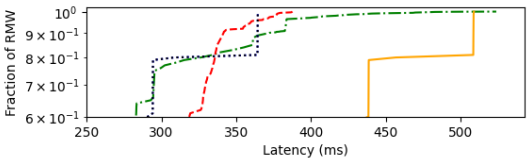


Figure 7: **Log-scale Latency CDF** ($n = 5$, 49.5% reads, 49.5% writes, 1% RMWs, 25% conflict rate, 120 clients).

operations lead to longer RMW tail latency in Gryff.

Balanced Workload with $n = 5$**.** We next use the same variant of YCSB-A from [10], which consists of 49.5% reads, 49.5% writes, and 1.0% RMWs with 2% and 25% conflicts. The balance between reads and writes and a higher conflict rate allow us to quantify the impacts of interleaving operations on latency. Figure 8 shows latency CDF for each operation type for $n = 5$. The leader is in the IR region.

1-RTT Reads. For $n = 5$, Gryff often completes reads in 1 RTT with 2% conflict rate (Figure 8a). Pineapple has slightly longer read tail latency, because Pineapple's slower RMWs increase the chance of concurrent operations. When conflict is low, Pineapple completes reads faster than Paxos systems.

2-RTT Writes. As indicated in Table 1, in the worst case, all systems have 2-RTT writes; however, the latency pattern is different as it depends on where the clients are located (Figure 8b). For Gryff and Pineapple, the furthest site is IR; hence, 2 RTTs to the nearest quorum take $2 * 145 = 290$ ms. In comparison, in Paxos systems, the tail latency for JP clients is $145 + 220 = 365$ ms.

Trade-offs. Pineapple has slower RMWs but faster writes and reads than Paxos Systems and EPaxos. It has longer read tail latency than "Multi-Paxos with lease" and PQR with 25% conflicts. Gryff has best latency in most cases under low workload. As a trade-off, Pineapple has better throughput. In these balanced workloads, Pineapple has 3x and 4x throughput, compared to closest competitors (c.f. Appendix C).

Figure 7 uses an identical configuration as in Figure 8, except with 120 clients per region. Under 25% conflicts, Gryff in fact has longer RMW tail latency than Pineapple.

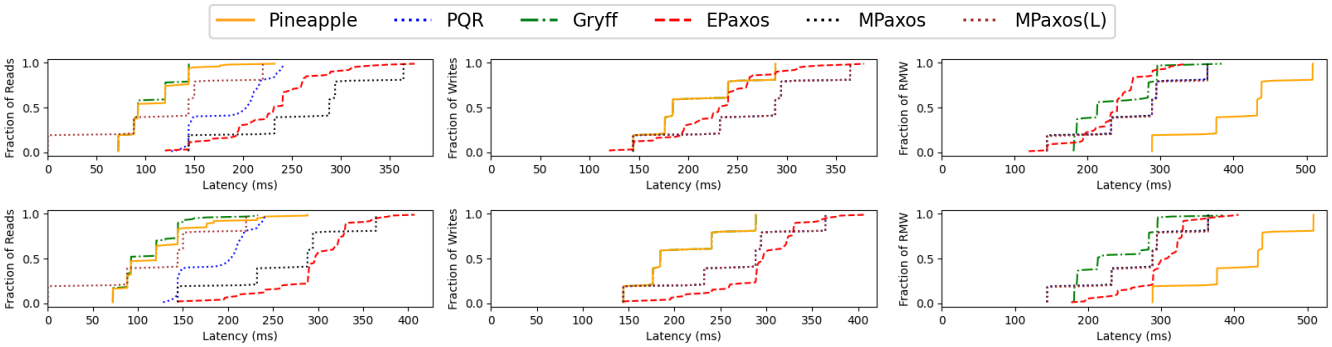
4.5 Performance in Real-World System

We evaluate how Pineapple compares to optimized implementation of Raft in a production key-value storage designed for web applications. We replace the consensus layer in etcd by Pineapple. These experiments use a 5-node etcd deployment with YCSB client (Zipfian workloads with default key-value size 100KB). Each configuration runs roughly 1 to 2 minutes.[4] YCSB's threadcount ranges from 2 to 90. In our experience, this number allows etcd to have the most stable performance.

Figure 9a plots the throughput against p50 and p90 latency for all operations under YCSB-C (100% reads). Raft in etcd adopts the heartbeat-based batching optimization for improving read latency; hence, we use YCSB-C to see how Pineapple compares to a practical optimization. Figure 9b uses a variant of YCSB-A workload with 50% reads, 25% writes/RMWs, which allows us to see the impact of Pineapple's 3-RTT RMWs. Figure 9c uses a variant of YCSB-B workload with 95% reads and 2.5% writes/RMWs.

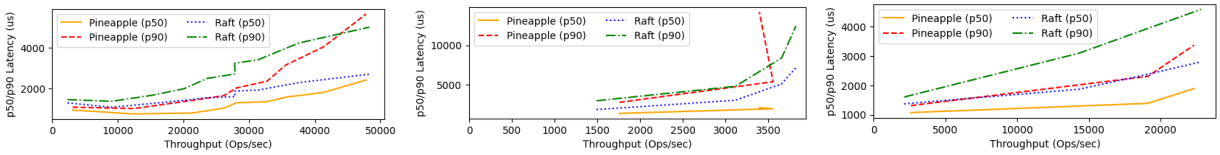
Figure 9a shows that before saturation, Pineapple has around 20% improvement in both p50 and p90 latency in all-read workload. Figure 9b demonstrates that before saturation, Pineapple has 50% improvement in p50 latency. Pineapple

[4]YCSB only allows the configuration on number of operations; hence, the experiment time varies.



(a) **Read** (b) **Write** (c) **RMW**

Figure 8: **Latency CDF** ($n = 5$, 49.5% reads, 49.5% writes, 1%RMW). Top row has 2% conflict rate, whereas bottom row has 25% conflict rate. Leader of Paxos systems and Pineapple is in IR region.



(a) YCSB-C (100% reads) (b) 50% reads, 25% writes/RMWs (c) 95% reads, 2.5% writes/RMWs

Figure 9: **Latency vs. Throughput in etcd**. ($n = 5$, Zipfian workloads)

has higher p90 latency and lower throughput mainly because of the high RMW ratio. However, Pineapple's p90 latency is within 10% and throughput within 15% of Raft. This indicates that Pineapple's RMWs has mild performance impact, when interacting with other components in a real-world system. Figure 9c shows that under read-heavy workload, Pineapple has a similar performance pattern, with 40% to 50% reduction in both p50 and p90 latency. In conclusion, offloading operations effectively reduces latency, even compared to an optimized implementation of Raft.

In these experiments, Pineapple's throughput is not higher, even when we observe the effect of saturated leader. This is mainly because Raft's read optimization enables an efficient disk batching (which is not implemented in Pineapple).

5 Related Work

Consensus is useful for making a service fault-tolerant [59]. Classical consensus uses a single leader, e.g., [33, 36, 39, 53]. Recent systems adopt a multi-leader design that distributes the ordering responsibility evenly to increase performance (e.g., Mencius [44], EPaxos [46], M^{2}Paxos [55], Caesar [6], ATLAS [24]). The multi-leader design uses a graph for keeping track of dependency, which often leads to long tail latency [57].

Recent systems make improvements by utilizing modern assumptions or hardware. For example, Copilot is focused on tolerating slow leader [51], Rabia on stable network [54], Mu [5] on RDMA, Speculative Paxos [56] and NOPaxos [40] on SDN, and Tempo [23] considers a small number of concurrent failures. Other systems focus on scenarios different from Pineapple, e.g., Giza [15], Pando [64] and CRaft [68] are focused on erasure coded storages. Nil-externality [26] utilizes operations that modify state within a storage but has no externalized effects. Clock-RSM [20], Caesar [6], EPaxos-TOQ [61] rely on loosely synchronized clock

Another category of systems target communication. For example, read leases allow reads to complete without any inter-replica communication [28, 47]. This design generally leads to slower writes. Compartmentalized Paxos [69] adopts the idea of proxy nodes to bypass various communication and computation bottlenecks. PigPaxos [14] uses piggybacking to reduce communication. These systems potentially require more resources or a better network infrastructure.

All these systems could potentially achieve better performance than Pineapple, under a right set of assumptions. Our focus is on general deployments targeted by Multi-Paxos and Raft, including both LAN and WAN, *without* specific assumptions on the environment or modern hardware.

Transactional systems, such as Janus [49], Calvin [60], and TAPIR [71]), provide richer semantics compared to Pineapple. In fact, Pineapple and Gryff are specifically designed and optimized for key-value storages with more restrictive operation semantics. Consequently, general transactional systems are facing more bottlenecks when serving single-key reads/writes and could have longer latency in general. For example, Janus requires the leader to resolve conflicting transactions, and Janus's use of dependency tracking (for reordering conflicting transactions) could also result in longer tail latency.

References

[1] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/, accessed May 2021.

[2] RedisRaft: Strongly-consistent Redis deployments. https://github.com/RedisLabs/redisraft, accessed December 2020.

[3] Replication Layer | CockroachDB Docs. https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html, accessed December 2020.

[4] RethinkDB 2.1: high availability. https://rethinkdb.com/blog/2.1-release/, accessed December 2020.

[5] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 599–616, 2020.

[6] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 49–60. IEEE Computer Society, 2017.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale, editors, *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012.

[8] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: facebook's distributed data store for the social graph. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 49–60. USENIX Association, 2013.

[10] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 591–617. USENIX Association, 2020.

[11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. Association for Computing Machinery.

[12] Wei Cao, Yusong Gao, Feifei Li, Sheng Wang, Bingchen Lin, Ke Xu, Xiaojie Feng, Yucong Wang, Zhenjun Liu, and Gejin Zhang. Timon: A timestamped event database for efficient telemetry data processing and analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 739–753, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Linearizable quorum reads in paxos. In Daniel Peek and Gala Yadgar, editors, *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*. USENIX Association, 2019.

[14] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 235–247. ACM, 2021.

[15] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 539–551. USENIX Association, 2017.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein,

Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[18] Intel Corporation. Intel optane persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.

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

[20] Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 343–354. IEEE Computer Society, 2014.

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

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[23] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 178–193. ACM, 2021.

[24] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 24:1–24:15. ACM, 2020.

[25] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In Tarek F. Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems, 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings*, volume 5923 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2009.

[26] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 440–456. ACM, 2021.

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

[28] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In Gregory R. Andrews, editor, *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, pages 202–210. ACM, 1989.

[29] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[30] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[31] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December*

13-16, 2016, Madrid, Spain, volume 70 of *LIPIcs*, pages 25:1–25:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[32] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In Paul Barham and Timothy Roscoe, editors, *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*. USENIX Association, 2010.

[33] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, DSN '11, page 245–256, USA, 2011. IEEE Computer Society.

[34] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, August 2008.

[35] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[36] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[37] Leslie Lamport. Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[38] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[39] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[40] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 467–483. USENIX Association, 2016.

[41] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[42] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 272–281. IEEE Computer Society, 1997.

[43] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[44] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[45] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, USA, 2013. USENIX Association.

[46] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[47] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke, editors, *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*, pages 22:1–22:13. ACM, 2014.

[48] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 479–494, USA, 2014. USENIX Association.

[49] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI*

16), pages 517–532, Savannah, GA, November 2016. USENIX Association.

[50] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 583–598. USENIX Association, 2020.

[51] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 583–598. USENIX Association, 2020.

[52] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013.

[53] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[54] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 472–487. ACM, 2021.

[55] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 156–167. IEEE Computer Society, 2016.

[56] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 43–57. USENIX Association, 2015.

[57] Tuanir França Rezende and Pierre Sutra. Leaderless state-machine replication: Specification, properties, limits. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 24:1–24:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[58] High Scalability. Latency is everywhere and it costs you sales - how to crush it http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, 2009, accessed 2021.

[59] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[60] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[61] Sarah Tollman, Seo Jin Park, and John K. Ousterhout. Epaxos revisited. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 613–632. USENIX Association, 2021.

[62] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[63] Lewis Tseng, Neo Zhou, Cole Dumas, Tigran Bantikyan, and Roberto Palmieri. Distributed multi-writer multi-reader atomic register with optimistically fast read and write. In Kunal Agrawal and Julian Shun, editors, *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, pages 479–488. ACM, 2023.

[64] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 157–180. USENIX Association, 2020.

[65] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), feb 2015.

[66] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), February 2015.

[67] Robbert van Renesse, Fred B. Schneider, and Johannes Gehrke. Nerio: Leader election and edict ordering. *CoRR*, abs/1109.5111, 2011.

[68] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. Craft: An erasure-coding-supported version of raft for reducing storage cost and network cost. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 297–308. USENIX Association, 2020.

[69] Michael J. Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, 2021.

[70] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[71] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 263–278, New York, NY, USA, 2015. Association for Computing Machinery.

[72] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association.

[73] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. Paxosstore: High-availability storage made practical in wechat. *Proc. VLDB Endow.*, 10(12):1730–1741, August 2017.

A Generalization of Pineapple

A.1 Handling Multi-key General Operations in One-shot Transactions

To handle multi-key operations, we need to modify Algorithm 2 in the following way:

- In the Get phase, send messages to read the value for each key in K_{input}.
- For each key in K_{input}, identify the value with largest pstamp from responses received from a quorum.
- Apply $f(\cdot)$ on the values identified in the previous step and generate output value for each key in K_{output}.
- All these output values will have the same pstamp, as the one generated in Line 6 in Algorithm 2.
- In the Put phase, contact a quorum to update the values.

The reason that this simple generalization works is because of the simplicity of Multi-Paxos. The existence of a leader simplifies the design. This is because the leader determines the order of the multi-key operation. Then later in the proof of linearizability, all the operations seen by the leader can be ordered beforehand.

On the other hand, in the case of Gryff, nodes make joint decision by using a dependency graph. It is not clear how they can jointly come to conclusion on the set of values that are the most recent ones for K_{input} and can be used by the one-shot transaction.

A.2 Handling Leader Change

To handle leader change or failures, we rely on the full version of Multi-Paxos to identify the leader. Moreover, we need to modify reads so that they do not return inconsistent values. Concretely, we need to make the following modifications:

- Include the ballot number and leader ID in the pstamp.
- Upon learning a larger ballot, the (potentially old) leader N_l knows that there exists another proposer that aims to be the leader. Hence, N_l needs to run the phase 1 of Paxos [36, 39] to elect a new leader.
- The comparison of pstamp needs to include the ballot number. In other words, any message from the old leader should be discarded (or reply with an NACK).
- Reads need to take extra caution, as if we adopt the same ABD reads, then read could return inconsistent results. This is because during leader change, a one-shot transaction from the old leader might get overwritten by the new leader.

To avoid this scenario, the reads need to back off and retry upon learning the existence of a new leader or a new proposer that is competing for becoming a leader. Reads can observe this from the ballot field in the received messages.

With this modification, a read is completed only if there exists a quorum that shares the same leader with the read and the same value-pstamp pair with the read.

Note that during the leader change, the one-shot transaction (from the old leader) might not be completed yet, as the Put phase has not get a confirmation from a quorum. However, it does not affect correctness, this is because as new leader completes phase 1 of Paxos, it will either learn and adopt the uncompleted one-shot transaction as its command for this slot, or it proposes a new one-shot transaction command. In the latter case, it means that there is no completed read observe the previously uncompleted one-shot transaction.

B Correctness Proof

Here, we prove that the algorithms presented in Algorithms 1 and 2 are correct. To prove linearizability, we need to show that under any execution of the algorithm, we can find a total ordering S which satisfies both the real-time constraint order and the semantic of the operations (i.e., the ordering is legal). Moreover, it satisfies liveness with partial synchrony.

B.1 Intuition

We use the pstamps to order operations. We start with the non-stable ordering of writes. After we place them in the order, we need to find the position for each RMW. Lastly, we find the place for each read.

To prove that the sequence we find following pstamps is correct, we need to show that pstamps correctly reflect real-time ordering and the semantics. This roughly follows the design of quorum intersection (of the Get phase), and the choice of using a single leader (in the Multi-Paxos component).

B.2 Proof

Given an execution of operations H, where operations may interleave with each other (as there are concurrent operations), we need to identify an ordering (or a sequence) S in which operations do not overlap with each other, i.e., the ordering that could occur on a single machine. Following the prior literature [8, 41, 42], we only need to consider completed operations. In the following proof, we assume that all operations are on a single key, as it is straightforward to extend the proof to multi-key operations.

Now we describe the following steps to find the position for each operation in S:

- Start with an empty sequence S.
- Order the writes using pstamps.
- For each RMW r that has a pstamp $(tag_r, slot_r)$, place them one by one, following the order of the slot index. There are only three possible cases:
  - tag_r is $(0,0)$ and $slot_r = 1$, meaning that this RMW is the first non-read operation: RMW r is placed first in S (before all writes);
  - There is another RMW m such that $tag_m = tag_r$ and $slot_m = slot_r - 1$: RMW r is placed right after m in S; and
  - Otherwise: in this case, there must exists a write w such that $tag_w = tag_r$: RMW r is placed right after w in S.

  Note that because of the guarantee of Multi-Paxos, there cannot exists two RMWs that have the same slot index but different pstamp.
- For each read o, since we consider only the completed operation, it must return a null value (the default value), a value either from a write, or a value from a RMW.
  - For a null value, o is placed first in S (before any writes or RMWs);
  - For a value from write w, o is placed right after w in S;
  - For a value from RMW r, o is placed right after r in S.

These steps place all the (completed) operations in S. Next we need to show that S is the total ordering that we are looking for.

Lemma 1. *S satisfies real-time ordering.*

Proof. This follows from the construction of pstamps. If an operation o_1 occurs before operation o_2, then o_2 must have the same pstamp (this only happens when o_2 is a read) or a larger pstamp. The reason is that any completed operation is observed by a quorum (due to the Put phase); hence, when o_2 starts, the Get phase for all three operations allows o_2 to observe o_1. Then the pstamp needs to be increased if o_2 is a write or a RMW. □

Lemma 2. *S satisfies the semantics of the interface defined in Section 2.1. That is, S is legal.*

Proof. First, we need to show that any read d returns the most recent value by a write w or a RMW r in S. This follows from how we find the position for d. In other words, because of how we place d right after the value it returns, there is no non-read operations between w and d or between w and r.

Note that between w and d and between r and d, there could be other reads, but these reads do not matter, since they do not change values.

Second, we need to show that RMWs are consistent. That is, the value after applying RMW is correctly observed by future reads. The proof is by contradiction. Assume that there are four operations: two writes w_1, w_2, one RMW r, and one read d such that

- w_1 is before w_2 in S;
- r is right after w_2 in S;[5]
- d is right after r in S;
- rd sees the effect of r does a RMW on w_1 – whereas a legal execution should have r does a RMW on w_2. This is because w_2 is the most recent write for RMW r, with respect to S.

The reason that we place r right after w_2 is that they have the same tag in associated pstamps. From the Get phase of Pineapple, this means that after this phase, the largest pstamp observed by the leader is from w_2. As a result, the v^{new} is the computation after applying $f(\cdot)$ on the value of w_2, not w_1. This implies that the read d cannot observe the effect of r does a RMW on w_1, a contradiction. This proves the lemma. □

The two lemmas above prove that Pineapple satisfies linearizability.

Finally, we prove liveness.

Lemma 3. *Each operation completes in Pineapple when there is a stable leader.*

Proof. When $n = 2f + 1$ and there is a stable leader, all the phases are non-blocking, as each node and the leader is always able to obtain a response from a quorum. Therefore, all the operation completes under this case. □

Since under partial synchrony, there is always a stable leader in Multi-Paxos. This together with the lemma above implies that Pineapple satisfies liveness.

C Throughput in WAN

Figure 10 shows the maximum throughput achieved by all the systems, except Gryff, with 80 to 130 close-loop clients that batch operations. The clients batch 5000 operations, send to a node, and wait for the completion of all the operations, before proceeding to the next batch. We choose this design to properly load the systems without the machines being throttled by the communication bottleneck. This is also how [46, 61] test the throughput of EPaxos in WAN. We did not evaluate Gryff, because the implementation of Gryff from [10] does not support client batching. As reported in [10], Gryff's throughput is similar to EPaxos.

We use the same workloads that we used in Section 4.4.2 but with different numbers of clients. The exact workloads are specified in the caption of Figure 10. We make the following observations:

- In read-heavy workloads, "Multi-Paxos with lease" has best performance. This is because with lease, the leader does not need to process as many messages.
- In a read-heavy workload with 25% conflicts, PQR performs slightly better than Pineapple. This is because with a higher number of concurrent conflicting operations, Pineapple's reads can no longer complete on the fast path. In contrast, PQR's reads complete after 1 RTT, reducing the among of message processing.
- In balanced workloads, Pineapple has 3x to 4x throughput improvement over the next closest competitor (PQR and "Multi-Paxos with lease"). This is because the leader in Paxos systems is saturated by writes and RMWs, whereas writes are evenly distributed in Pineapple.
- In balanced workloads, the throughput of PQR and "Multi-Paxos with lease" are only slightly larger than Multi-Paxos, because the leaders are already saturated and cannot no longer process reads as efficiently as in the read-heavy workloads.
- EPaxos in general has slightly better performance than Multi-Paxos, but has lower throughput than other systems. This is expected because the dependency grows much faster in WAN, as also reported in [10, 61].

[5] Again, there might be some other reads between w_2 and r, but we do not care about reads here.

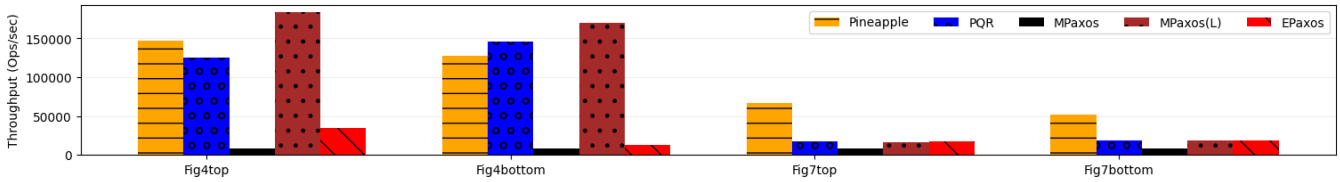


Figure 10: **Maximum attainable throughput in WAN.**
Fig4top: $n = 3$, 94.5% reads, 4.5% writes, 1% RMWs, 2% conflicts
Fig4bottom: $n = 3$, 94.5% reads, 4.5% writes, 1% RMWs, 25% conflicts
Fig7top: $n = 5$, 49.5% reads, 49.5% writes, 1% RMWs, 2% conflicts
Fig7top: $n = 5$, 49.5% reads, 49.5% writes, 1% RMWs, 25% conflicts

Ladder: A Convergence-based Structured DAG Blockchain for High Throughput and Low Latency

Dengcheng Hu[1], Jianrong Wang[1], Xiulong Liu[1], Hao Xu[1], Xujing Wu[2],
Muhammad Shahzad[3], Guyue Liu[4], Keqiu Li[1]
[1]*Tianjin University* [2]*Jd.Com, Inc* [3]*North Carolina State University* [4]*Peking University*

Abstract

Recent literature proposes the use of Directed Acyclic Graphs (DAG) to enhance blockchain performance. However, current block-DAG designs face three important limitations when fully utilizing parallel block processing: high computational overhead due to costly block sorting, complex transaction confirmation process, and vulnerability to balance attacks when determining the pivot chain. To this end, we propose Ladder, a structured twin-chain DAG blockchain with a convergence mechanism that efficiently optimizes parallel block processing strategy and enhances overall performance and security. In each round, a designated convergence node generates a lower-chain block, sorting the forked blocks from the upper-chain, reducing computational overhead and simplifying transaction confirmation. To counter potential adversarial disruptions, a dynamic committee is selected to generate special blocks when faulty blocks are detected. We implemented and evaluated Ladder in a distributed network environment against several state-of-the-art methods. Our results show that Ladder achieves a 59.6% increase in throughput and a 20.9% reduction in latency.

1 Introduction

Background: Within blockchain networks, nodes process incoming transactions and collectively engage in a consensus protocol, assembling the transactions into a block that is securely linked to the distributed ledger using hash pointers. In traditional chain-based structures, if multiple blocks are generated by different nodes simultaneously, only one is accepted while the others are discarded, resulting in the wastage of the computational resources expended in producing those blocks [21]. In contrast, Directed Acyclic Graph (DAG)-based structures allow those blocks to become part of the blockchain such that all blocks are included when generating the ledger [3, 6, 17–19, 22, 24, 34]. Broadly, DAG-based structures can be classified into two types: **tx-DAG**, where nodes represent transactions, and **block-DAG**, where nodes represent blocks composed of multiple transactions [32]. This paper focuses on block-DAGs as they offer better compatibility with existing protocols and provide stronger scalability than tx-DAGs. Broader structures in block-DAG blockchains, while intuitively improving performance through parallel block generation, may also increase transaction ordering overhead, complicate confirmation processes, and exacerbate specific security risks. *Therefore, this paper explores how to fully leverage the advantages of the DAG by optimizing the parallel block processing strategy.*

Limitations of Prior Art: While several promising block-DAG based blockchain designs have been proposed [18, 19, 24–26], they have the following three important limitations when processing parallel blocks.

1) High Sorting Cost. Nodes need to independently sort blocks to maintain consistency when multiple valid blocks are generated in parallel. In approaches such as Conflux [19] and Inclusive [18], every node is required to recursively sort the newly generated blocks along with their predecessors, which leads to considerable redundant computations. As each node independently determines the order without a coordinated strategy, inconsistencies in block views arise, causing additional computational overhead as nodes attempt to reconcile these discrepancies. This lack of synchronization in the sorting process results in high sorting costs, leading to delays in block confirmation, especially under high concurrency.

2) Complex Confirmation Process. In DAG-based blockchains, the validity of individual transactions depends on the confirmation of their respective blocks, often requiring intricate inter-block validation processes. As multiple forks emerge, nodes need to independently verify and reference forked blocks, resulting in a buildup of cross-references that complicates the confirmation process. For example, in Spectre [24], the block-voting mechanism relies on each node incrementally accumulating support for previous blocks, which can be cumbersome as nodes attempt to reconcile conflicting views on the validity of numerous competing forks. Similarly, Conflux [19] utilizes forward references for faster pivot chain confirmation, but the absence of a coordinated approach to

managing these forks causes transactions in non-standard blocks to gain confirmation weight at a slower rate, further prolonging the finalization process. This lack of an efficient mechanism to centrally handle forks leads to a fragmented confirmation process, where the full potential of DAG's parallelism remains underutilized.

3) Susceptibility to Balance Attacks. DAG blockchains define rules to identify the pivot chain [18, 19, 26], allowing multiple valid blocks generated simultaneously to be added to the ledger without conflict. However, when multiple candidate pivot chains emerge, the lack of coordination leads to independent node decisions, hindering prompt convergence on a single pivot chain. This lack of synchronization allows adversaries to exploit decentralized decision-making to delay ledger updates and maintain parity between competing chains. For instance, while Conflux [19] employs a probabilistic method to address these issues, its independent chain weight evaluation at each node may introduce inconsistencies, which adversaries can exploit for balance attacks, delaying convergence to a single pivot chain. As a result, the absence of a mechanism to manage competing chains may lead to delays, threatening blockchain stability and reliability.

These limitations suggest that incorporating an effective convergence mechanism could better coordinate and optimize parallel block processing.

Proposed Approach: In this paper, we propose Ladder, a block-DAG-based blockchain that utilizes a convergence mechanism to achieve high transaction throughput and low confirmation latency, addressing the three key limitations of existing approaches. Ladder is composed of two chains: an upper-chain and a lower-chain, as shown in Figure 1. The upper-chain consists of blocks containing transactions, while the lower-chain coordinates the convergence of forked blocks in the upper-chain. In each round, nodes generate blocks for the upper-chain using Proof of Work (PoW). When multiple blocks are generated in a round, they form a block set. One block is selected as the standard block, while the rest are designated as forked blocks. The upper-chain continues from the standard block, as shown in Figure 1, but all forked blocks are retained to preserve the computational effort involved in their creation. A convergence node, responsible for each round, records the order of forked blocks in the lower-chain. This allows nodes to reconstruct the ledger efficiently by traversing both chains, ensuring a consistent view of the blockchain.

Technical Challenges: We identified two key challenges in the design of Ladder that required resolution. The first challenge is *how to use the lower-chain to effectively converge PoW-generated upper-chain blocks.* A node may lack sufficient computational power to complete the PoW in a timely manner to generate a lower-chain block. Such delays could impede Ladder's progress. To resolve this, we eliminate the PoW requirement for lower-chain blocks, enabling any node to generate a lower-chain block swiftly. To prevent malicious lower-chain blocks, the node that generated the previous round's standard upper-chain block is selected as the convergence node for the current round. This process ensures that no single node can dominate unless it controls a significant proportion of the computational power, which is highly unlikely.

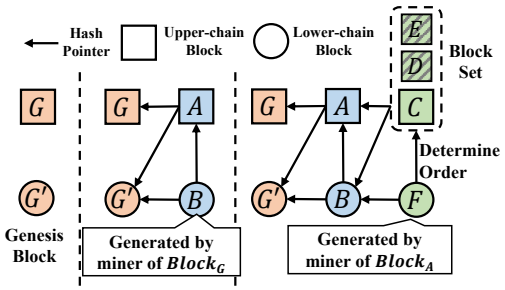


Figure 1: An introduction to the structure of Ladder.

The second challenge is *how to ensure system security by preventing adversaries from becoming convergence nodes.* An adversary could disrupt the blockchain by failing to produce or delaying the lower-chain block, hindering transaction confirmations and reducing throughput. Additionally, the adversary could ignore valid forked blocks or reference nonexistent blocks to disrupt the blockchain structure. To address this, Ladder introduces a super block in the lower-chain to override any faulty lower-chain block. If a fault is detected or a lower-chain block is not generated within a predefined duration, a committee forms and uses a Byzantine Fault Tolerance (BFT)-like consensus mechanism (*i.e.,* HotStuff) to generate a super block. BFT is a class of methods that ensures deterministic consensus within a fixed number of participants [16].

Key Advantages: In this paper, we propose a structured twin-chain DAG with a convergence mechanism for parallel block processing. The advantages of Ladder over the prior art are three-fold: i) compared to the block sorting by all nodes in [19, 24, 25], we introduce a mechanism that designates specific convergence nodes to handle upper-chain block sorting and record this information in lower-chain blocks, reducing sorting overhead and optimizing resource utilization; ii) compared to the complex transaction confirmation mechanism in [18, 19, 25, 26], our approach uses a convergence process to confirm transactions in unresolved upper-chain blocks, eliminating the need for intricate cross-referencing and weight accumulation, and thereby streamlining the transaction confirmation process; iii) compared to [18, 19, 25, 26], Ladder's structure inherently avoids additional computations for determining the pivot chain in the DAG topology, enhancing resistance to balance attacks and providing robust security guarantees against adversarial actions.

Summary of Experimental Results: We implemented Ladder and four state-of-the-art approaches: GHOST [26], Inclusive [18], Phantom [27], and Conflux [19]. Our experiments were conducted on a test bed of 80 servers deployed across a distributed network. Results show that Ladder outperformed prior approaches, achieving a 59.6% increase in throughput

Table 1: Comparison of block-DAG blockchains.

	Sorting by all nodes	Complex Confirmation logic	Balance attack resistance	Performance	
				Trans. txs / sec[1]	Confirmation time
GHOST	Yes	No	No	200	< 60min
Inclusive	Yes	Yes	No	350	< 1min
Spectre	Yes	Yes	-	-	< 1min
Phantom	Yes	Yes	No	40	< 1min
Conflux	Yes	Yes	No	2823	< 1min
OHIE	Yes	Yes	Yes	2513[2]	< 10min
Ladder	No	No	Yes	4506	< 1min

and a 20.9% reduction in transaction confirmation latency compared to the best-performing Conflux.

Limitation of Ladder: A limitation of PoW+BFT architecture is that while BFT consensus ensures security with fewer than $1/3$ malicious nodes, PoW's probabilistic nature does not guarantee this strict security threshold. For adversaries controlling less than 30% computational power, the committee can ensure with high probability that malicious nodes remain below $1/3$, satisfying BFT security requirements. However, the probability of more than $1/3$ of the committee being malicious in at least one of the R rounds is given by $1-(1-P)^R$, where P is the probability of selecting more than $1/3$ malicious nodes in a single BFT round. This probability increases with R, raising the risk of adversaries compromising the system by controlling over $1/3$ of the BFT committee, breaking consensus and disrupting system integrity. We analyze this limitation further in Sec. 4, where we study the effectiveness of increasing the BFT committee size on reducing the probability of malicious nodes exceeding $1/3$ of the committee.

2 Related Work

DAG for BFT Consensus: BFT consensus ensures agreement among distributed nodes with a bounded number of malicious participants and is more suited for permissioned networks [5, 16]. Several works have explored integrating DAG into BFT consensus, mainly in permissioned settings, such as Shoal [28], Bullshark [29], and Narwhal [8]. The fundamental difference between these solutions and Ladder is that Ladder is designed for permissionless blockchains. Ladder relies on PoW to ensure decentralization in trustless settings, while BFT is incorporated only as a lightweight fallback to handle adversarial scenarios.

DAG-based Structures: Prior work on utilizing DAGs in PoW-based blockchains can be broadly divided into two categories based on node representation: tx-based DAGs, where each node represents a single transaction [6, 17, 22], and block-based DAGs, where each node represents an entire block [18, 19, 24–26]. Tx-based DAGs enable decentralized validation but introduce challenges in transaction ordering and validation complexity. The independent nature of transactions increases communication overhead and complicates maintaining a consistent global transaction order. This is challenging in large-scale systems like Byteball [6], Tangle [22], and Nano [17], which face scalability and efficiency trade-offs. For block-based DAGs, a prominent block-based DAG design is GHOST [26], which extends Bitcoin's linear chain into a tree to enhance security. However, GHOST prioritizes security while offering limited improvements in throughput. Inclusive [18] extends GHOST by increasing block inclusion without full ledger ordering, limiting smart contract support. Similarly, Spectre [24] improves performance but sacrifices ledger ordering, restricting functionality. Phantom [25, 27] resolves the ordering issue but introduces a computationally expensive recursive sorting algorithm, hindering performance. Conflux [19] extends GHOST by leveraging a block-DAG structure to achieve high throughput. However, its reliance on probabilistic methods for balance attack defense leaves certain vulnerabilities unresolved.

Hierarchical and Parallel Structure: Meshcash [4] employs a hierarchical DAG, using block references as votes for confirmation. While enhancing security, this design disadvantages nodes with low computational power or poor network connectivity, leading to delayed confirmations and degraded performance. OHIE [37] improves throughput with a parallel chain architecture. However, each parallel chain in OHIE retains a linear blockchain structure, meaning block forks can still result in orphaned blocks. In contrast, Ladder utilizes a block-DAG structure to avoid valid orphaned blocks. Prism [2] adopts a decoupled design, using vote blocks to select a block for ledger ordering. However, vote blocks rely on PoW solely for voting, increasing resource consumption. Moreover, Prism's performance depends on near-ideal network conditions, such as a block propagation delay of less than one second. In contrast, Ladder packages transactions within vote blocks and does not require strict network assumptions, making it more adaptable to practical P2P networks.

Table 1 summarizes several related work, highlighting the limitations addressed by Ladder.

3 Detailed Design of Ladder System

In this section, we provide a detailed description of Ladder. We already provided an overview of Ladder in Sec. 1 under Proposed Approach. Here, we start with a system-level view of Ladder nodes and the system model. Based on this, we present various notations and terms that we use in this paper. Subsequently, we describe how nodes generate Ladder blocks and determine the pivot chain in the event of forking. We then explore various corner cases that may arise and how Ladder handles them. Finally, we end this section with a discussion about various performance-related aspects of Ladder.

3.1 System Model and Node Operations

We illustrate the system-level process of node participation in Ladder, as shown in Figure 2. Ladder operates under

[1] As measured in our experimental environment.

[2] We set the number of parallel chains to 40 in our experiments.

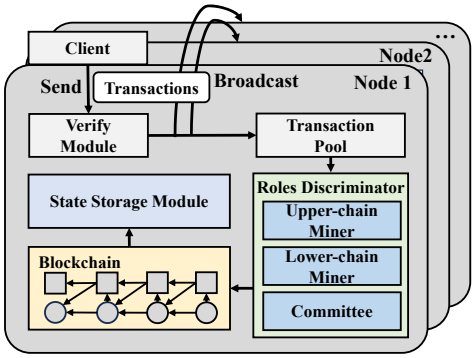


Figure 2: A system view of Ladder nodes.

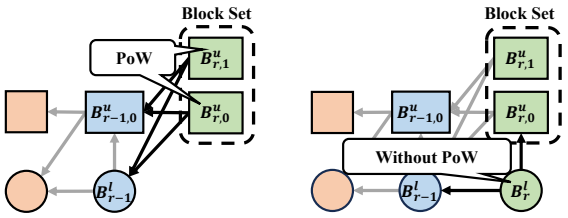
a δ-synchronous network assumption (following [21, 37]), where messages between honest nodes are delivered within a bounded time δ under normal conditions. The system tolerates temporary disruptions, such as message delays or node crashes, as long as the network eventually stabilizes. We assume that adversaries control at most p percent of the total computational power, where $p < 30\%$, ensuring the security of both upper-chain PoW and lower-chain BFT consensus with high probability. In Ladder, nodes participate as upper-chain miners, lower-chain miners, and/or BFT committee members, and are incentivized for their contributions to block generation. Upper-chain miners use PoW to generate blocks. Lower-chain miners handle transaction ordering and produce lower-chain blocks. In the event of waiting duration timeouts or faults, the BFT committee generates super blocks to ensure consensus continuity and maintain system integrity.

3.2 Model, Definitions, and Notations

We model Ladder using a DAG $\mathcal{G} = (\mathcal{B}, \mathcal{V}, \mathcal{E})$. Next, we describe in detail what $\mathcal{B}$, $\mathcal{V}$, and $\mathcal{E}$ represent.

Blocks $\mathcal{B}$: We represent the set of all valid blocks with $\mathcal{B}$. The set $\mathcal{B}$ consists of three types of blocks: B^u represents all valid blocks in the upper-chain. B^l represents all valid blocks in the lower-chain, and B^s represents all super blocks. Before we define these three types of blocks, we first introduce the concept of a *round*.

Round: A round in Ladder defines the position of a block within the upper- and lower-chains. Each round begins with the generation of the lower-chain block from the previous round and ends with the generation of the lower-chain block in the current round. During each round, one new block is added to the lower-chain and at least one block is added to the upper-chain. The upper-chain and lower-chain blocks generated in round r are denoted as B^u_r and B^l_r, respectively. Ladder does not require all honest nodes to have the same round number, as network delays may cause lower-chain blocks to be received at different times. The round number is simply used to track block order and maintain the local ledger. Even if nodes are temporarily in different rounds, the system ensures eventual consistency through block propagation and validation, within the bounds of the δ-synchronous network model.



(a) Upper-chain (b) Lower-chain

Figure 3: Normal operation of Ladder.

Upper-chain Blocks B^u: The upper-chain blocks in Ladder are generated by nodes using the PoW consensus mechanism. Despite being energy-intensive, PoW remains a cornerstone of blockchain security, and improving its performance and scalability is crucial for maintaining its relevance in decentralized and trustless environments. Upper-chain blocks are further categorized into standard and forked upper-chain blocks. Standard upper-chain blocks are those that become part of the upper-chain. We represent a standard upper-chain block in round r with $B^u_{r,0}$. $B^u_{r,0}$ contains two hash pointers. One pointer links to the standard upper-chain block $B^u_{r-1,0}$ of the preceding round, and the other links to the lower-chain block B^l_{r-1} of the preceding round. This process is shown in Figure 3(a). Forked upper-chain blocks are legitimate blocks generated by some nodes in the same round in which the standard upper-chain block is generated. We represent a forked upper-chain block in round r with $B^u_{r,i}$, where $i = 1, 2, 3, \ldots$ depending on how many forked upper-chain blocks were created during round r. A forked upper-chain block $B^u_{r,i}$ also contains two hash pointers. One pointer links to the standard upper-chain block $B^u_{r-1,0}$ of the preceding round, and the other to the lower-chain block B^l_{r-1} of the preceding round. The primary difference is that the upper-chain blocks in the next round will point to the standard upper-chain block of the current round, but not to forked blocks. In other words, the chain continues only from a standard upper-chain block, not from any forked upper-chain blocks. This process is shown in Figure 3(b).

Lower-chain Blocks B^l: Lower-chain block B^l_r in round r contains information about the standard upper-chain block, any forked blocks, and their respective sequence numbers. Lower-chain block B^l_r is generated in round r by the node that generated the standard upper-chain block $B^u_{r-1,0}$ in the previous round. The generation of lower-chain blocks is computationally inexpensive and does not require any PoW mechanisms. Lower-chain block B^l_r contains two hash pointers. One pointer links to the standard upper-chain block $B^u_{r,0}$ of the current round, and the other to the lower-chain block B^l_{r-1} of the preceding round. It also contains reference links to all forked upper-chain blocks in round r, facilitating the convergence of forks in the upper-chain.

Super Blocks B^s: A super block is added to the lower-chain when the referenced upper-chain blocks contain invalid trans-

actions from the current or previous rounds, or when a lower-chain block is not received within a predefined time duration despite the presence of valid upper-chain blocks. In Ladder, super blocks are generated through the HotStuff consensus mechanism [36]. Sec. 3.3 will provide further details on fault conditions and the generation process of the three types of blocks (upper-chain, lower-chain, and super).

Nodes $\mathcal{V}$: We represent the set of all nodes with $\mathcal{V}$ and any arbitrary node in $\mathcal{V}$ with V. The ratio of a single node V's computing power relative to the total computing power of all nodes is represented by $P(V)$, such that $\sum_{\forall V \in \mathcal{V}} P(V) = 1$.

Edges $\mathcal{E}$: We represent the set of link relationships among all blocks with $\mathcal{E}$. For example, if block $B^u_{r,0}$ points to block B^l_{r-1}, then $(B^u_{r,0}, B^l_{r-1}) \in \mathcal{E}$ represents a directed link from $B^u_{r,0}$ to B^l_{r-1}. More generally, expressing a link as (a,b), if $(a,b) \in \mathcal{E}$, a property of Ladder is that $a \neq b$ and $(b,a) \notin \mathcal{E}$.

Serial Numbers and Ledger Generation: To enable a node to generate the ledger from the blocks in the upper- and lower-chains of Ladder, we assign a serial number to each block. In any given round $r \geq 0$, the lower-chain block has the serial number $2r+1$ and the standard upper-chain block has the serial number $2r$. If round r has any forked upper-chain blocks, then the serial numbers of forked blocks are denoted by adding a subscript i to the value $2r$, where $i = 1, 2, 3, \ldots$. We represent the serial number of any block B with $O(B)$. For example, if B^u is a standard upper-chain block in round 3, then $O(B^u) = 6$. If B^u is the second forked upper-chain block in round 3, then $O(B^u) = 6_2$. Similarly, if B^l is a lower-chain block in round 3, then $O(B^l) = 7$. Suppose rounds 0, 1, 4, and 5 do not have any forked blocks but only standard upper-chain blocks. Suppose round 2 has one forked block, and round 3 has three forked blocks. To generate the ledger, the upper-chain blocks will be arranged in the following sequence: $0, 2, 4, 4_1, 6, 6_1, 6_2, 6_3, 8, 10, \ldots$. Transactions are inherently sequential, so while parallel block processing enables higher throughput, ordering the blocks ensures all transactions follow the correct sequence. This guarantees a totally ordered ledger, allowing Ladder to support not only simple transfer transactions but also complex smart contract executions.

3.3 Block Generation

Next, we describe the generation process for the different block types mentioned in Sec. 3.2.

Upper-chain Blocks: In each round r, each node attempts to find a nonce that meets the difficulty target required for block generation. Once a node finds a valid nonce, it broadcasts the resulting block to all nodes in the network. Round r concludes when the lower-chain block B^l_r is generated. Before the round concludes, it is common for multiple upper-chain blocks to be generated by different nodes. In such cases, one of these blocks is designated as the standard upper-chain block, while the others are classified as forked upper-chain blocks. If applicable, information about the standard and forked blocks in round r is stored in the corresponding lower-chain block B^l_r.

Lower-chain Blocks: Suppose we are currently in round r. This means that the standard upper-chain block and any forked upper-chain blocks from round $r-1$ have already been determined, with this information stored in the lower-chain block B^l_{r-1} of the previous round. In the current round r, the node whose block was designated as the standard upper-chain block in round $r-1$ is responsible for generating the lower-chain block. By generating the lower-chain block, Ladder achieves fast convergence of upper-chain blocks. Additionally, only the miner of the standard upper-chain block from round $r-1$ is required to generate the lower-chain block. The node collects upper-chain blocks for a predefined duration, checks for faults, discards faulty blocks, and selects the standard upper-chain block $B^u_{r,0}$ for round r according to the *Hardest Chain Principle*. This block must have hash pointers to the standard upper-chain block $B^u_{r-1,0}$ and lower-chain block B^l_{r-1}. It marks the remaining upper-chain blocks as forked and assigns them serial numbers $(2r)_1$, $(2r)_2$, and so on. It inserts this information into the lower-chain block B^l_r for round r, along with the hash pointers for $B^u_{r,0}$ and B^l_{r-1}. By producing lower-chain blocks, the convergence node exclusively records forked blocks and ensures a globally consistent transaction order. Unlike other solutions, Ladder optimizes block ordering while preserving decentralization, thereby enhancing both efficiency and finality. An upper-chain block in round r may reference $B^u_{r-1,0}$ and B^l_{r-1} but fail to reach the lower-chain block generator in time. Instead, it may arrive at the generator for round $r+e$ ($e \geq 1$) before its lower-chain block is generated. Consequently, this block will be sorted in round $r+e$, and it will be classified as a forked upper-chain block rather than a standard one.

Super Block: Suppose node V is in round r. It detects a fault if either of the following two situations occurs.

1. Node V receives a lower-chain block for round r, but the referenced upper-chain blocks contain transactions that conflict with those in the current or previous rounds.

2. Node V does not receive a lower-chain block for round r within a predefined time duration, despite receiving several valid upper-chain blocks.

To resolve such faults, Ladder employs the BFT consensus mechanism. Since lower-chain blocks must be unique, BFT consensus guarantees agreement on a single canonical lower-chain block among all nodes. As a deterministic consensus mechanism, BFT consensus has been widely used in various fields [1, 5, 7, 15, 30, 31, 35, 36]. The HotStuff consensus [36] is a BFT-based consensus executed by a committee comprising the nodes that generated the most recent n standard upper-chain blocks. HotStuff is selected for its simplicity, efficiency, and linear message complexity. Compared to other BFT mechanisms in some PoS proposals, HotStuff offers

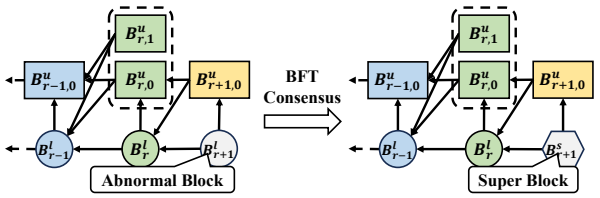


Figure 4: Example of the use of super block in Ladder.

superior scalability, making it suitable for super block generation. Upon committee formation, the leader is selected via VRF [12]. During Ladder's operation, if a node detects an issue, it notifies the committee members. The committee verifies the fault, discards conflicting upper-chain blocks, and reselects a standard upper-chain block based on the *Hardest Chain Principle*. The committee leader then generates a super block, replacing the lower-chain block, as shown by the hexagonal block B^s_{r+1} in Figure 4. This super block contains hash pointers to the previous round's lower-chain block and the current round's upper-chain block, along with references to any forked upper-chain blocks. Subsequent upper- and lower-chain blocks are generated through the standard process described earlier. Notably, a dynamically sized committee may cause inconsistencies in node perceptions, so we adopt a fixed size to ensure information consistency.

At the end of each round, nodes are rewarded based on their roles in that round, incentivizing sustained participation.

3.4 Hardest Chain Principle

Unlike Bitcoin, to enable faster transaction insertion on the ledger, a blockchain may produce blocks at a high rate. However, this can cause honest nodes to link blocks to different sub-chains based on their local views, fragmenting their computing power. An adversary can take advantage of this. It can concentrate all its computing power on a target sub-chain to attack it. To address this issue, GHOST [26] proposes the *heaviest chain principle*, where the pivot chain is determined by the number of sub-blocks linked to a block. However, GHOST is vulnerable to liveness attacks, where an adversary delays transaction confirmations by creating multiple forks. To mitigate this, Conflux [19] proposed GHAST, which introduces dynamic block weight (*e.g.,* special blocks with high weights) to mitigate liveness attacks. While GHAST improves resilience, it may not fully address balance attacks.

Ladder proposes the *Hardest Chain Principle*, which extends the *heaviest chain principle* with dynamic weight adjustment to support efficient parallel block processing. Unlike the *heaviest chain principle*, which relies on the number of sub-blocks, the *Hardest Chain Principle* selects the pivot chain based on the cumulative difficulty of blocks in a sub-tree. The difficulty of block B_i, denoted as $\mathcal{D}(B_i)$, is calculated by the number of leading zeros in its hash and the hashes of all blocks in its sub-tree, as defined by the following equation:

$$\mathcal{D}(B_i) := \sqrt{\mathcal{Z}(B_i)} + \sum_{B_j \in S_i} \mathcal{D}(B_j) \quad (1)$$

Here, $\mathcal{Z}(B_i)$ represents the number of leading zeros in the hash of block B_i, and S_i denotes the set of all legal blocks with hash pointers pointing to block B_i. Putting $\mathcal{Z}(B_i)$ under square root achieves two properties. First, in the case of a fork, the side with more legal blocks following the forked block has higher difficulty and is selected. This makes it easier to select a side. Second, if the number of legal successor blocks on both sides is equal, as long as the number of leading zeros is different in all the sub-blocks, the difficulty of the two sides will be unequal, enabling the selection of one of the sides. This significantly reduces the likelihood of a balance attack. If both sub-chains have m legal blocks after the fork position, it can be proven that the probability of the blocks having equal difficulty is bound by $P \leq \frac{1}{12}(1-\frac{1}{4^k})$, where k is the maximum number of leading zeros in the hash value.

We now prove that the side with more legal blocks following the forked block has higher difficulty. Consider a scenario where a fork leads to the formation of two sub-chains. Let sub-chain 1 have m blocks with a leading zeros each, and sub-chain 2 have $m-t$ blocks with b leading zeros, where $a < b$. In other words, each block in sub-chain 2 has more leading zeros than those in sub-chain 1. The generation probability of a block with x leading zeros is given by $P = \frac{1}{2^{x+1}}$. Thus, the generation probability of all m blocks in sub-chain 1 is $\prod_{i=1}^{m} P_i = \prod_{i=1}^{m} \frac{1}{2^{a+1}} = \frac{1}{2^{m(a+1)}}$. Similarly, the generation probability of sub-chain 2 is $\frac{1}{2^{(m-t)(b+1)}}$. If the generation probabilities of the two sub-chains are equal (*i.e.*, the computing power is equally divided between the two chains), then $b = \frac{ma+t}{m-t}$. To show that sub-chain 1 has greater weight than sub-chain 2, we must prove $\sum_{i=1}^{m} \sqrt{a} > \sum_{i=1}^{m-t} \sqrt{b}$. Alternatively, we need to demonstrate $m\sqrt{a} > (m-t)\sqrt{\frac{ma+t}{m-t}}$, which can be simplified to $m^2 a > (m-t)(ma+t)$, implying $ma > m-t$. The conclusion holds given that $m > 0$, $a > 1$, and $t > 1$. This proves that the side with a greater number of valid blocks following a given block must have a higher difficulty.

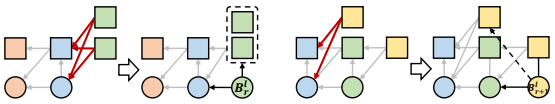
The *Hardest Chain Principle* can mitigate liveness attacks and balance attacks by selecting the chain with the highest cumulative difficulty, where a block's difficulty is determined by the number of leading zeros in its hash and the hashes of all blocks in its subtree. This approach ensures that an attacker must generate blocks with higher difficulty to influence the pivot chain, requiring substantial computational power and making such an attack economically infeasible. Furthermore, the *Hardest Chain Principle* can serve as an alternative to GHOST [26] and GHAST [20] in their respective systems.

3.5 Exception Handling in Block Generation

Exceptions in Upper-Chain Blocks: We categorize exceptions in the upper-chain into the following cases:

Invalid block: An invalid block is a block that contains duplicate or conflicting transactions either with itself or with previous upper-chain blocks. Its creator receives no reward.

Forks: Two types of forks may occur in the upper-chain:



(a) Tip fork in upper-chain (b) Chain fork in upper-chain

Figure 5: Exception cases in upper-chain.

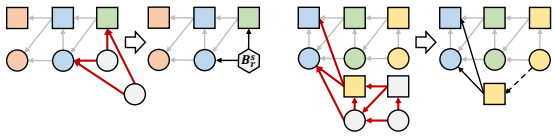
tip forks and chain forks. A tip fork occurs when multiple valid upper-chain blocks are generated in round r. In this case, the convergence node selects one block as the standard block and designates the others as forked blocks for round r. The convergence node then generates block B_r^l to include this information, as shown in Figure 5(a). A chain fork occurs when a valid upper-chain block is generated in round r but does not arrive in time at the node responsible for generating the lower-chain block for round r, instead reaching the node responsible for round $r+1$. In this case, the lower-chain block B_{r+1}^l contains a pointer to this upper-chain block as a forked block, as shown in Figure 5(b). Such an upper-chain block is never used as a standard upper-chain block.

Exceptions in Lower-Chain Blocks: We also categorize exceptions in the lower-chain into the following cases:

Block timeout: Block timeout occurs when a node belonging to the BFT committee does not receive a lower-chain block within a specified time. In such cases, the committee initiates HotStuff consensus and generates a super block.

Invalid block: Similar to invalid upper-chain blocks, a lower-chain block is considered invalid if it contains faulty information or is generated by an unauthorized node. Invalid blocks are ignored. However, if an invalid block is generated by the convergence node of the current round, the BFT consensus is triggered to generate a super block.

Fork: If a lower-chain block is not generated by the convergence node in any given round, it is simply ignored by the nodes. Lower-chain forks can also be classified into tip forks and chain forks. A tip fork occurs in the lower-chain when the node responsible for generating the lower-chain block malfunctions and generates more than one lower-chain block in a given round, as shown in Figure 6(a). This triggers the HotStuff consensus to generate a super block B_r^s, where the node that malfunctioned is not allowed to participate. A chain fork occurs in the lower-chain when different nodes accept different upper-chain blocks as standard blocks in a given round, and consequently accept lower-chain blocks from different nodes in the next round. This results in the propagation of two separate Ladders in the network, as shown in Figure 6(b). When eventually any given node detects the existence of multiple Ladders, it employs the *Hardest Chain Principle*, as discussed in Sec. 3.4, to select one Ladder and discard the rest. Any Ladder may contain super blocks in place of lower-chain blocks. In such cases, when applying the *Hardest Chain Principle*, the difficulty of the super block is quantified by the number of nodes participating in HotStuff.



(a) Tip fork in lower-chain (b) Chain fork in lower-chain

Figure 6: Exception cases in lower-chain.

Exceptions in Super Blocks: A super block is considered invalid if it contains conflicting information or fails committee validation due to adversarial behavior. Since the committee uses BFT to generate super blocks, such issues can be mitigated through BFT's internal mechanisms as long as the proportion of adversarial nodes remains below $1/3$. When an invalid super block is detected, the committee initiates a view change [36] and elects a new leader to restart consensus, ensuring super block reliability.

3.6 System Analysis

This section discusses Ladder's performance in throughput, latency, and scalability, as well as the bootstrapping process.

Throughput: Effective utilization of a node's computing power is key to improving system throughput. Nodes use PoW to generate blocks. Following the *Hardest Chain Principle*, they can become miners of standard upper-chain blocks. This grants them the role of convergence nodes, giving them the right to generate lower-chain blocks. Ladder improves upon GHOST and GHAST for pivot chain selection while maintaining comparable computational overhead, as the required information can be derived from block metadata. Convergence nodes determine block order, reducing network-wide coordination overhead while preserving throughput. The lower-chain allows nodes to package more transactions without PoW, increase rewards, and eliminate computational bottlenecks, improving efficiency.

Latency: In Ladder, the analysis of Bitcoin's longest chain principle and confirmation rule is still applicable for the *Hardest Chain Principle*. Consider a scenario where the adversary fully controls the computing power of malicious nodes. Let p and q represent the proportion of total computing power of honest and malicious nodes, respectively, where $p+q=1$. After a transaction is included in a block, a waiting period of z upper-chain blocks is necessary to prevent selfish mining and anomalies in block processing caused by the adversary. Recall that the explanation of lower-chain forking cases was presented in Sec. 3.5. The adversary's selfish mining behavior follows a Poisson distribution, where λ represents the expected number of upper-chain blocks secretly generated by the adversary during the waiting period of z upper-chain blocks. Since the number of generated upper-chain blocks is proportional to computing power, we have $\frac{\lambda}{z} = \frac{q}{p}$, or equivalently, $\lambda = \frac{zq}{p}$. Thus, the probability of the adversary producing k upper-chain blocks during the waiting period of z

upper-chain blocks can be calculated using the formula $\frac{\lambda^k e^{-\lambda}}{k!}$. In the case where the adversary generates k (where $k \leq z$) upper-chain blocks, the probability of catching up to the z revealed upper-chain blocks is $(\frac{q}{p})^{(z-k)}$. If $k > z$, the probability of catching up is 1. Consequently, the probability of being caught up by the adversary after waiting for z upper-chain blocks is given by $1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!}(1 - (\frac{q}{p})^{(z-k)})$.

Scalability: Ladder demonstrates excellent scalability, allowing nodes to join or leave the system freely without affecting the PoW of other nodes. Furthermore, as the number of nodes increases, the system's total computing power grows, thereby optimizing parallel block processing, improving throughput and reducing confirmation delays. In contrast, solutions like OHIE [37] extend the ledger to multiple parallel chains, which improves throughput. However, due to their chain-like structure, OHIE's parallel chains still face the issue of orphan blocks. A promising future direction is to combine these methods, utilizing DAG-based parallel chains with local consensus and lightweight global synchronization, which could enhance throughput while ensuring consistency.

Bootstrapping: Ladder requires an initial BFT committee to operate. However, relying on a global random selection process could render the system susceptible to Sybil attacks [9]. To address this, the system employs Nakamoto consensus during the bootstrap phase, then transitions to the Ladder protocol after k blocks have been generated [14].

4 Parameter Settings

BFT Committee Size: In Ladder, the presence of a BFT committee may introduce two security risks: selfish mining and adversaries exceeding $1/3$ of the committee. Therefore, the BFT committee size is critical to system security, as it determines which nodes are responsible for generating the super block.

To mitigate selfish mining, a larger committee size is advantageous, as it increases the difficulty of generating super blocks, making it harder for adversaries to withhold and release blocks to manipulate the chain order [10]. We show in Appendix A that the probability of a selfish mining attack occurring is bounded by $P = \frac{1}{2^{2\sqrt{k}}}$.

For adversaries exceeding $1/3$ of the committee, as noted in the Limitation of Ladder section (Sec. 1), Ladder requires malicious nodes within the committee remain below $1/3$. However, due to the probabilistic nature of PoW, this guarantee cannot be strictly ensured. Appendix B shows that this probability decreases exponentially as the committee size increases. Furthermore, over multiple rounds, the probability of at least one successful attack accumulates. In Table 2, we present the per-round probability of adversarial dominance and the number of rounds required for the cumulative probability of at least one successful attack to reach 99%. As the committee size grows, the probability of more than $1/3$ adversaries decreases, and the required number of rounds increases.

Table 2: Impact of Committee Size on Adversarial Risk

Committee Size	120	180	240	300
Probability	4.9×10^{-3}	1.8×10^{-4}	6×10^{-6}	1.96×10^{-7}
Rounds for 99%	9.2×10^{2}	2.6×10^{4}	7.7×10^{5}	2.4×10^{7}

For example, when the committee size is 300, the probability of selfish mining is approximately 4.67×10^{-11}, while the probability of more than $1/3$ adversaries in a single round is about 1.96×10^{-7}. This event requires roughly 2.4×10^{7} rounds to reach a cumulative probability of 99%.

Node Incentive Mechanism: As previously introduced, Ladder employs an incentive model to reward nodes for their participation. The block generation rewards in the upper- and lower-chain are similar and correlated with the number of valid transactions stored in the block. The super block generation reward is evenly distributed among all participating committee members in the BFT consensus algorithm. If a block is not selected as the standard upper-chain block but is selected as a forked upper-chain block, its reward is influenced by the difference in the number of rounds between its creation and its inclusion in the upper-chain. Specifically, this reference is added to a lower-chain block. To calculate the reward accurately, the reward should be multiplied by a block generation reward factor, which is less than 1. Let the upper-chain block B^u be linked to block B. The block generation reward factor is defined as $S = \frac{e^{l-d}}{1+e^{l-d}}$, where $d = R(B^u) - R(B)$ represents the difference in the number of rounds, and l is a manually set parameter that determines the rate of decrease of the block generation reward as the difference in rounds approaches l. For the lower-chain block generation node, the reward received for adding a reference to an upper-chain block should also be multiplied by the block generation reward factor. This incentivizes the lower-chain block generation node to promptly reference upper-chain blocks within lower-chain blocks. Both the block generation reward and the reward for adding a reference to an upper-chain block are stored in the lower-chain block for the respective round.

The incentive for the node to mine a single upper-chain block is calculated as $r_u = Block\ Reward + \sum_{Tx \in B} Txfee * (1 - \alpha)$. Similarly, for the convergence node, the incentive to generate a lower-chain block is determined by $r_l = Block\ Reward + \sum_{B \in round} \sum_{Tx \in B} Txfee * \alpha_i$. Next, we explain what α_i represents. The blocks in each round are generated by different nodes, and each node selects the transactions to include in its block from the transaction pool. Thus, it is possible that one or more transactions get included in multiple blocks generated by different nodes in any given round, resulting in duplicate transactions. α_i represents the proportion of transaction i among all duplicate transactions in a given round. Formally, let x_i represent the number of occurrences of transaction i that appears at least twice among all the blocks generated in a given round. Then $\alpha_i = \frac{x_i - 1}{\sum_{\forall i}(x_i - 1)}$.

To execute a double spending attack with V, the attack

involves generating n blocks during its phase, and the cost for each individual block generation is denoted as c. According to Hoeffding's inequality, take $\varepsilon = 1/2 - q$, $Pr[|A| \geq n/2] \leq \exp(-\frac{(1-2q)^2 n}{2}) = \exp(-\frac{(p-q)^2 n}{2})$. Then, to make the attack gain in the desired sense not to exceed the cost, $Pr[|A| \geq n/2] * V \leq \frac{n}{2}(c - r_d)$ is needed to obtain $V \leq \frac{n(c-r_d)}{2exp(\frac{(p-q)^2 n}{2})}$.

Waiting Time Duration: Following the generation of an upper-chain block in a specific round, the remaining nodes continue mining upper-chain blocks until they receive the lower-chain block from the designated convergence node. If the lower-chain block is not produced within the predefined waiting duration, the committee triggers the BFT consensus. Next we discuss how to set the waiting time duration.

Let W_i^j represent the waiting time for node j to receive a block with sequence number i, and T_i^j represent the timestamp when node j receives a block from node $V(i)$. We use $V(B)$ to represent the node that generated block B. $PT(V(i), j)$ indicates the time it takes for a block to propagate from node $V(i)$ to node j, and $MT(i)$ represents the mining time required by node $V(i)$ to generate a single block. Recall that the upper-chain block in round r is identified by the serial number $2r$, while the lower-chain block is denoted by the serial number $2r+1$. Assuming equal computing power, the mining time is the same for all nodes, while the propagation time varies. The waiting time W_i^j solely accounts for the propagation, excluding mining time, since any nonce can be used in the generation of lower-chain blocks. Thus, $W_i^j = PT(V(i), j)$. A node can estimate the mining and propagation times by analyzing the timestamps of the latest upper- and lower-chain blocks. Formally, this can be expressed as Eq.(2). Here, $MT(B^u_{r-1})$ is the mining time for the upper-chain block in round $r-1$, $PT(V(B^u_{r-1}), j)$ is the propagation time from the upper-chain block's generator to node j, and $PT(V(B^l_{r-1}), j)$ is the propagation time from the lower-chain block's generator to node j.

$$W^j_{2r+1} \approx \underbrace{(T^j_{2(r-1)} - T^j_{2(r-2)+1})}_{MT(B^u_{r-1})+PT(V(B^u_{r-1}),j)} - [\underbrace{(T^j_{2(r-2)} - T^j_{2(r-3)+1})}_{MT(B^u_{r-2})+PT(V(B^u_{r-2}),j)} - \underbrace{(T^j_{2(r-1)+1} - T^j_{2(r-1)})}_{PT(V(B^l_{r-1}),j)}] \quad (2)$$

The node responsible for generating the legal lower-chain block in round $r-1$ is the same as the node that generated the upper-chain block in round $r-2$ (denoted as $V(B^u_{r-2}) = V(B^l_{r-1})$), and the time for each node to generate a single block is equal ($MT(B^u_{r-1}) = MT(B^u_{r-2})$). Consequently, Eq.(2) can be simplified as $W^j_{2r+1} \approx PT(V(B^u_{r-1}), j)$. This means that the waiting time for node j to receive a block in the current round is equivalent to the propagation time it takes for a single block to propagate from the previous round's upper-chain block generator to node j.

The analysis presented above demonstrates that in Ladder, any given node can determine the waiting time for lower-chain blocks by utilizing the historical information it has collected regarding the arrival times of the standard block in round $r-1$ and lower-chain blocks in the next round r.

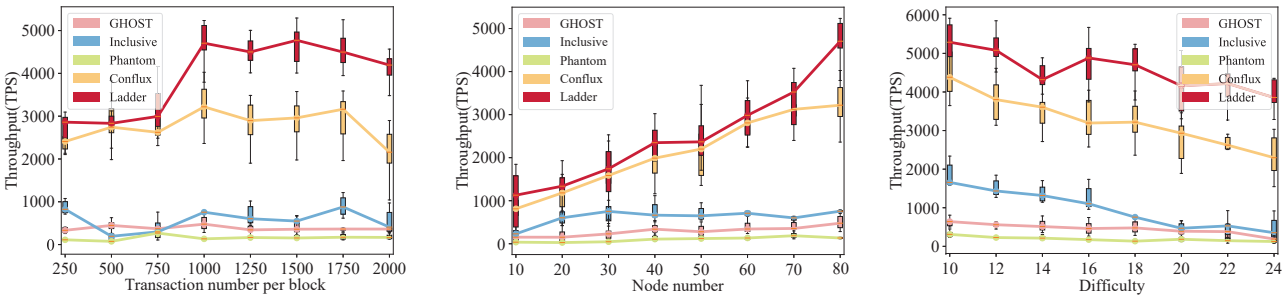
5 Evaluations

In this section, we present results from a comprehensive evaluation of Ladder and compare them with prior approaches.

Setup and Experiments: We implemented a prototype of Ladder and compared it with four recent approaches: GHOST [26], Inclusive [18], Phantom [27], and Conflux [19]. Experiments are conducted on an 80-node LAN testbed, with each machine equipped with an Intel(R) Core(TM) i5-4590 CPU @3.30 GHz and 8 GB of RAM. The inter-node communication latency is set to approximately 80-120 ms to align with real-world network conditions [33]. We use a P2P communication protocol where each node is connected to approximately 10 other nodes. Experiments are run in a controlled, adversary-free environment to observe Ladder's optimal performance without adversarial interference. To align with Conflux, which confirms a block after 6 subsequent blocks, Ladder also confirms a block after 6 subsequent upper-chain blocks. Unlike using a fixed block interval [19, 37], we employ PoW for block generation. In our experiments, the throughput results of previous approaches match their reported values under the following conditions: block difficulty requires 18 leading zeros, and each block contains 1000 randomly generated payment transactions (each approximately 300 bytes), as derived from our interpretation of the protocol specifications.

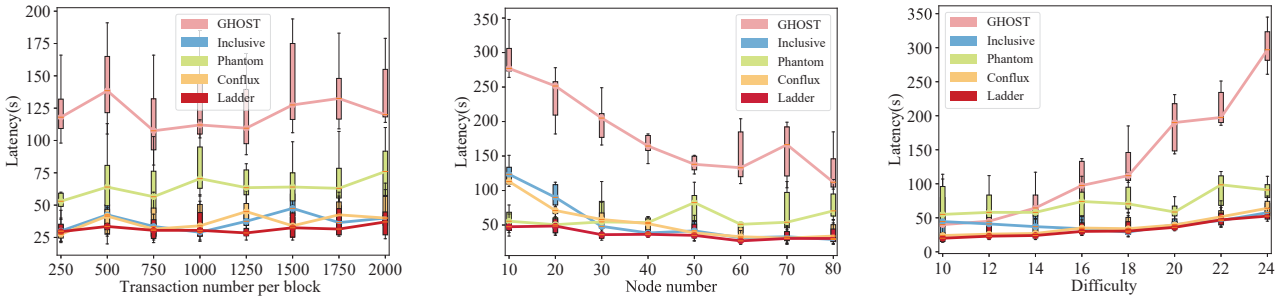
Metrics: We primarily evaluate the end-to-end performance of Ladder and prior approaches. Throughput is measured as the number of confirmed transactions per second (TPS), while confirmation latency is defined as the time from a transaction's submission to its block confirmation. All experiments are repeated 10 times. We conduct experiments over a 10-minute stable operation starting from the genesis block, following Conflux's methodology [19] to ensure a consistent evaluation period. Metrics are recorded every minute, yielding 10 data points per experiment and 100 in total across all runs. This ensures statistically meaningful measurements while accounting for PoW's inherent randomness in block generation intervals. Box plots are used to assess variance, preventing short-term fluctuations from distorting metric trends. To capture metric variations caused by PoW's stochastic nature, we use whiskers, which extend to the most extreme data points within $1.5 \times IQR$ from the first ($Q1$) and third ($Q3$) quartiles, minimizing the impact of outliers. Additionally, we use line plots with points representing the medians of the metrics.

Aggregate Observations: The results show that Ladder outperformed all prior approaches. Among prior approaches, Conflux performed the best, with a median throughput of 2823 TPS and a confirmation latency of 43 seconds. Compared to Conflux, Ladder showed a 59.6% improvement in



(a) Throughput with different block sizes (b) Throughput with different number of nodes (c) Throughput with different block difficulties

Figure 7: Throughput of Ladder and its comparison with prior approaches.



(a) Latency with different block sizes (b) Latency with different number of nodes (c) Latency with different block difficulties

Figure 8: Confirmation latency of Ladder and its comparison with prior approaches.

throughput and a 20.9% reduction in latency under the same settings. Ladder achieved a median throughput of 4506 TPS and a confirmation latency of 34 seconds. Next, we analyze the impact of block size, network size, and difficulty on the performance of Ladder and prior approaches.

Impact of Block Size: Figures 7(a) and 8(a) show the throughput and confirmation latency of various approaches across different block sizes under a difficulty level of 18. Results indicate that Ladder outperformed all prior approaches across the evaluated block sizes. For block size below 1000, Ladder exhibits lower throughput. As the block size increases from 1000 to 1750, throughput remains stable between 4000 and 5300 TPS. In this stage, the increased transaction capacity of larger blocks compensates for the growing propagation delay, maintaining stable throughput. However, at a block size of 2000, both Ladder and Conflux show a noticeable drop in throughput, likely due to higher network overhead and resource strain from larger block propagation. Throughput in Ladder fluctuates by around 1000 TPS, while latency varies by approximately 20 seconds. These fluctuations are primarily caused by PoW's randomness in block generation, with additional contributions from LAN congestion and scheduling delays affecting transaction propagation. Despite these fluctuations, Ladder maintains the highest efficiency among all tested approaches.

Impact of Network Sizes: Figures 7(b) and 8(b) show the throughput and confirmation latency of various approaches across different network sizes, with a fixed block size of 1000 and a difficulty level of 18. As the network size grows, throughput increases significantly, primarily due to the higher total computational power from more nodes participating in the PoW, resulting in more frequent block generation. Ladder's median throughput rises from 1043 TPS with 10 nodes to 4506 TPS with 80 nodes. Larger network sizes may introduce longer block propagation delays. However, confirmation latency decreases from a median of 47 seconds with 10 nodes to 34 seconds with 80 nodes, a reduction of 27.7%. The primary reason for this improvement is that increased computational power accelerates block generation, reducing the time required for transactions to be included in a block. This helps mitigate the impact of longer propagation delays by enabling faster confirmation through parallel processing.

Impact of Difficulty: Figures 7(c) and 8(c) show the throughput and confirmation latency at different block difficulty levels with a block size of 1000. Block difficulty significantly affects both throughput and confirmation latency. As the difficulty increases, throughput decreases, and confirmation latency increases across all prior approaches. Specifically, when the difficulty level is 10, the median throughput of Ladder reaches 5314 TPS, which is 27.2% higher than at level 24. This behavior occurs because higher difficulty reduces the block generation frequency, resulting in longer confirmation times. Despite the decline in performance with increasing difficulty, Ladder consistently outperforms all prior approaches.

Result Analysis: Ladder achieves higher throughput and lower latency across varying conditions due to several key

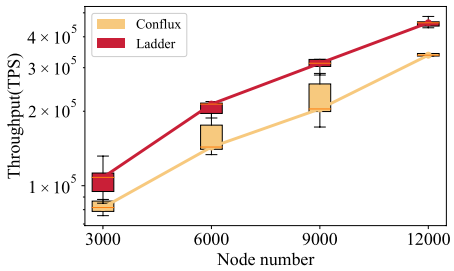


Figure 9: Throughput under large-scale node conditions.

factors. The convergence nodes streamline ledger consolidation by handling transaction ordering. This reduces the computational burden on other nodes, allowing them to focus on parallel block generation. Additionally, Ladder's convergence mechanism simplifies lower-chain block generation and transaction confirmation, thereby reducing overall system complexity. Even during anomalies that require committee intervention, upper-chain block production continues until a new lower-chain block is generated.

Large Scale Simulation: The simulation is conducted on a server with 128GB of memory, with time delays introduced to simulate the block generation and broadcasting processes. Block generation time is positively correlated with block difficulty, while broadcasting time is affected by both block size and the number of participating nodes. The experiments simulated networks of 3000, 6000, 9000, and 12000 nodes. As shown in Figure 9, the results indicate that Ladder's throughput improves by 34.2% compared to Conflux as the number of nodes increases to 12000.

Heterogeneous Nodes: We conduct experiments on Alibaba Cloud using a heterogeneous node setup of 80 virtual machines (VMs): 30 VMs with 4 cores and 8 GB RAM, 30 VMs with 4 cores and 16 GB RAM, and 20 VMs with 8 cores and 16 GB RAM. Due to the intensive hash computations required by PoW, CPU utilization on each VM remains high, ranging from 90% to nearly 100%. Memory usage averages 22.5% on 8 GB VMs and 13.25% on 16 GB VMs, primarily allocated for the transaction pool, blockchain state, and PoW computation caching. We evaluate system performance under varying bandwidth constraints of 10, 20, and 30 Mbps, with 1000 transactions per block and a difficulty level of 18. At 10 Mbps, throughput is 2011 TPS, indicating that bandwidth is the primary bottleneck. Increasing bandwidth to 20 Mbps raises throughput to 3986 TPS, approaching the theoretical maximum. At 30 Mbps, throughput reaches 4652 TPS, showing only marginal improvement over the 20 Mbps results, suggesting that PoW computation efficiency becomes the new bottleneck. These experiments show Ladder's ability to adapt to heterogeneous environments and optimize performance based on available resources and network conditions.

Impact of BFT Committee: We also perform latency testing on the committee in Ladder. Based on our analysis in BFT Committee Size section (Sec. 4), we choose a committee size of 300 nodes. Since committee consensus operates as an independent module, we conduct separate experiments simulating a scenario where nodes simultaneously receive two conflicting lower-chain blocks. We use 80 VMs to emulate a 300-node committee, including 99 Byzantine nodes. Under this setup, the average time cost for committee consensus is 3.25 seconds, which exceeds the lower-chain broadcast time. Although the committee introduces some performance overhead, it plays a crucial role in maintaining system security by mitigating the impact of malicious nodes and ensuring consensus integrity.

6 Security Analysis

6.1 Transaction Trustworthiness

Ladder consists of a growing structure, similar to chain growth in conventional blockchains, where upper and lower-chain blocks serve as the key elements binding the structure. Although the node whose block was selected as the standard upper-chain block in the previous round chooses the next standard block, it cannot always select its own block due to the competitive nature of upper-chain block generation. The probability of a node repeatedly generating blocks without faults is negligible, as it must also handle verification, sorting, and parallel processing of received upper-chain blocks. This makes it further improbable for a node to consistently generate lower-chain blocks in subsequent rounds and keep adding blocks to the upper-chain with invalid transactions. This establishes the trustworthiness of confirmed transactions.

6.2 Resistance Against Common Attacks

This subsection demonstrates how Ladder is resilient against some common attacks.

Sybil Attack [9]: In a Sybil attack, the attacker creates a large number of nodes to manipulate the consensus protocol. As Ladder relies on PoW rather than node count to determine if a given block can be added to the upper-chain, it is resilient against Sybil attacks. Furthermore, only nodes that have previously contributed a standard upper-chain block can generate lower-chain blocks or participate in HotStuff consensus for super block generation. This makes launching a successful Sybil attack on Ladder prohibitively costly.

Denial of Service (DoS) Attack: In PoW-based blockchains, block verification is fast, while block generation is slow and resource-intensive, providing inherent protection against DoS attacks. However, predictable leader election can expose specific nodes to targeted DoS attacks. To mitigate this, Ladder uses VRF for leader selection, reducing the likelihood of successful targeting by adversaries. Even if an attack occurs, other nodes can continue verification and mining without interruption, ensuring the system remains functional.

Double-Spend Attack: To perform a double-spend attack, the adversary cannot immediately engage in malicious activities but must first generate a block that is selected by the pivot chain node as an upper-chain block. The adversary must also

keep the pivot chain operating normally while mining in the next round to generate the lower-chain block. The limited computing power and the hardest chain principle make it nearly impossible for an attacker to execute a double-spend attack within any practically meaningful time frame (*e.g.*, one day). Attacks become highly improbable when adversary resources fall below a certain threshold in Ladder [23].

Eclipse attack [13]: Ladder relies on computing power to establish credibility. While the Eclipse attack impacts the HotStuff consensus component, other nodes that are not part of the BFT committee still generate blocks using PoW. This practically eliminates the impact of Eclipse attacks on the overall system performance.

6.3 Security and Availability

A blockchain network must ensure two key characteristics: security, which ensures that only valid transactions are included in the ledger, and availability, which guarantees that transactions are confirmed within a finite time. Prior work [11] has shown that the three properties of common-prefix, finality, and liveness are sufficient to achieve these characteristics. We define these three properties along with an additional one, and prove that Ladder satisfies both security and availability through lemmas and theorems. Theorems are proven in this section, while proofs of lemmas are provided in the Appendix.

Common-prefix: At any given time, if all nodes in the blockchain network have, with very high probability, the same knowledge of the verified portion of the ledger, the blockchain satisfies the common-prefix property.

Finality: If the verified portion of the ledger of any honest node does not change over time (such as in response to an attempted attack or due to intermittent inconsistencies among nodes), the blockchain ledger satisfies the finality property.

Liveness: If a valid transaction becomes part of the verified portion of the ledger within a predefined amount of time, the blockchain satisfies the liveness property.

l-balance: A sequence of $m \geq l$ nodes is said to be l-balanced when in any subsequence of l or more consecutive nodes, the honest nodes are in majority.

Lemma 1. *The probability that an adversary gets to cast more than $1/3$ of the votes in Ladder's BFT committee (which generates the super block) decreases exponentially with the committee size.*

Lemma 2. *Let k represent the number of blocks in the lower-chain. If the sequence of nodes that generated the given sequence of standard upper- and lower-chain blocks is $2k+2$-balanced, the blockchain ledger generated from the corresponding rounds satisfies the common-prefix property.*

Lemma 3. *If the sequence of nodes that generated the given sequence of standard upper-chain and lower-chain blocks is $2k$-balanced, the blockchain ledger generated from the corresponding rounds satisfies the finality property.*

Lemma 4. *If the sequence of nodes that generated the given sequence of standard upper-chain and lower-chain blocks is $2k$-balanced, the blockchain ledger generated from the corresponding rounds satisfies the liveness property.*

Lemma 5. *The sequence of nodes that generate the given sequence of standard upper-chain and lower-chain blocks is, with very high probability, $2k$-balanced.*

Theorem 1. *Any block in the lower-chain of Ladder is a valid block with very high probability.*

Proof of Theorem 1. For an adversary to generate a lower-chain block in a given round, the adversary must first gain the authority to generate the lower-chain block in that round, which can happen with the probability of just $\frac{q^3+q(1-q)}{q^3+1-q}$ (Appendix B). If an adversary is able to contribute a lower-chain block where it enters any malicious/incorrect information, that will be detected and a super block generation process will start.To succeed in the BFT consensus, the adversary must control more than $1/3$ of the committee. According to Lemma 1, the probability of this happening is extremely low. □

Theorem 2. *Ladder satisfies common prefix, finality, and liveness properties.*

Proof of Theorem 2. By Lemma 5, we know that the sequence of nodes that generated the given sequence of k lower-chain blocks in Ladder satisfies the $2k$-*balance*. Consequently, by Lemmas 2, 3, and 4, we conclude that the resulting blockchain ledger in Ladder satisfies the common prefix, finality, and liveness properties. □

These two theorems, together, prove that Ladder has both security and availability characteristics.

7 Conclusion

We propose Ladder, a structured twin-chain DAG blockchain that optimizes parallel block processing to achieve high throughput, low latency, and resilience against balance attacks. Ladder improves sorting efficiency by designating a convergence node per round, eliminates PoW for lower-chain blocks, and incentivizes node participation for enhanced performance and security. To counter adversarial disruptions, a committee is deployed to generate special blocks when faulty blocks are detected. Experiments on an 80-node network show that Ladder improves throughput by 59.6% and reduces latency by 20.9% compared to state-of-the-art methods.

Acknowledgements

We thank the anonymous NSDI reviewers and our shepherd Prof. Vincent Liu for their constructive feedback and suggestions. This work is supported by the National Natural Science Foundation of China under Grant No. 62032017. Xiulong Liu is the corresponding author: xiulong_liu@tju.edu.cn

References

[1] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[2] Vivek Kumar Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 585–602. ACM, 2019.

[3] Leemon Baird. Hashgraph consensus: Fair, fast, Byzantine fault tolerance. *Swirlds Tech Report*, 2016.

[4] Iddo Bentov, Pavel Hubácek, Tal Moran, and Asaf Nadler. Tortoise and hares consensus: The meshcash framework for incentive-compatible, scalable cryptocurrencies. In *Cyber Security Cryptography and Machine Learning (CSCML)*, volume 12716, pages 114–127. Springer, 2021.

[5] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX, 1999.

[6] Anton Churyumov. Byteball: A decentralized system for storage and transfer of value, 2016. URL: https://byteball.org/Byteball.pdf.

[7] James A. Cowling, Daniel S. Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 177–190. USENIX, 2006.

[8] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient BFT consensus. In *European Conference on Computer System (EuroSys)*, pages 34–50. ACM, 2022.

[9] John R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429, pages 251–260. Springer, 2002.

[10] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[11] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. *Journal of the ACM*, 71(4):1–49, 2024.

[12] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Symposium on Operating Systems Principles (SOSP)*, pages 51–68. ACM, 2017.

[13] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on Bitcoin's peer-to-peer network. In *USENIX Security Symposium (Security)*, pages 129–144. USENIX, 2015.

[14] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *USENIX Security Symposium (Security)*, pages 279–296. USENIX, 2016.

[15] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, 2009.

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Language and Systems*, 4(3):382–401, 1982.

[17] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network, 2018. URL: https://nano.org/en/whitepaper.

[18] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *Financial Cryptography and Data Security (FC)*, pages 528–547. Springer, 2015.

[19] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A decentralized blockchain with high throughput and fast confirmation. In *USENIX Annual Technical Conference (ATC)*, pages 515–528. USENIX, 2020.

[20] Chenxing Li, Fan Long, and Guang Yang. GHAST: Breaking confirmation delay barrier in Nakamoto consensus via adaptive weighted blocks. *arXiv preprint arXiv:2006.01072*, 2020.

[21] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL: https://bitcoin.org/bitcoin.pdf.

[22] Serguei Popov. The Tangle. *White paper*, 1(3):30, 2018.

[23] Meni Rosenfeld. Analysis of hashrate-based double spending. *arXiv preprint arXiv:1402.2009*, 2014.

[24] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. *IACR Cryptol. ePrint Arch.*, page 1159, 2016.

[25] Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. PHANTOM GHOSTDAG: A scalable generalization of nakamoto consensus: September 2, 2021. In *ACM Conference on Advances in Financial Technologies (AFT)*, pages 57–70. ACM, 2021.

[26] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in Bitcoin. In *Financial Cryptography and Data Security (FC)*, pages 507–527. Springer, 2015.

[27] Yonatan Sompolinsky and Aviv Zohar. PHANTOM: A scalable blockdag protocol. *IACR Cryptol. ePrint Arch.*, page 104, 2018.

[28] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving DAG-BFT latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.

[29] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2705–2718. ACM, 2022.

[30] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 135–144. IEEE, 2009.

[31] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[32] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. Sok: Dag-based blockchain systems. *ACM Computing Surveys*, 55(12):1–38, 2023.

[33] WonderNetwork. Global ping statistics: Ping times between wondernetwork servers, Apr. 2018. URL: https://wondernetwork.com/pings.

[34] Jie Xu, Qingyuan Xie, Sen Peng, Cong Wang, and Xiaohua Jia. Adaptchain: Adaptive scaling blockchain with transaction deduplication. *IEEE Transactions on Parallel and Distributed Systems*, 34(6):1909–1922, 2023.

[35] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. Dispersedledger: High-Throughput Byzantine consensus on variable bandwidth networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 493–512, 2022.

[36] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–356. ACM, 2019.

[37] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. OHIE: Blockchain scaling made simple. In *IEEE Symposium on Security and Privacy (SP)*, pages 90–105. IEEE, 2020.

A Selfish Mining Analysis

We analyze the probability of adversary-generated upper-chain blocks exceeding the difficulty of super blocks in the context of selfish mining. If the adversary generates at least m blocks with x leading zeros, it may exceed the difficulty of the k-member super block committee. The probability of generating these blocks, which would exceed the super blocks, can be calculated as $P = \frac{1}{2^{m(x+1)}}$. The difficulty of these upper-chain blocks is $\sum_{i=1}^{m} \sqrt{x} = m\sqrt{x}$, while the super blocks have a difficulty of $\sqrt{k}$. Therefore, the condition for the adversary to surpass the super block's difficulty is $\sqrt{k} \leq m\sqrt{x}$, which implies $x \geq k/m^2$. Substituting this into the probability expression for P, we can obtain that $P = \frac{1}{2^{m(x+1)}} \leq \frac{1}{2^{m(k/m^2+1)}} = \frac{1}{2^{k/m+m}} \leq \frac{1}{2^{2\sqrt{k}}}$.

B Proof of Lemmas

Proof of Lemma 1

Proof. Let $Pr(H|H)$ represent the probability that the honest lower-chain block generation node selects an honest node as the next round lower-chain block generation node. Let $Pr(A|H)$ represent the probability that the honest lower-chain block generation node selects an adversary node as the next round lower-chain block generation node. Let $Pr(H|A)$ represent the probability that the adversary lower-chain block generation node selects an honest node as the next round lower-chain block generation node. Finally, let $Pr(A|A)$ represent the probability that the adversary lower-chain block generation node selects an adversary node as the next round lower-chain block generation node.

If the block generation node in the current round is an honest node, the probability that the block generation node in the next round is an honest node or the adversary corresponds to the respective computing power share. Let q represent the fraction of the computational power contributed by the adversary to the blockchain network. Since there must be a next block generation node, we have $Pr(H|H)+Pr(A|H) = 1$, and $Pr(A|H) = q$, $Pr(H|H) = 1-q$.

If the block generation node in the current round is an adversary node, then if the adversary can compute a legal block within the waiting duration (see Waiting Time Duration in Sec. 4), then the block generation node in the next round will still be an adversary node with probability $Pr(A|A) = \frac{q}{q^2+1-q}$ If the adversary fails to compute a legal block and an honest node computes a legal block, then the block generation node in the next round will be an honest node with probability $Pr(H|A) = \frac{(1-q)^2}{q^2+1-q}$.

The selection of the lower-chain block generation node depends only on the current round's lower-chain block, independent of previous rounds. According to the Markov property, the steady-state probability distribution can be calculated with the calculated state transfer probability, and it can be solved that the probability of the next block generation node being an adversary in each round is less than $1/3$ when the q is less than 0.3.

As we know, the probabilities $Pr(H)$ and $Pr(A)$ can be expressed using the following equations:

$$Pr(H) = Pr(H|H)Pr(H) + Pr(H|A)Pr(A)$$

$$Pr(A) = Pr(A|H)Pr(H) + Pr(A|A)Pr(A)$$

By substituting the expressions for various terms, we can get

$$Pr(H) = \frac{(1-q)^2}{q^3+1-q}$$

and

$$Pr(A) = \frac{q^3+q(1-q)}{q^3+1-q}$$

Let the committee size $k = 3m$, then

$$\begin{aligned} Pr(|A| > 1/3) &= \sum_{i=m}^{3m} Pr(H)^{3m-i} Pr(A)^i \\ &< \sqrt{\frac{8m}{3\pi}} (\frac{16\sqrt{2}}{27})^m \end{aligned}$$

As m increases, the upper bound on the probability decreases exponentially, where $m \geq 3$. □

Proof of Lemma 2

Proof. Suppose the block sequence does not satisfy common-prefix property when the sequence of block generation nodes satisfies $2k$+2-balance. Let the first round be numbered 0. Let the two rounds when the ledger appears to have conflicting transactions/blocks be d and d', such that $d' - d \geq k$. Clearly, there exist at least $2k$+2 block generation nodes. The number of honest nodes $\leq d'$ in the ledger when honest nodes produce at most one block at each height. In the ledger, the number of adversary nodes $\geq (d-0) + (d'-d) = d'$, and the number of adversary nodes $\geq$ honest nodes in the ledger for the block sequence initial block to block d' block generation node sequence, which contradicts the $2k$+2-*balance* property. □

Proof of Lemma 3

Proof. Suppose the block sequence does not satisfy the **Finality** property when the given sequence of block generation nodes is $2k$-balanced. In other words, there exists a block b in round d that is part of ledger when ledger is generated in round $i > d+k$ but not a part of the ledger when ledger is generated in round $j > i$. At this point, for the last $\geq k+1$ blocks of the ledger, there is an inconsistent view, and that would require the block generation nodes to not be $2k$-balanced, contradicting the lemma statement. □

Proof of Lemma 4

Proof. After $2kt$ rounds, the height of the ledger increases by $\geq (k+1)t$, where $\leq (k-1)t$ is generated by the adversary.

Then $\geq (k+1)t-(k-1)t = 2t$ blocks are honest, and among them, $\geq 2t-k$ have been confirmed. This means that at least $2t-k$ honest blocks are added to the block sequence in every $2kt$ rounds.

Therefore, a transaction that is recognized by all honest nodes will definitely be added to the block sequence (ledger) and finally confirmed. □

Proof of Lemma 5

Proof. For a given sequence of length l, where the probability that the adversary accounts for more than 50% does not exceed $exp(-cl)$

By Chernoff's inequality $Pr[X \geq (1+\delta)\mu] < exp(-\frac{\mu\delta^2}{2})$, where $\mu = ql, \delta = \frac{1-2q}{2q}$, we obtain

$$\begin{aligned} Pr[X \geq (1+\delta)\mu] &= Pr[X \geq \frac{l}{2}] \\ &\leq exp(-\frac{l(1-2q)^2}{8q}) \\ &= exp(-cl), c > 0 \end{aligned}$$

For a sequence of block generation nodes of length L, any subsequence of length at least l has a probability of at most $\frac{(L-l+1)(L-l+2)}{2} exp(-cl) < L^2 exp(-cl)$ that the adversary controls more than 50%.

Then $Pr[Ladder's\ node\ sequence\ satisfies\ 2k\text{-}balance] > 1-\delta(l)$ and $\delta(l)$ decreases exponentially with l. □

Vegeta: Enabling Parallel Smart Contract Execution in Leaderless Blockchains

Tianjing Xu[1*], Yongqi Zhong[1*], Yiming Zhang[1,2*], Ruofan Xiong[3], Jingjing Zhang[4], Guangtao Xue[1,2], Shengyun Liu[1,2#]

[1]*Shanghai Jiao Tong University* [2]*Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3* [3]*Xiamen University* [4]*Fudan University*

Abstract

Consensus and smart contract execution play complementary roles in blockchain systems. Leaderless consensus, as a promising direction in the blockchain context, can better utilize the resources of each node and/or avoid incurring the extra burden of timing assumptions. As modern Byzantine-Fault Tolerant (BFT) consensus protocols can order several hundred thousand transactions per second, contract execution is becoming the performance bottleneck. Adding concurrency to contract execution is a natural way to boost its performance, but none of the existing frameworks is a perfect fit for leaderless consensus.

We propose *speculate-order-replay*, a generic framework tailored to leaderless consensus protocols. Our framework allows each proposer to (pre-)process transactions prior to consensus, better utilizing its computing resources. We instantiate the framework with a concrete concurrency control protocol Vegeta. Vegeta speculatively executes a series of transactions and analyzes their dependencies before consensus, and later deterministically replays the schedule. We ran experiments under the real-world Ethereum workload on 16-vCPU virtual machines. Our evaluation results show that Vegeta achieved up to 7.8× speedup compared to serial execution. When deployed on top of a leaderless consensus protocol with 10 nodes, Vegeta still achieved 6.9× speedup.

1 Introduction

Blockchain or state machine replication (SMR) is a promising solution for providing trustworthy services among a group of untrusted entities or *nodes*. To mask failures, blockchain systems rely on a consensus protocol [32, 50, 51, 58] to reach an agreement on the sequence of blocks (and transactions). Each node is abstracted as a state machine [65], which sequentially executes the ordered transactions based on the same initial *world state*. Towards wide-scale adoption, boosting performance of blockchain systems is becoming a hot topic in both academia and industry [36, 59, 68, 77].

As Nakamoto-style consensus (such as Bitcoin [58] and Ethereum 1.0) has limited performance, modern permissionless and permissioned blockchains [17, 24, 37] are adopting Byzantine-Fault Tolerant (BFT) consensus protocols [22, 25, 29, 49, 57, 78] as their consensus layer. Most classical BFT protocols are leader-based [22, 25, 39, 78], meaning that a single leader is in charge of disseminating proposals and coordinating consensus. To select a single and stable leader, however, they rely on additional timing assumptions which may not be realistic in large-scale Wide Area Networks (WANs). Besides, such a single node may easily become the performance bottleneck [47, 68, 76].

Recently, many leaderless protocols are proposed targeting deployment in blockchain systems. Leaderless variants either have no timing assumption to elect any special role [17, 29, 41, 55, 57, 60], or assign the leader role to every node [28, 68]. Each node is required to act as a proposer and make a block proposal, and the final decision is a (sub)set of all proposals. Leaderless protocols are becoming an appealing solution to blockchain systems. For instance, Sui [7, 38], HashGraph [17] and Conflux [52] use a DAG-based protocol as their consensus layer, while AntChain leverages an asynchronous consensus for adaptive advancement [55].

BFT consensus protocols nowadays can process hundreds of thousands of transactions per second (TPS) [29], or even beyond one million [19] when deployed within a LAN. In contrast, the serial transaction execution engine of Ethereum platform only achieves around 100 TPS throughput [44], throttling the whole system. A natural way to improve efficiency of smart contract execution is to leverage modern multi-core architecture and add concurrency to transaction processing [26, 30, 36, 53].

When designing a concurrency control protocol in blockchain contexts, we argue that the consensus layer should also be taken into consideration as they play complementary roles. Currently, there are two trends in designing such pro-

*Equally contributed authors.
#Corresponding author: shengyun.liu@sjtu.edu.cn

tocols [66]: the *order-execute* framework [15, 36, 48], where transactions are (concurrently) executed after consensus in a deterministic way and produce the same results; and the *execute-order-validate* framework [14, 42, 46], where transactions are first speculatively executed by proposer(s) before consensus, and (re-)executed and/or validated by other nodes after consensus. In the order-execute framework, transaction execution is totally decoupled from consensus and thus can be readily plugged into any consensus protocol (leader-based or leaderless). The execute-order-validate framework allows proposer(s) to orchestrate transaction execution before consensus, such that other nodes can follow the order produced by proposer(s) and execute them in an efficient manner.

Despite being a more promising direction towards blockchain deployments, leaderless consensus introduces new challenges for efficient transaction processing, and neither of the aforementioned frameworks is a perfect fit. On the one hand, the order-execute framework cannot fully inherit the scalability feature from leaderless consensus, as each node must execute every block after consensus. Besides, they must schedule transactions in a deterministic way in order to produce the same execution results, limiting speedups when inconsistency resolution is frequently involved. On the other hand, the execute-order-validate framework either targets leader-based consensus [15, 42, 44, 46], or aborts transactions if new conflicts are introduced after consensus [14]. In the second case, aborted transactions must go through the whole procedure (i.e., execute, order and validate) again. Besides, neither takes leaderless consensus into special consideration.

We observe that the transaction execution layer of any blockchain system or SMR protocol can be divided into two parts: the pre-consensus part that allows every node to individually process distinct transactions, and the post-consensus part that requires every node to perform the same set of tasks (e.g., transaction execution and validation). To better adapt to leaderless consensus and improve scalability, in general we should place more work in the former if this can effectively reduce the burden of the latter.

Notably, the idea of allowing nodes to (pre-)process transactions prior to replication has been extensively explored in deterministic databases [40, 45, 69, 73, 79]. For instance, for transactions which cannot determine read/write sets a priori (also named dependent transactions), reconnaissance queries [72] is typically performed before replication. This pattern is a natural fit for transferring the scalability feature of leaderless consensus to the smart contract layer and thus deserves further study.

We accordingly propose a *speculate-order-replay* framework for efficient smart contract execution in leaderless blockchains. Before disseminating block proposals, each node first speculatively executes a series of transactions and/or performs any task that may help accelerate post-consensus tasks. Once blocks are totally ordered, each node enters the *replay* phase, during which it tries to (efficiently) execute transactions with the help of the information provided by speculation. Because in leaderless consensus there exist other concurrent proposals, such information may not be accurate with regard to the actual execution context. In case some transactions/operations introduce new conflicts in the replay phase, they must be processed in a deterministic way.

We instantiate this framework by a concrete concurrency control protocol (Vegeta). Vegeta realizes transaction-level parallelism, meaning that each transaction is treated as an indivisible unit. In the speculation phase, Vegeta executes transactions in a fully parallel manner, and obtains their read/write sets in order to construct a DAG that encapsulates the dependency information. In the replay phase, each node tries to "replay" the schedule provided by speculation. If a transaction tx produces the same read/write sets, its execution is finalized. Otherwise, if tx introduces new dependencies by accessing new keys, tx is re-executed after other transactions complete. The rationale of Vegeta is simple: although proposers are less likely to obtain correct execution results based on an inaccurate world state, the dependency information provided is most likely accurate.

We have implemented Vegeta in Golang and evaluated it with the Ethereum workload, meaning all the transactions and blocks are real and taken from the most representative smart contract platform. We ran experiments on Amazon EC2 platform, both on a single virtual machine and among a group of at most 10 machines in order to evaluate Vegeta with a leaderless consensus protocol. The evaluation results demonstrate that Vegeta on a single machine achieved an average speedup of 7.8× compared to serial execution. We also evaluated the performance of AriaFB [56], a deterministic optimistic Concurrency control protocol. Vegeta outperformed AriaFB by up to 2.1×. We postpone the proof of Vegeta to Appendix A due to space limit.

2 Background and Related Work

2.1 System Model

We focus on the blockchain or state machine replication (SMR) problem [65], a solution to which is a highly reliable and available distributed system. Each node is abstracted as a state machine and sequentially executes a series of requests or *transactions* issued by clients. We target authenticated Byzantine failure model [51], meaning that the faulty nodes may exhibit any malicious behavior except for breaking cryptographic schemes. We assume at most f nodes can be faulty and there are a total of $n = 3f + 1$ nodes, the lower bound for BFT SMR protocols under asynchrony assumption [32].

Transaction semantics. Each transaction contains several operations and may read and/or write a set of keys or objects, which are collectively maintained as a world state in blockchains [5]. The world state can be simply considered a key-value store. Blockchain systems provide serializabil-

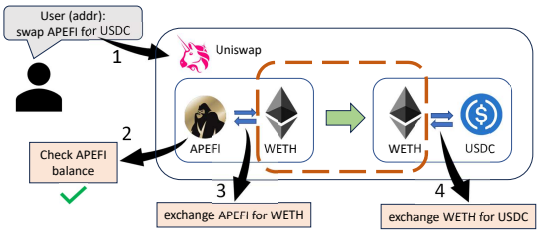
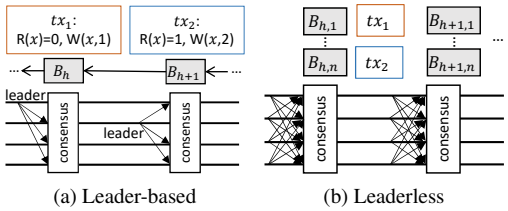


Figure 1: Example for Ethereum smart contract invocation.



(a) Leader-based (b) Leaderless

Figure 2: Comparison between leader-based and leaderless consensus protocols.

ity [21] semantics, in the sense that any execution result is equivalent to the outcome of sequentially executing all transactions in some (pre-defined) order. To enable parallel processing, we further distinguish between conflicting and non-conflicting transactions. We say two transactions tx_1 and tx_2 are *conflicting* if they access the same key x and at least one of them writes x. Otherwise, tx_1 and tx_2 are *non-conflicting*.

2.2 Smart Contracts

Smart contracts [71] implement application logic and can be simply considered a piece of code deployed on top of blockchains. Each transaction invokes the code of a contract, which may recursively invoke another contract.

In Figure 1 we give an example for Ethereum smart contract invocation. This example [2] is stripped off from the Ethereum mainnet. A user wanted to trade some Ape Finance tokens (APEFI) [1] for USDC [9], a stablecoin pegged to US dollar. Uniswap [8] is a Decentralized Exchange (DEX) contract for facilitating token swap. The user first invoked the Uniswap contract (step 1), which then checked the balance of the user by invoking the APEFI contract (step 2). At that particular moment, there was not a liquidity pool available for exchanging APEFI and USDC. Hence, the Uniswap contract leveraged Wrapped ETH (WETH) [10] as an intermediary token for achieving the desired trade (step 3). Finally, the Uniswap contract exchanged WETH for USDC (step 4). By this example we can observe, when a user initiates a transaction, the user is not aware of the (full) list of addresses (or keys) that the transaction will access. To accurately obtain such information, an Ethereum node must execute this transaction based on the most recent world state.

2.3 Leader-Based vs. Leaderless Consensus

Existing SMR protocols can be roughly classified into two categories according to the ways they order proposals: leader-based and leaderless. Leader-based protocols rely on a special role (named leader) to coordinate consensus, i.e., disseminating proposals to and/or collecting votes from other nodes.

We take PBFT [25] and HotStuff [78], two representative leader-based BFT protocols for example. PBFT assigns the leader role to a single node until this node is faulty or partitioned, upon which a view change sub-protocol is triggered to elect a new leader. HotStuff proactively rotates the leader role, ensuring that every node has the opportunity to propose. Upon making a new block proposal B_{h+1} (see Figure 2a), the leader is aware of the most recent proposal B_h that precedes B_{h+1}. Transitively, the leader knows all the previous proposals. Thus, the leader can pre-execute tx_2 based on the execution result of tx_1, which is included in B_h. In summary, leader-based protocols have the following feature:

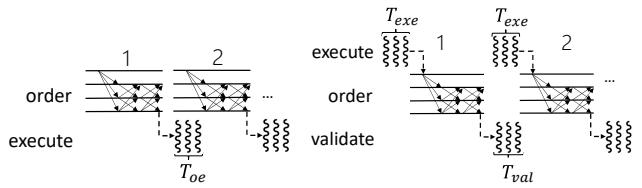
Definition 1. *(Perfect Context) we say a consensus protocol has perfect contexts, if when any node makes a proposal $\mathcal{B}$, the node is aware of an ordered list of all proposals that are (or will be) committed before $\mathcal{B}$.*

Both PBFT and HotStuff and several permissionless blockchains such as Bitcoin [58] and Ethereum [5] have perfect contexts. With this feature, the proposer (e.g., the single leader) can (pre-)execute all transactions before consensus, give a total or partial order among them and insert (the digest of) execution results as well as transaction dependencies into the proposal. Other nodes, upon receiving the proposal, can (re-)execute the transactions in the same pre-defined order and obtain the same results.

Leaderless protocols, in contrast, have no fixed leader role but allow every node to act as a proposer. One type of leaderless protocols [16,28,68] pre-partitions sequence space and assigns them to every node, while another type [29,31,38,41,57] is designed under asynchrony assumption. The final output of leaderless protocols is a combination of all committed proposals. By allowing each node to act as a proposer, leaderless protocols can distribute workloads and better utilize resources (e.g., CPU and network capacity). Since leaderless protocols allow concurrent proposals made by multiple proposers, they have no perfect context. For instance, in Figure 2b, concurrent proposals $B_{h,1},\ldots,B_{h,n}$ are proposed by distinct nodes, and hence are totally ordered only after consensus. Their proposers have no perfect knowledge of other concurrent proposals and thus cannot get correct results.

2.4 Parallel Execution

To integrate parallel execution into blockchains or SMR protocols, existing approaches can also be categorized into two types: the order-execute and execute-order-validate frameworks, as shown in Figure 3.



(a) order-execute (b) execute-order-validate

Figure 3: Existing frameworks for parallel execution.

Order-execute. The order-execute framework [15, 36, 48] (Figure 3a) orchestrates a (parallel) execution phase that emerges after consensus. Each node individually tries to parallelize transaction execution in a deterministic way. Typically, the results of parallel execution must be equivalent to the ones obtained by sequentially executing those transactions in the order given by consensus. The order-execute framework "blindly" imposes constraints on the order of transactions and may incur additional costs for, e.g., sequentially executing non-conflicting transactions and/or re-executing conflicting ones. The execution component in order-execute is decoupled from consensus, and thus can be integrated into any consensus protocol (leader-based or leaderless). Saber [54] improves transaction execution by allowing different groups to execute different transactions. The idea of Saber is akin to inter-node parallelism and largely orthogonal to our contribution.

Execute-order-validate. The execute-order-validate framework [14, 18, 42, 46, 67] (Figure 3b), in contrast, enables proposer(s) to (pre-)execute transactions before consensus. There are two trends in instantiating this framework: they either target a leader-based consensus protocol [18, 42, 67] or resolve conflicts in a trivial way [14, 46]. In the former case, the single leader orchestrates transaction execution and re-orders them if necessary, while other nodes follow the schedule and accordingly execute transactions in the same pre-defined order. In the latter case, existing protocols tend to *trust* the execution results obtained prior to consensus, but resort to a heavy fallback mechanism to resolve newly-introduced conflicts. For example, Eve [46] assumes the read/write sets of each transaction can be obtained a priori, otherwise it must rely on a leader-based protocol (i.e., PBFT) to resolve conflicts. Hyperledger Fabric [14] requires every peer to speculatively execute transactions and later reach consensus also on execution results, so as to deal with non-deterministic operations. If new conflicts are introduced after consensus, conflicting transactions must go through the whole procedure again.

Deterministic database systems. Concurrency control has long been a hot topic in database systems [13, 21, 64]. There are numerous works targeting deterministic scheduling and execution [12, 33, 34, 43, 72–74]. To enable parallel execution while still producing identical results, deterministic databases usually require the reach/write sets of each transaction can be obtained a priori. Otherwise, reconnaissance queries [33, 61, 62, 72, 73] or speculative execution should be performed before transaction scheduling. Such a technique [72] decomposes a *dependent transaction* with unknown read/write sets into a combination of some read-only transactions and a conditional-write transaction. The read-only transactions discover the read/write sets (say, before replication),while the conditional-write transaction commits (after replication) if the read values match. Calvin [73] adopts a lock-level concurrency control protocol and orchestrates lock acquisition based on a pre-defined total order. Calvin does not readily support dependent transactions, but can leverage the mechanism discussed above (also named Optimistic Lock Location Prediction, OLLP) to obtain read/write sets. Caracal [62] and Epic [61] utilize multi-version concurrency control (MVCC) to reduce conflicts and can also run a similar identification phase to determine read/write sets. Reconnaissance query is analogous to the speculation in Vegeta, but Vegeta and our framework specifically target leaderless blockchains (under Byzantine faults) and are more focused on co-design between pre- and post-consensus tasks. Besides, Vegeta does not consider the speculation output as potential results, but only relies on it for dependency analysis.

Aria [56] does not require read/write sets to be known before execution but first executes transactions batch by batch in a fully parallel manner. If any transaction *tx* introduces conflicts with an earlier transaction in the same batch, *tx* is aborted and re-tried with the next batch. When a workload results in a very high abort rate, Aria further resorts to a fallback mechanism (e.g., Calvin [73]) to deterministically resolve the conflicts, instead of repeatedly trying aborted transactions. In a sense, Vegeta uses a similar batching mechanism in the replay phase, but further leverages the speculation phase to wisely group transactions into batches. Morty [23] is a recently-proposed concurrency control method that leverages transaction re-execution to improve throughput. Our method instead introduces re-execution to ensure consistency among different nodes. Basil [70] is a newly-proposed BFT transactional storage system, which demonstrates that the leaderless feature can effectively improve performance.

3 The Speculate-Order-Replay Framework

We observe that the successful story of leaderless consensus protocols stems from the equality of all nodes in disseminating proposals and processing transactions [75], thus effectively balancing workloads. We intend to further achieve such a feature in the contract execution layer. To this end, we study a generic framework called speculate-order-replay (SOR, as shown in Figure 4). Similar to utilizing resources of all nodes in a leaderless consensus protocol, the speculate-order-replay framework allows each node to process and speculatively execute distinct transactions before consensus. We call this step *speculation*. Although proposers do not have a perfect execution context, they may still obtain crucial information about transaction conflicts, read/write sets and program traces,

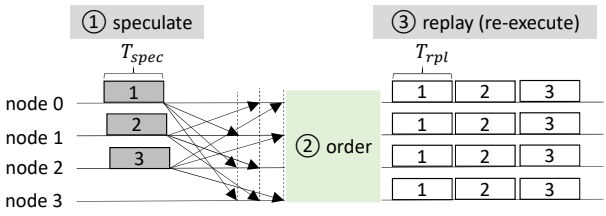


Figure 4: The speculate-order-replay framework.

which can greatly help maximize parallelism.

Once the speculation phase completes, each proposer takes the transactions and other information as input and enters the *ordering* phase. The ordering phase is run solely by a consensus protocol, and thus we omit its detail in this paper. Once the sequence of committed proposals is agreed upon by consensus, each node enters the *replay* phase. In general, conflicting operations must be executed in the same (serial) order, while non-conflicting operations can be processed in parallel. If the information provided by speculation is almost accurate, nodes in the replay phase may execute transactions with a maximum degree of parallelism, while at the same time paying no/few extra costs for handling (unnecessary) aborts. However, if some transactions impose new conflicts, they must be carefully treated. We postpone the detail to §4 as it is related to specific parallelism schemes.

Analysis of speedups. We give a simple analysis for demonstrating why the SOR framework may scale better with regard to the speedup achieved. The results are also shown in Table 1. We assume on average, sequentially executing a block of transactions takes T_{seq} seconds, which is the baseline for further comparison. Executing m blocks takes mt_{seq} seconds. Similarly, the order-execute framework (Figure 3a) takes mT_{oe} seconds to execute m blocks if each block on average costs T_{oe} seconds. Compared to serial execution, the speedup achieved by the order-execute framework is $\frac{T_{seq}}{T_{oe}}$.

As for the execute-order-validate framework (Figure 3b), each block first goes through an execution phase, which takes T_{exe} seconds. After consensus, each node validates the execution results by (re-)executing them, which takes T_{val} seconds. Thus, each block incurs $T_{exe}+T_{val}$ seconds for execution. Due to pipelining techniques, the execution phase of B_{h+1} can be overlapped with the validation phase of B_h. By doing so, the speedup approaches $\frac{T_{seq}}{T_{exe}}$ if m is large and $T_{exe} \geq T_{val}$. The latter assumption holds because the leader usually takes more time to orchestrate transaction execution [15].

Finally, we assume in SOR, the speculation phase takes T_{spec} seconds, while the replay phase takes T_{rpl} seconds (Figure 4). We further assume transactions are equally distributed to each node, which is an ideal assumption for SOR and in practice can be implemented by, e.g., applying load-balancing [68]. For each block, the speculation phase is only executed by one node, and different blocks can be speculatively executed in parallel. Thus, each block still takes $T_{spec}+T_{rpl}$ seconds, but the speedup, which takes m blocks into consideration, is $\frac{mT_{seq}}{mT_{spec}/n+mT_{rpl}} = \frac{T_{seq}}{T_{spec}/n+T_{rpl}}$.

Table 1: Analysis of speedups of different frameworks. $T_{oe} \approx \frac{T_{seq}}{\#cores} \approx T_{exe} \approx T_{val}$ if very few conflicts between transactions exist (which is not the case for Ethereum workloads). In contrast, $T_{rpl} \approx \frac{T_{seq}}{\#cores}$ if the information provided by the speculation phase is almost accurate and the degree of parallelism is approximately equal to the number of cores (though with many conflicts).

	latency	speedup
sequential	T_{seq}	-
order-execute	T_{oe} (Figure 3a)	$\frac{T_{seq}}{T_{oe}}$
execute-order-validate	T_{exe}+T_{val} (Figure 3b)	$\frac{T_{seq}}{T_{exe}}$
speculate-order-replay	T_{spec}+T_{rpl} (Figure 4)	$\frac{T_{seq}}{T_{spec}/n+T_{rpl}}$

We can observe that the bottleneck of speculate-order-replay mainly locates at the replay phase, which must be performed by every node. Intuitively, we may (moderately) put additional efforts into speculation if T_{rpl} can decrease. In contrast to order-execute, nodes in our framework can freely choose in what order transactions are executed, thus maximizing parallelism and alleviating re-execution burden. Compared to execute-order-validate, our solution is tailored to leaderless protocols and may better scale as nodes can speculate in parallel.

The SOR framework can be instantiated on different layers, e.g., transaction-level parallelism (Vegeta), lock-level parallelism or instruction-level parallelism. One may even integrate other techniques (e.g., zero-knowledge rollups [11] or I/O prefetching [27]) into SOR, so long as they help alleviate the burden of replay and the extra costs of speculation are acceptable.

4 Vegeta

We then elaborate on Vegeta, which follows the speculate-order-replay framework and implements transaction-level parallelism. Vegeta does not need to maintain a multi-version key-value store, nor does it need to roll back any transaction.

4.1 Speculation

The main purpose for the speculation phase is to give a specific (partial) order among transactions within a block, such that by following this order each node (in the replay phase) can efficiently execute them. A primary advantage of processing transactions prior to consensus lies in the absence of (unnecessary) ordering constraints between transactions. In

Algorithm 1: Speculation phase

```
Variables: BLOCK: a block containing a list of transactions
           DAG: a two-dimensional array storing dependencies
           access: a map from key to tx_chain
1  procedure Speculate():
2      txs ← BLOCK.transactions
3      ParallelExecute(txs)
4      tx_chains ← SortDependencyChains(txs)
5      txs ← ∅
6      for chain ∈ tx_chains do
7          for tx ∈ chain do
               /* Adjust the order of txs                    */
8              if tx ∉ txs then
9                  append(txs, tx)
10     BLOCK.transactions ← txs
11     DAG ← BuildDAG(txs)
12     call Consensus(⟨BLOCK, DAG⟩)
13 function ParallelExecute(txs):
14     for tx ∈ txs do
           /* Run in parallel mode                           */
15         (tx.RS, tx.WS) ← execute(tx)
16 function SortDependencyChains(txs):
17     for tx ∈ txs do
18         for ∀ key ∈ tx.RS ∪ tx.WS do
19             append(access[key], tx)
20     for chain ∈ access do
21         append(tx_chains, chain)
22     Sort tx_chains from the longest to the shortest
23     return tx_chains
24 function BuildDAG(txs):
25     for tx ∈ txs do
26         for ∀ tx′ : tx′.idx < tx.idx do
27             if tx′.WS ∩ tx.WS ≠ ∅ then
28                 DAG [tx][tx′] ← WAW
29                 continue
30             if tx′.RS ∩ tx.WS ≠ ∅ then
31                 DAG [tx][tx′] ← WAR
32             if tx′.WS ∩ tx.RS ≠ ∅ then
33                 if DAG [tx][tx′] == WAR then
34                     DAG [tx][tx′] ← WAW
35                 else
36                     DAG [tx][tx′] ← RAW
37     return DAG
```

other words, each proposer can orchestrate an order at will, maximizing parallelism as much as possible.

On the contrary, any pair of conflicting transactions need to be processed sequentially (if they access the same key and one of them writes the key). Those conflicting transactions form a so-called dependency chain, which is the main enemy of high degree of parallel processing. The long dependency

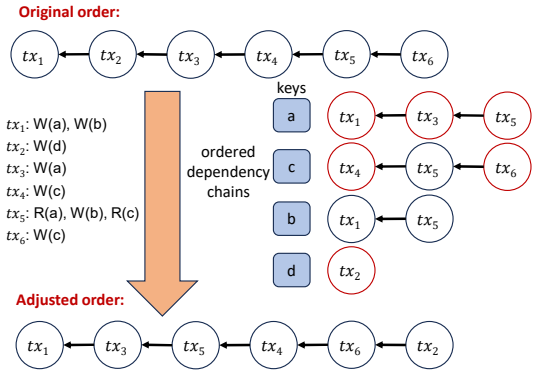


Figure 5: Example for the speculation phase. After obtaining the read/write sets, transactions are (re-)ordered according to the lengths of dependency chains. The corresponding DAG is depicted in Figure 6.

chain problem also appears in the Ethereum workload and was discussed in [35]. The heuristic here is we should give transactions within a longer dependency chain a higher priority, and thus in the replay phase the longer chain starts execution prior to other shorter chains. We thus have the following rule:

Rule 1. *Transactions should be sequenced in descending order based on the lengths of their dependency chains.*

The pseudocode is given in Algorithm 1. First, each node gets a series of transactions and executes them in parallel (Line 2-3). To obtain transaction dependencies, the proposer only needs the read/write sets of each transaction. Thus, at this stage transactions are executed in parallel and will not modify the world state. Based on the read/write sets, we are able to analyze transaction dependencies and create a schedule for the replay phase.

To this end, we first count the number of transactions that have accessed a given key and group them into a dependency chain (Line 17-21). A transaction may appear in several chains. Note that this simple method only obtains an approximation of accurate dependency chains, as each chain only takes one key for consideration. Nonetheless, this simple solution already leads to prominent performance (as shown in §6). Then, we sort different chains according to their lengths (Line 22). We finally arrange the transactions within a longer chain in front of those within a shorter chain and obtain an ordered list *txs* (Line 6-9).

Based on the order given in *txs*, we can analyze their dependencies and accordingly build a Directed Acyclic Graph (DAG) (Line 24-36). The dependency relationship forms a DAG (with no circle) because any transaction may only depend on ones with lower indices, but not vice versa. There are three types of dependencies: write after read (WAR), write after write (WAW) and read after write (RAW). If a transaction writes the same key that its previous transaction reads, the corresponding place of these two transactions in DAG is tagged with WAR. Other two types of dependencies are

tagged similarly. Note that if two transactions have multiple dependencies, we prioritize WAW (Line 27-29) due to a specific optimization [56] that we will elaborate on §4.2. Also note that if two transactions have both WAR and RAW dependencies (for different keys), we treat them as having a WAW dependency (Line 34). Finally, the transactions, their read/write sets, as well as the DAG are inserted into a block proposal (Line 12). With such information, every node in the replay phase can process these transactions deterministically.

4.2 Replay

After consensus, blocks are totally ordered and each node enters the replay phase. The main purpose for replay is to deterministically and efficiently execute all transactions within each block. We only exploit intra-block parallelism, though enabling inter-block parallelism may further speed up execution, which we leave for future work. The replay phase is further divided into two stages: (1) the first stage that follows the DAG and execute transactions accordingly; and, (2) the second stage that re-executes transactions that may introduce new dependencies. For the first stage, our concurrency control method is inspired by Aria [56]. For the second stage, we first describe a scheme that simply re-executes all possibly affected transactions. In §4.3 we describe an optimization that further reduces re-execution rate.

The pseudocode is given in Algorithm 2. Each node gets a block, the read/write set of each transaction, as well as the DAG from the consensus component (Line 2). To enable parallel processing, transactions within a block are further divided into several *batches*, where in each batch all the transactions can be executed in parallel (if they do not access some new keys compared to the ones obtained in the speculation phase). Transactions within a block are executed in a batch-by-batch manner, and in each batch the execution results will not update the world state until all the transactions complete.

Each node repeatedly obtains a batch of transactions that have no *forward dependency* from the transaction list *txs* (Line 4), then executes them in parallel. We have the following specific rule for defining whether a transaction has no forward dependency:

Rule 2. *A transaction tx has no forward dependency if: (1) tx has no WAW dependency on any previous transaction that has not completed, and (2) tx does not have both WAR and RAW dependencies on any previous transactions that have not completed.*

The intuitions of Rule 2 are: (1) even if transaction tx WAR depends on tx', tx can start execution immediately because its update will not be visible to tx', as they are in the same batch; and (2) even if tx RAW depends on tx', tx can be re-ordered before tx' because the update of tx. The idea of Rule 2 is elaborated in [56].

Once tx finishes execution, we need to check its newly-generated read/write sets. If the read/write sets have changed

Algorithm 2: Replay phase

```
   Variables: TxsRe: a set of transactions to be re-executed
1  procedure Replay():
2      (BLOCK, DAG) ← Consensus()
3      txs ← BLOCK.transactions
4      ready ← PopTxsBatch(txs)
5      while ready ≠ ∅ do
6          for tx ∈ ready do
               /* Run in parallel mode                    */
7              (rs, ws) ← execute(tx)
8              if rs ≠ tx.RS ∨ ws ≠ tx.WS then
9                  append(TxsRe, tx)
10                 ready ← ready \ {tx}
11         for tx ∈ ready do
12             commit(tx)
13         ready ← PopTxsBatch(txs)
14 procedure ReExecute():
15     for tx ∈ TxsRe do
16         execute(tx)
17         commit(tx)
18 function PopTxsBatch(txs):
19     ready_txs ← ∅
20     for tx ∈ txs do
21         if tx has no WAW-dependencies then
22             if no WAR ∨ no RAW then
23                 append(ready_txs, tx)
24                 txs ← txs \ {tx}
25     return ready_txs
26 function commit(tx):
27     Apply updates in tx.WS to the world state
```

compared to the results obtained in the speculation phase, tx may introduce new dependencies with other transactions in the same block, and thus will be *re-executed* at the end of the replay phase (Line 8-10). Nonetheless, we can still commit other transactions in *ready* by updating the world state according to their write sets (Line 11).

Finally, we present a simple re-execution stage that happens at the end of the replay phase (Line 14-17). Transactions in $TxsRe$ are sequenced based on their indices and executed in serial order. In §A we prove that $TxsRe$ is the same among all correct nodes. In Figure 6 we depict an example for demonstrating the replay phase.

4.3 Reducing Re-Execution Rate

Because transaction re-execution may badly affect performance, we further improve our strategy for determining whether a transaction needs to be re-executed. The new strategy is described in Algorithm 3, which replaces Line 5-12 in Algorithm 2.

Remember that in Algorithm 2, a transaction should be re-

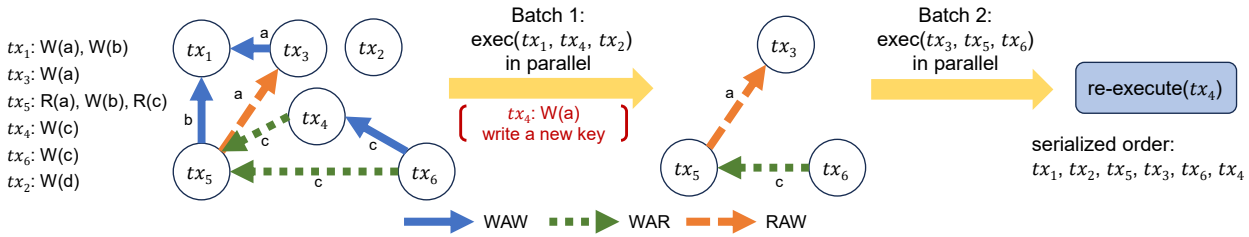


Figure 6: Example for the replay phase. On the left we show the read/write sets of each transaction obtained in the speculation phase. Batch 1 contains tx_1, tx_4 and tx_2 as they have no forward dependency (Rule 2). tx_4 newly writes key a, and thus will be re-executed at the end. Batch 2 contains tx_3, tx_5 and tx_6, even though tx_6 WAR depends on tx_5. The execution results are equivalent to executing tx_1, tx_2, tx_5, tx_3, tx_6 and tx_4 in serial order.

Algorithm 3: Improved replay strategy

```
1  all_keys ← all keys accessed by tx in txs in speculation
2  while ready ≠ ∅ do
3      for tx ∈ ready do
           /* Run in parallel mode                    */
4          (rs, ws) ← execute(tx)
5          if tx reads/writes a new key key' ∈ all_keys then
6              append(TxsRe, tx)
7              ready ← ready \ {tx}
8              continue
9          if tx reads a new key key' ∉ all_keys then
10             append(read, tx)
11         if tx writes a new key key' ∉ all_keys then
12             new_keys ← new_keys ∪ {key'}
13     read ← read \ {tx : tx has not read any key ∈ new_keys}
14     commit (tx ∈ ready \ read) based on DAG
15     for tx ∈ read do
16         execute(tx)
17         commit(tx)
```

executed once its read/write sets have changed. In fact, some changes do not necessarily lead to re-execution, as they do not introduce new dependencies. For instance, if tx accesses fewer keys in the replay phase. We intend to further reduce re-execution rate, but at the same time keeping the method simple and straightforward to be implemented. To this end, we first group all the keys that have been accessed in the speculation phase (Line 1 in Algorithm 3). Then we further distinguish the following three cases:

1. If a transaction tx reads/writes a new key that other transactions also accessed during speculation (Line 5-8), it is highly possible that tx will introduce new dependencies. In this case, tx will be re-executed.

2. If tx reads a new key key' that other transactions did not access during speculation (Line 9-10, 15-16), key' may not necessarily introduce new dependencies, but tx (in *read*) has to wait until other transactions in the batch complete execution. If some transaction has newly written key' (in *new_keys*), tx will be executed again and get committed. Otherwise, tx is removed from *read* and committed with other transactions by following the serial order extracted from DAG.

3. If tx writes a new key key' while other transactions did not access key' during speculation (Line 11-14), we record key' in *new_keys*. tx will also be committed with other transactions once all the transactions complete execution.

In other cases, tx will not be re-executed.

4.4 Further Discussion

This paper solely focuses on improving the performance of contract execution. In decentralized finance applications, however, transaction ordering is also crucial for participants' profits. Regarding the speculation phase in Vegeta, miners or proposers may be motivated to initiate frontrunning and sandwich attacks [3, 63] to increase their profits, rather than maximizing parallelism. To defend against adversarial players, existing literature either introduces additional restrictions on transactions ordering (e.g., by providing a fairness property [80]), or resorts to some economic mechanism (e.g., the proposer/block-builder separation [6]). For the former case, proposers cannot freely choose at what order transactions are replayed, thus limiting its performance gains. For the latter case, we may quantify the degree of parallelism and also leverage an incentive mechanism to motivate rational proposers.

5 Implementation

We have implemented Vegeta in Golang and adopted Go-Ethereum, a.k.a., Geth as our Ethereum execution environment. We integrated Ethereum Virtual Machine (EVM) [4] with Vegeta to enable transaction execution. We still used the data structure *StateDB* in EVM for caching and storing the world state in memory. *StateDB* can be simply considered an in-memory key-value store. As EVM only provides single-threaded execution, we modified *StateDB* in order to support shared access. To do so, we used a sync.Map structure

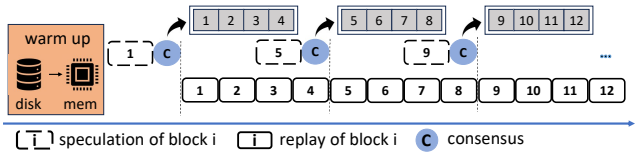


Figure 7: Example from node 1's perspective for consensus-integrated implementation ($n = 4$).

for thread-safe state management. We started a new EVM instance for each transaction, and all instances can concurrently access *StateDB*.

To focus on transaction execution performance, we skipped updating the Merkle Patricia Trie (MPT) in *StateDB* (after each execution). MPT is a combination of Patricia trie and Merkle tree. MPT is used to cryptographically authenticate and verify the content of the world state. One exception is we used MPT for correctness validation (§6.1), which updated MPT after the execution of each block. We also disabled data persistence to eliminate its impact on performance evaluation, allowing us to focus solely on transaction execution.

Consensus integration. In Vegeta the speculation phase of a given block B_h may overlap with the replay phase of some earlier block $B_{h'}$ ($h' < h$), so the two phases may compete for computing resources on each node. To study this impact, we also integrated a consensus component, which is a seminal leaderless consensus protocol proposed by Ben-Or, Kelmer, and Rabin (BKR) [20]. BKR is the core component of several recent works [31, 57, 77]. We chose BKR also because it allows us to easily distribute Ethereum blocks to each node, in such a way that the total order given by the consensus protocol is the same as given by Ethereum. In general, BKR requires each node to make a proposal in each consensus instance (i.e., for each sequence number). The final output of each instance is a subset of all proposals (ranging from $n - f$ to n proposals). To strictly follow the order of Ethereum workloads, we further restrict the number of proposals output by BKR to n, ensuring that no proposal is abandoned.

We assigned block B_h to a node i for speculation if $i = h\%n$, where n is the number of nodes[1]. We also configured a parameter K, where on each node i, if only K blocks are left for replay, node i takes the next block and starts its speculation phase. In principle, the smaller the K, the better the accuracy for speculation, as node i is with a more recent world state. However, to effectively pipeline the two phases of different blocks, K should not be too small. In our experiments, we set K to 2, as on average the time it takes to speculatively execute a block is less than twice the time it takes to replay a block (see §6.2). Before each test there is a warm-up phase, which loads the world state and all blocks after that state into memory. Figure 7 demonstrate how we integrate Vegeta into the consensus protocol. Our code is available at https://github.com/Decentralized-Computing-Lab/Vegeta.

[1]Note that in real-world cases, nodes continuously receive new transactions and batch them into blocks. No pre-defined order exists.

Table 2: Ethereum datasets used in our evaluation.

Tag	Date	Block range (height)	Tx count	Longest chain	Ratio
S_1	3/7/2023	16774645-16779644	739863	88136	8.39
S_2	3/7/2023	16774645-16777644	436115	52862	8.25
S_3	3/7/2023	16774645-16774745	15129	1779	8.50
S_4	11/17/2023-11/18/2023	18581726-18586725	747651	89961	8.31

6 Evaluation

We evaluated Vegeta under the real-world Ethereum workloads. We compared Vegeta with serial execution and a deterministic optimistic concurrency control protocol, AriaFB, which was derived from Aria [56]. AriaFB is a typical way of instantiating the order-execute framework. We adopted the same execution and commit phases used in Aria, and also integrated a fallback strategy (i.e., our replay phase) to get AriaFB, as vanilla Aria cannot handle high abort rates (see §6.2).

Testbed. We conducted experiments on Amazon EC2 platform. Our testbed consists of up to 10 m6i.4xlarge instances, with 3.5 GHz Intel Xeon Ice Lake 8375C processor of 16 vCPUs, 64 GB RAM, running Ubuntu 20.04 LTS. Every node has downloaded Ethereum workloads.

Datasets. Our evaluation was conducted mainly with the Ethereum datasets shown in Table 2. S_1 contains 5,000 blocks and is used to evaluate the performance in single-node mode and validate the correctness of Vegeta. S_2 contains 3,000 blocks and is used for multi-node test. S_3 contains 101 blocks and is used for time-consumption analysis of each phase (Figure 8 in §6.2). S_4 contains 5,000 more recent blocks and is used to evaluate the performance in both single-node and multi-node modes. In Table 2, tx count refers to the total number of transactions of all blocks. For each block, we first calculate the length of its longest dependency chain, i.e., the maximum number of transactions that have accessed a given key. Then, we sum the lengths of the longest chains of all blocks and get the longest chain in Table 2. Finally, the ratio equals to the tx count divided by the longest chain. The ratio approximately represents the maximum speedup Vegeta can achieve for a specific workload, if we assume every transaction has equal execution time. For simplicity, we did not change the block structure of Ethereum, but in practice we may only treat the workload as a transaction stream.

6.1 Correctness Validation

The correctness of Vegeta has been extensively validated as a by-product of our evaluation. This is because Ethereum

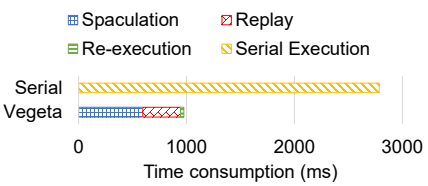


Figure 8: Time consumption of each phase.

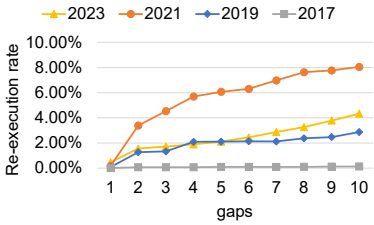


Figure 9: Re-execution rates.

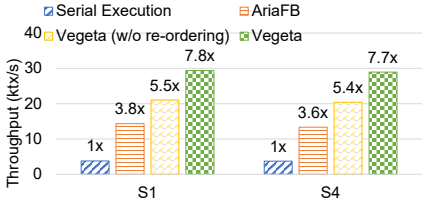


Figure 10: Throughput and speedups in single-node test.

maintains the world state as an MPT. Two states are identical if and only if the values of their MPT roots are equal. Every block carries a value of the state's MPT root after all the transactions in the block are executed. Although the execution result of Vegeta differs from the original state of Ethereum due to the re-ordering rule we adopted in the speculation phase, we can still perform the execution repeatedly (i.e., the speculation and replay phases) with the same series of blocks and validate whether their MPT root values are equal. We have managed to process a total of 5,000 blocks (S_1) carrying 739,863 transactions for 100 times, always resulting in a matching value of the MPT root in every block.

6.2 Single-Node Performance

We demonstrate the performance of Vegeta and other protocols when they were deployed on a single node.

Time consumption of each part. We first tested 101 blocks carrying 15,129 transactions (S_3 in Table 2) to measure the time consumption of each phase in Vegeta. Figure 8 depicts the results. In this measurement, the speculation phase was based on the latest Ethereum world state, i.e., block B_h is speculatively executed based on the world state of block B_{h-1}. The speculation yields 1.47% re-execution rate. The cost of re-execution is insignificant compared to other parts. The speculation phase consumed more time than the replay phase because of the overhead of ordering transactions and building the DAG for capturing dependencies. It can also be observed that Vegeta spent much less time in executing all phases compared to serial execution. Note that in real systems the speculation phase is executed only by a single node and can be further overlapped with the replay phase of some previous blocks.

Re-execution rate. We conducted evaluation in the re-execution rate by performing the speculation phase based on different world states. We then counted how many transactions are re-executed in the relay phase and accordingly calculated the rate. The results are shown in Figure 9. We performed the similar experiment based on the Ethereum data ranging from 2017 to 2023. X-axis indicates the gap of block heights between the speculation and the world state it is based on. For instance, 0 means the speculation results of block B_h is based on the world state of block B_{h-1}, while 9 means the speculation results of block B_h is based on the world state of B_{h-10}. With more recent world states the speculation can obtain more accurate dependencies among transactions, thus reducing the re-execution rate. Even with the most recent state, the re-execution rate is still a non-zero value. This is because in the speculation phase Vegeta executes transactions in a fully parallel manner. A concrete example is actually depicted in Figure 1, where the check-balance operation (step 2) for APEFI token failed during speculation but succeeded in the relay phase, because another transaction (within the same block) transferred a sufficient number of tokens to the user's address. The re-execution rate is relatively small except 2021, presumably due to the explosion of DeFi and NFT applications.

Speedup. We compared Vegeta with AriaFB and serial execution to demonstrate the effectiveness of our approach. In the speculate-order-replay framework, the burden of speculation can be evenly distributed to every node and amortized by pipelining, while the replay (and re-execution) phase must be performed by every node. Therefore, in the single-node test, we only measured the time it takes to perform the replay phase and compared it with that of other approaches. The speedup can be considered the upper bound Vegeta can achieve for the given Ethereum workloads.

The evaluation was conducted on the datasets S_1 and S_4 (in Table 2), each of which contains 5,000 blocks. The results are depicted in Figure 10. The performance bottleneck of Vegeta for Ethereum datasets mainly stems from the longest dependency chain, which was also discussed in [35]. Transactions in the same dependency chain must be executed in serial order. We also observe that the longest dependency chain is generated by Wrapped Ethereum (WETH) decentralized application [10], which swaps ethers (cryptocurrency in Ethereum) between layer-1 (native coin) and layer-2 (token) for facilitating trade with ERC-20 tokens. The ratio shown in Table 2 also reflects an upper limit to some extent (8.39× versus 7.8× and 7.7×).

AriaFB treated each block as a batch and first executed all transactions in parallel. If any transaction had a forward dependency (according to Rule 2), the transaction was aborted. We found that without fallback, aborted transactions in vanilla Aria kept accumulating and eventually led to out-of-memory error. We thus integrated our replay phase into vanilla Aria

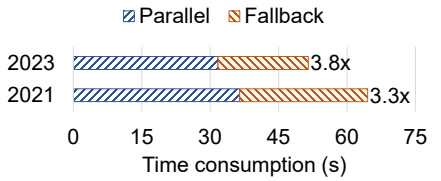


Figure 11: Time consumption and speedups of AriaFB.

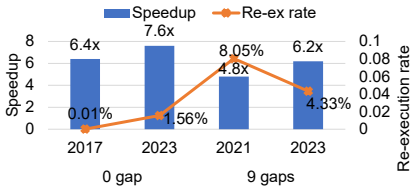


Figure 12: Speedups with different re-execution rates.

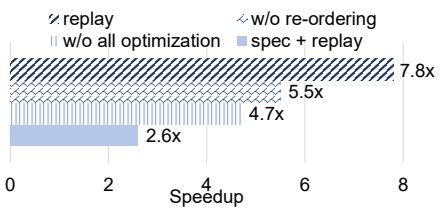


Figure 13: Ablation study of Vegeta.

and obtained AriaFB[2]. For datasets S_1 and S_4, respectively, 40.2% and 42.3% transactions were aborted and further re-sorted to the fallback for re-execution. Compared to AriaFB, the speedup gain of Vegeta largely comes from the speculation that wisely orders transactions and groups them into batches, effectively reducing abort (re-execution) rate. We further measured the execution time of each part under the datasets of 2021 and 2023 (as shown in Figure 11). The abort rate of 2023 dataset is 40.2%, and the corresponding speedup is 3.8×, while the abort rate of 2021 dataset is 52.1%, so the speedup decreased to 3.3×.

To assess the impact of re-execution rates, we also measured the speedups with the datasets from different years (thus leading to different re-execution rates), as shown in Figure 12. The datasets from 2017 and 2023 exhibit lower re-execution rates, resulting in higher speedups. However, the longest dependency chain problem in the 2017 dataset has a greater impact on the degree of parallelism, leading to a lower speedup compared to the 2023 dataset. To evaluate the performance with even higher re-execution rates, we then purposely performed the speculation phase with a gap of 9 blocks. The speedups were negatively affected.

Ablation study of Vegeta. We evaluated the influence of each optimization in Vegeta. The results are shown in Figure 13. To assess the impact of re-ordering in the speculation phase, we purposely disabled this approach (i.e., sorting and selecting dependency chains, Line 4-9 in Algorithm 1) and preserved the original order in each Ethereum block. The speedup (without re-ordering) decreased to 5.5×, as transactions in some longer dependency chains may start execution later during the relay phase, highlighting that transaction order significantly impacts performance improvement. We emphasize that in real-world deployment, there is no concept of re-ordering, as proposers can freely choose in what order transactions are executed. We then stepped further and removed the optimization for re-execution, which is described in Algorithm 3. The speedup decreased to 4.7×. Finally, we evaluated the performance of Vegeta when both phases (i.e., speculation and replay) are performed after consensus by every node. It can be regarded as a deterministic concurrency control protocol based on the order-execute framework. The speedup decreased to 2.6×. This result demonstrates the benefit of distributing speculation across all nodes.

[2]As also discussed in [56], any deterministic concurrency control can potentially be used.

Table 3: Re-execution rates in multi-node test.

	S_2	S_4
4 nodes	1.66%	1.58%
10 nodes	1.89%	1.82%

Multi-core scalability. To evaluate the multi-core scalability, we conducted experiment with different configurations of 2, 4, 8, 16 and 32 cores. The results are shown in Figure 14. The speedup of Vegeta grows linearly from 2 to 8 cores. As the number of cores continues to increase, this trend diminishes.

6.3 Multi-Node Performance

When it came to multi-node deployment, we integrated transaction execution with a consensus module. We set the number of nodes n to 4 and 10 in this experiment, which respectively tolerates 1 and 3 Byzantine nodes. We did not conduct even larger-scale experiments because the Ethereum workloads were generated sequentially. With larger n, there will be a larger gap between speculation and replay, which are not realistic in practice. Nonetheless, as Vegeta aims to provide scalability in the speculation phase, we expect Vegeta will perform even better when n becomes larger. The evaluation was conducted on the datasets S_2 and S_4 in Table 2.

As with a leaderless consensus protocol, we evenly assigned blocks to every node, i.e., node i is responsible for speculatively executing block B_j (in Vegeta) and proposing B_j in consensus if $j\%n = i$. When the consensus layer outputs, n blocks are delivered to the upper layer for execution. These n blocks are totally ordered based on proposers' IDs.

Re-execution rate. As already shown in Figure 9, a more recent world state can provide a higher precision for dependency analysis. Due to this reason, a node would not perform speculation of a new block until its remaining blocks ready for replay is equal to or less than K. We set K to 2 in this experiment because as shown in Figure 8, the time taken by speculation is less than twice the replay time. Table 3 shows the re-execution rate.

Leaderless vs. leader-based protocols. We first compared the performance of Vegeta integrated with BKR (a leaderless protocol) with that integrated with a leader-based protocol.

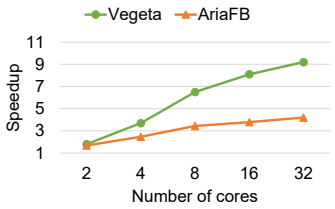


Figure 14: multi-core scalability.

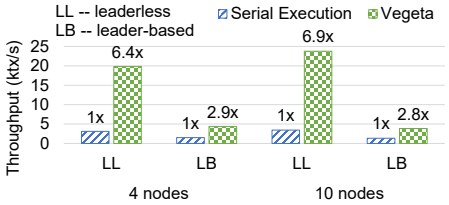


Figure 15: Throughput of leaderless and leader-based protocols.

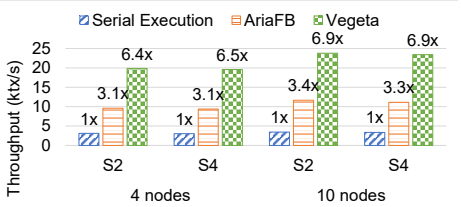


Figure 16: Throughput and speedups in multi-node test.

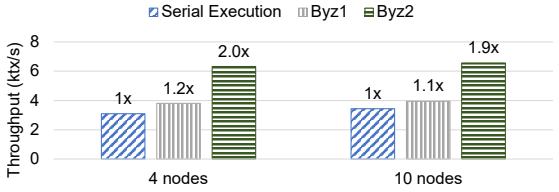


Figure 17: Throughput under Byzantine faults.

For the latter case, our evaluation chose PBFT [25] as the ordering module. The single leader performs the speculation phase, while every node replays the block after consensus. The evaluation was conducted on the datasets S_2 in Table 2, and the results are shown in Figure 15. Because the single leader took the heaviest burden of computation, the leader node throttled Vegeta's performance.

Speedup. The results are shown in Figure 16. Being affected by consensus, the throughput of all protocols decreased compared to Figure 10. Because of the competition between speculation and replay for computing resources, the performance of Vegeta is further affected. Although for 10-node case the re-execution rate is slightly higher than that for 4-node case (Table 3), the $n = 10$ throughput raised because with larger n, leaderless consensus can commit more proposals in each instance. The performance of serial execution and AriaFB exhibited a similar trend due to the same reason.

6.4 Performance under Byzantine Faults

As blockchain systems target Byzantine faults, in this experiment we evaluate their impact on performance by configuring f nodes to exhibit two typical Byzantine behaviors. Specifically in our design, a Byzantine node can purposely generate incorrect dependencies and/or read/write sets in the speculation phase. Such an anomaly cannot affect the safety property of Vegeta but may badly affect its performance.

The results are shown in Figure 17. In the first scenario (Byz1), f Byzantine nodes provided a serial order and empty read/write sets for each block proposal, which forced other correct nodes to sequentially execute each transaction twice (one for replay, and the other for re-execution). Note that the proposals of other correct nodes were still correctly speculated. The performance gains coming from correct nodes are almost eliminated by Byzantine nodes. Although in 10-node case Vegeta had higher speedup compared to 4-node case (Figure 16), it also suffered more badly than 4-node case due to the higher proportion of Byzantine nodes (1/4 versus 3/10). We argue that the impact of this typical malicious behavior may be simply mitigated by, e.g., sequentially executing each transaction only once, as the speculation already provided serial order.

The second scenario (Byz2) is a bit more subtle, where Byzantine nodes provided correct dependencies but empty read/write sets, which still led to 100% re-execution rate for their proposals. The throughput of Vegeta still significantly decreased. For the block proposals made by Byzantine nodes, all the transactions were first executed (efficiently) in the replay phase and then re-executed in serial order. Note that Byzantine behaviors may also badly affect the performance of AriaFB, e.g., by deliberately orchestrating a sequence of transactions that causes frequent aborts.

7 Conclusion

Towards efficient transaction processing in blockchains, we presented the speculate-order-replay framework. Our framework is tailored to leaderless consensus, which brings both challenges for deterministic parallelism and opportunities for boosting performance. Our framework allows each node to independently pre-execute transactions and analyze their dependencies in the speculation phase, in order to provide information that enables efficient parallel execution in the replay phase. We instantiated our framework through Vegeta, a transaction-level concurrency control protocol. Experimental evaluation under Ethereum workload showed promising performance compared to other parallel processing schemes.

Acknowledgments

We are very grateful to our shepherd, Xiaoqi Chen, and the anonymous reviewers for their insightful feedback. This work was supported by the National Natural Science Foundation of China (grant no. 62372293) and the Shanghai Action Plan for Science, Technology and Innovation (grant no. 24BC3201300). Shengyun Liu is the corresponding author.

References

[1] Ape finance. https://ape.fi/swap.

[2] Ethereum example for smart contract innocation. https://etherscan.io/tx/0xf195576ee9915449726ccdad80d389cb6a1853cea685368ea98da2b79e1b57c9.

[3] Ethereum is a dark forest. https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest.

[4] Ethereum virtual machine. https://ethereum.org/en/developers/docs/evm/.

[5] Ethereum whitepaper. https://ethereum.org/en/whitepaper/.

[6] Proposer/block builder separation. https://ethresear.ch/t/proposer-block-builder-separation-friendly-fee-market-designs/9725.

[7] Sui documentation: Validator committee. https://docs.sui.io/guides/operator/validator-committee.

[8] Uniswap. https://uniswap.org/.

[9] Usd coin. https://www.circle.com/en/usdc.

[10] Wrapped ethereum. https://101blockchains.com/wrapped-ethereum/.

[11] Zero-knowledge rollups. https://ethereum.org/en/developers/docs/scaling/zk-rollups/.

[12] Daniel J. Abadi and Jose M. Faleiro. An overview of deterministic database systems. *Commun. ACM*, 61(9):78–88, aug 2018.

[13] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 23–34, New York, NY, USA, 1995. Association for Computing Machinery.

[14] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 83–92, 2019.

[16] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, and Roger Wattenhofer. Fnf-bft: Exploring performance limits of BFT protocols. *CoRR*, abs/2009.02235, 2020.

[17] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds, Inc. Technical Report SWIRLDS-TR-2016*, 1, 2016.

[18] C. Basile, Z. Kalbarczyk, and R.K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 17(5):448–465, 2006.

[19] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 173–184, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 183–192, New York, NY, USA, 1994. Association for Computing Machinery.

[21] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[22] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[23] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. Morty: Scaling concurrency control with re-execution. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 687–702, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[25] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[26] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 570–587, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding i/o latency with pre-execution prefetching for parallel applications. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–10, 2008.

[28] T. Crain, C. Natoli, and V. Gramoli. Red belly: A secure, fair and scalable open blockchain. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1501–1518, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[29] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 303–312, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Sisi Duan, Michael K. Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2028–2041, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[33] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, jan 2017.

[34] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 15–26, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2315–2326, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 232–244, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Bullshark: DAG BFT protocols made practical. *CoRR*, abs/2201.05677, 2022.

[39] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, 2019.

[40] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, jun 1996.

[41] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 803–818, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[43] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 164–173, 2000.

[44] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. A high performance concurrency protocol for smart contracts of permissioned blockchain. *IEEE Transactions on Knowledge and Data Engineering*, 34(11):5070–5083, 2022.

[45] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, aug 2008.

[46] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 237–250, Berkeley, CA, USA, 2012. USENIX Association.

[47] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[48] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584, 2004.

[49] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2007.

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

[51] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[52] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A decentralized blockchain with high throughput and fast confirmation. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 515–528. USENIX Association, July 2020.

[53] Haoran Lin, Yajin Zhou, and Lei Wu. Operation-level concurrent transaction execution for blockchains, 2022.

[54] Jian Liu, Peilun Li, Raymond Cheng, N. Asokan, and Dawn Song. Parallel and asynchronous smart contract execution. *IEEE Trans. Parallel Distrib. Syst.*, 33(5):1097–1108, may 2022.

[55] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. Flexible advancement in asynchronous bft consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 264–280, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.

[57] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[58] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[59] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 35–48, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Afonso Oliveira, Henrique Moniz, and Rodrigo Rodrigues. Alea-bft: Practical asynchronous byzantine fault tolerance. *CoRR*, abs/2202.02071, 2022.

[61] Shujian Qian and Ashvin Goel. Massively parallel Multi-Versioned transaction processing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 765–781, Santa Clara, CA, July 2024. USENIX Association.

[62] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 180–194, New York, NY, USA, 2021. Association for Computing Machinery.

[63] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214, 2022.

[64] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2plsf: Two-phase locking with starvation-freedom. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 39–51, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[66] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 105–122, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: A software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 69–84, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 17–33, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1150–1160. VLDB Endowment, 2007.

[70] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[71] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), Sep. 1997.

[72] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1–2):70–80, sep 2010.

[73] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[74] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 59–72, New York, NY, USA, 2007. Association for Computing Machinery.

[75] Bo Wang, Shengyun Liu, He Dong, Xiangzhe Wang, Wenbo Xu, Jingjing Zhang, Ping Zhong, and Yiming Zhang. Bandle: Asynchronous state machine replication made efficient. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 265–280, New York, NY, USA, 2024. Association for Computing Machinery.

[76] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, jul 2021.

[77] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, Renton, WA, April 2022. USENIX Association.

[78] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

[79] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with rdma networks. *Proc. VLDB Endow.*, 12(11):1637–1650, jul 2019.

[80] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649. USENIX Association, November 2020.

A Correctness proof of Vegeta

We prove that the execution results on each node are equivalent to the same serial order. Since all information (e.g., transactions and their dependencies) leveraged by the replay phase is agreed upon using consensus and each node executes transactions in a batch-by-batch manner, we only need to prove that each batch produces the same execution results. Note that within a batch, transactions are ordered by their indices. We give the following three lemmas.

Lemma 1. *If nodes i and j execute batch b according to the same world state and tx is re-executed on node i, then tx is re-executed on node j.*

Proof. Within b, every transaction is executed based on the same world state and no one updates any key until they all complete execution. If tx is re-executed on node i, then tx must change its read/write sets (Line 8 in Algorithm 2) or read/write a new key $key' \in all_keys$ (Line 5 in Algorithm 3). Then obviously node j should also re-execute tx because the read/write sets of each transaction are obtained during speculation and agreed upon by consensus. □

With Lemma 1, in the following we only consider the transactions that are not re-executed. We discuss the cases without and with an improved replay strategy, respectively.

Lemma 2. *(With a simple re-execution strategy in Algorithm 2) The execution results of each batch are equivalent to the same serial order on every node.*

Proof. For any two transactions tx and tx' in the same batch, they should not have WAW dependency, nor should they have both WAR and RAW dependencies (Line 34 in Algorithm 1). Without loss of generality, we assume the serial order of transactions in the batch is: $\cdots \rightarrow tx' \rightarrow \cdots \rightarrow tx \rightarrow \cdots$. We then discuss three cases:
Case 1: If they are non-conflicting, they are serializable. And we determine their order based on the order of their indices.
Case 2: If tx WAR-depends on tx', then since the result of execution will only be committed after all transactions in the batch finish execution, tx' cannot read the value written by tx, they are serialized as tx' happens before tx, the assumed serial order is satisfied.
Case 3: If tx RAW-depends on tx', their execution is equivalent to tx happens before tx' (i.e., re-ordering), so the serialized order can be changed to $\cdots \rightarrow tx \rightarrow \cdots \rightarrow tx' \rightarrow \cdots$. Also note that tx in this case must have no WAR dependency with any other transaction, so no dependency circle will be generated in the same batch.

In all cases, the execution results of all the transactions within each batch are equivalent to a certain serial order. It is also trivial to see that every node proceeds based on the same world state and consensus-provided information. So their execution results should be equivalent, □

Lemma 3. *(With an improved strategy in Algorithm 3) The execution results of each batch are equivalent to the same serial order on every node.*

Proof. With the proof of Lemma 2, we only need to discuss the following three cases.
Case 1: Neither tx nor tx' reads a new key that is (newly) written by some transaction. In this case, both tx and tx' are committed by following the serial order extracted from DAG (Line 14 in Algorithm 3). Even if tx and tx' newly write the same key key', there is no RAW-dependency between them, so their execution results are still equivalent to the serial order.
Case 2: tx reads a new key key' that is also (newly) written by some transaction, while tx' does not. In this case, tx' gets committed before tx (Line 14), while tx is executed again and committed (Line 16). Even if tx' also writes key', tx can read the update of key' if necessary. Anyway, tx' is executed before tx in the serial order.
Case 3: both tx and tx' read some new keys, say key' and key'', that are also written by some transactions. In this case, both tx and tx' are executed (sequentially) again and committed. Without loss of generality, tx is executed before tx' in the serial order. □

With Lemma 1, every node re-executes the same set of transactions sequentially. With Lemma 2, Lemma 3 and every node executes batches one after another, the execution results of all transactions are equivalent to the same serial order.

Shoal++: High Throughput DAG BFT Can Be Fast and Robust!

Balaji Arun, *Aptos Labs*
Zekun Li, *Aptos Labs*
Florian Suri-Payer*, *Cornell University*
Sourav Das*, *UIUC*
Alexander Spiegelman, *Aptos Labs*

Abstract

Today's practical partially synchronous Byzantine Fault Tolerant consensus protocols trade off low latency and high throughput. On the one end, traditional BFT protocols such as PBFT and its derivatives optimize for latency. They require, in fault-free executions, only 3 message delays to commit, the optimum for BFT consensus. However, this class of protocols typically relies on a single leader, hampering throughput scalability. On the other end, a new class of so-called DAG-BFT protocols demonstrates how to achieve highly scalable throughput by separating data dissemination from consensus, and using every replica as proposer. Unfortunately, existing DAG-BFT protocols pay a steep latency premium, requiring on average 10.5 message delays to commit transactions.

This work aims to soften this tension, and proposes Shoal++, a novel DAG-based BFT consensus system that offers the throughput of DAGs while reducing end-to-end consensus commit latency to an average of 4.5 message delays. Our empirical findings are encouraging, showing that Shoal++ achieves throughput comparable to state-of-the-art DAG BFT solutions while reducing latency by up to 60%, even under less favorable network and failure conditions.

1 Introduction

This paper presents Shoal++, a novel partially synchronous DAG-based Byzantine Fault Tolerant (BFT) consensus protocol that matches the throughput of contemporary DAG-BFT protocols while significantly reducing latency, narrowing the gap with the theoretical optimum.

BFT consensus offers the attractive abstraction of a single, always available server, that remains correct even as a subset of replicas may fail arbitrarily. Fueled by recent advances to simplicity and practicality [15, 19, 35, 47], interest and adoption of BFT protocols is skyrocketing: it is at the core of aspiring multi-national projects such as the digital Euro [2], confidential data sharing frameworks [6] and the growing Web3 industry [1, 5]. Already today, numerous blockchain companies [40–42, 44] are actively deploying BFT protocols [16, 22, 31, 34, 35], supporting millions of daily worldwide users.

Following the seminal PBFT [15] protocol, most traditional BFT consensus protocols [24, 28, 47] operate in a primary-backup like regime in which a single, dedicated leader replica proposes an ordering of commands, and follower replicas vote to ensure consistency and durability.

*The work was done while the authors were interns at Aptos Labs.

As demonstrated by PBFT, this approach can enjoy excellent latency in fault-free executions, requiring only three message delays (md) to order a leader's proposal, the optimum achievable for BFT protocols. A dedicated leader, however, poses not only a single point of failure, but constrains the achievable throughput to the processing and networking bandwidth available to a single server. To improve scalability, and meet the high throughput demands of modern applications, recent proposals emphasize maximizing resource utilization and balancing load across *all* replicas.

Initial approaches propose multi-log frameworks [37] that partition the request space, and operate multiple consensus instances in parallel (each led by a different replica) whose logs are carefully intertwined. More recent designs take scalability a notch further by entirely separating data dissemination and consensus. Pioneered by Aleph [21] and DAG-Rider [26], and made practical by Narwahl [19] a new class of so-called DAG-BFT protocols have risen to popularity.

DAG-BFT protocols employ a structured, highly parallel, asynchronously responsive data dissemination layer as a backbone, and layer consensus atop. Data dissemination proceeds through a series of *rounds* in which each replica proposes new transaction batches via reliable broadcast (RB); each proposal (DAG *node*) references a subset of preceding proposals (DAG *edges*), resulting in a Directed Acyclic Graph (DAG) of temporally (or *causally*) ordered proposals. The consensus layer reaches agreement on prefixes of the DAG, and derives an ordering for all causally referenced proposals.

State-of-the-art DAG protocols [34, 35] avoid running an explicit consensus protocol, and instead embed the consensus process onto the DAG directly: they ensure that all replicas converge on the same view of the DAG, and interpret it's structure locally to implicitly determine commitment. Consensus simulates leaders by assigning pre-determined "fixpoint" nodes in the DAG (henceforth *anchors*) and confirms agreement of an anchor by waiting until it has gathered sufficient references in future DAG rounds that guarantee its durability.

Upon committing an anchor node, all of its causally referenced proposals are implicitly committed and ordered. Such design is appealingly simple – it requires no additional communication for consensus and avoids the notoriously complex view change logic inherent to traditional BFT protocols. It is also very resource efficient – each replica acts as a proposer, thus allowing high throughput with a large number of replicas. As a result, DAG-BFT has prompted swift adoption by several large blockchain systems [42, 44].

However, today's DAG protocols fall short in terms

of latency. They require several rounds, each consisting of reliable broadcasts to *certify* DAG nodes, to commit a transaction. Bullshark [35] for instance requires, even in fault-free cases, up to 4 rounds to commit a proposal (12md in expectation, §3.2); recent advances made by Shoal [34] reduce it to 3 rounds (10.5md expected, §3.2). A far cry from the 3md possible by traditional BFT protocols!

This work proposes Shoal++: a novel DAG-based BFT system that achieves near-optimal latency while maintaining the high throughput and robustness of the state-of-the-art certified DAG construction.

We begin by analyzing sources of end-to-end (e2e) latency in DAG protocols (§3.2); at a high level, it can be broken down into three stages. (i) DAG proposals happen only at regular intervals (rounds), and transactions that narrowly miss inclusion (e.g. arrive just after a node is formed) must wait for the next round. We refer to the time a transaction must wait to be included in a DAG proposal as **Queuing Latency**. (ii) In order to be committed, a proposed DAG node must be referenced by a committed anchor. Unfortunately, in existing DAG protocols [34, 35], at most one anchor is scheduled per round[1]. This results in additional **Anchoring latency**, the time until a node is referenced by the next committed anchor. (iii) Finally, in existing DAG protocols, an anchor requires *at least* two more DAG rounds to be committed; we refer to this as **Anchor Commit Latency.**

Key ideas. To reduce end-to-end consensus latency, Shoal++ employs three techniques, respectively addressing each latency stage. First, Shoal++ identifies and optimizes an inefficiency in the existing Bullshark commit rule to reduce common case Anchor Commit Latency to only 4md. Next, we configure Shoal++ to dynamically attempt to make *every* node an anchor, thus eliminating Anchoring Latency. Shoal++ builds on Shoal's idea to dynamically re-interpret the anchor assignment at each commit, and introduces small timeouts to help replicas advance rounds in lock-step to ensure that future rounds sufficiently reference anchor candidates. Finally, to minimize Queuing latencies, Shoal++ orchestrates *multiple* staggered DAG instantiations in parallel, and intertwines their resulting logs; this allows transactions that miss inclusion to simply board the new round of the next, staggered DAG.

Put together, these techniques allow Shoal++ to reduce expected end-to-end consenus latency to only 4.5md, shaving off 6md compared to Shoal, the state of the art DAG-BFT.

We've implemented a prototype of Shoal++ and evaluated it empirically against two popular DAG-BFT protocols (Bullshark [35] and Shoal [34]), a concurrent DAG-BFT protocol (Mysticeti [12]), and a traditional BFT protocols (Jolteon [22]). Our results are promising, demonstrating matching throughput of existing DAG BFT solutions while reducing latency by up to 60% over Bullshark in the common case and by up to 10x over Mysticeti in the failure case.

[1]Bullshark [35] schedules an anchor every second round; Shoal [34] improves it to every round.

2 Model

We adopt standard BFT assumptions [15, 47]. We refer to participants that follow the protocol as correct, and those that deviate as faulty (or *Byzantine*). There are a total of $n = 3f+1$ replicas, of which at most f may be faulty at any time. We make no assumptions on the number of faulty clients. We assume the presence of a strong, yet static adversary that may arbitrarily delay messages and coordinate all faulty particpiants, but may not break standard cryptographic primitives such as signatures. We denote signed messages using $\langle m \rangle_\sigma$. We assume that all signatures and Quorums are validated, and henceforth omit explicit mention.

The goal of BFT consensus is to establish a common totally ordered log across all correct replicas. We operate under the partial synchrony model [20]. The system is safe at all times – no two correct replicas ever disagree on a committed prefix of the log. Liveness is not guaranteed during periods of asynchrony, but eventually there arrives a Global Stabilization Time (GST) after which the network (temporarily) behaves synchronously, and progress is guaranteed.

3 Dissecting DAG-BFT

Before we present Shoal++, we provide a brief background on the core mechanisms of DAG-BFT, and analyze the latency breakdown of existing protocols. We implicitly focus on *certified* DAG protocols derived from Narwahl[19]. Although some recent works propose uncertified DAGs [12, 27] in an effort to save latency, this comes at the cost of reduced robustness and introduces new sources of unwanted latency (§3.2).

3.1 DAG core

At the core of DAG-BFT is the round-based certified DAG structure proposed in Narwahl [19]. Replicas proceed through a series of structured rounds, in which replicas continuously propose new nodes containing batches of transactions. Each replica maintains their own local view of a DAG.

A replica adds another replica's proposal as node in its local DAG once the proposal is *certified*. Proposals in round r must reference $n-f$ certified proposals from round $r-1$ to be considered *valid* candidates, and each replica may propose and certify at most one candidate per round. To certify a proposal a replica follows a simple Reliable Broadcast procedure.

1. It broadcasts a signed proposal $P := \langle r, B, edges \rangle_\sigma$ for round r containing (i) a batch B of transactions, and (ii) $n-f$ unique proposal certificates from round $r-1$ (*edges* for short).
2. Upon receiving a valid proposal, a replica checks whether it has already received a different proposal from the same author for round r. If it has, it ignores

the proposal; if it has not, it casts a signed vote $S := \langle r, d = hash(P) \rangle_\sigma$ back to the proposing replica.

3. A replica waits for $n - f$ matching signatures for its proposal P and aggregates them into a certificate $C := \langle r, d, \{S\} \rangle$ that it broadcasts to all replicas.
4. Upon receiving a proposal certificate a replica adds a node to its local DAG.

Figure 1 illustrates a DAG of certified nodes as it appears in a local view of one of the validators. We note that because proposals are certified, no replicas can produce two conflicting nodes for the same (round, replica) position. Thus, all replicas eventually converge on the **same** local view of the DAG.

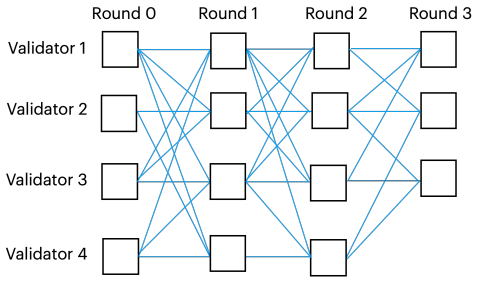


Figure 1: Narwhal's round-based DAG from a validator's local point of view.

3.1.1 Consensus core.

By itself, the DAG is no more than a scalable and robust mempool. It continues to grow at the pace of the network (i.e. *responsively*), and, by virtue of being certified, ensures that all disseminated transaction batches are reliably available. A consensus layer is needed to establish agreement. It can be modularly layered by running an **external** consensus blackbox that commits views of the DAG (e.g., Narwhal+Hotstuff [19]), or logically **embedded** into the existing DAG messages without the need for additional messages (e.g., Tusk [19], Bullshark [35], or Shoal [34]). The rest of the paper focuses on embedded DAG-BFT protocols as they are more efficient.

Embedded consensus. At a high level, consensus is projected onto the DAG structure by designating specific DAG nodes as fixpoints (henceforth *anchors*) that simulate a leader, and interpreting DAG edges as votes for a leader. Upon committing an anchor, all of its causal history can be implicitly committed as well; an ordering can be derived using any deterministic function, e.g. using a topological sort.

Anchor nodes are chosen deterministically in advance (much like leaders in traditional BFT protocols) and placed at regular intervals. Bullshark [35], for instance, uses a round-robin scheme to select anchor candidates and places an anchor candidate per every other round of the DAG. Figure 2 illustrates the anchor commit logic in Bullshark.

An anchor in round r becomes committed as soon as $f+1$ nodes in a round $r' > r$ link to it (**Direct Commit Rule**). The

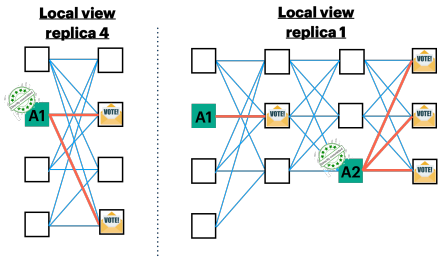


Figure 2: Bullshark Commit example. Replica 4 observes $f+1=2$ votes for anchor A1, and directly commits it. Replica 1 observes only one vote ($< f+1$), and cannot directly commit A1; it commits A1 indirectly upon committing A2 and its causal history, ordering A1 before A2.

safety intuition is straightforward: (i) all nodes are certified, and thus every anchor is equivocation-free, and (ii) all nodes in rounds $r'' > r'$ must include an edge to at least one of the nodes that link to the anchor, and thus the anchor is durable.

In particular, every future committed anchor must link to all previously committed anchors, thereby establishing a consistent total order amongst anchors. Upon committing an anchor, a replica recursively traverses the anchor's causal history, and first commits all preceding anchors (**Indirect Commit Rule**). Conversely, if an anchor candidate is *not* in a committed anchors causal history, then it could never have committed, and it is safe to skip it. Intuitively, the Indirect Commit Rule ensures that, even if a replica does not invoke the Direct Commit rule on some anchors (e.g. because its local DAG view is temporarily out of sync), all correct replicas will agree on the same sequence of committed anchors.

We note that while this design is always safe, it relies on conservative timeouts to ensure liveness (similarly to traditional BFT protocols). Since nodes include only $2f+1$ edges, anchor candidates are not guaranteed to receive the $f+1$ links required to commit. In the worst-case, candidates are repeatedly skipped, thus stalling commit progress.

To reduce the expected latency, Shoal [34] optimizes Bullshark in two ways. First, it leverages the deterministic anchor commit sequence to dynamically re-interpret the anchor schedule upon every commit. Rather than placing anchors every other round, this approach allows for an anchor to be placed every round. Second, it introduces an anchor reputation scheme to attempt to select as candidates only the fastest and well-connected replicas; such candidates are more likely to be included as edges. This allows Shoal to avoid timeouts in all but extremely unlikely scenarios.

3.2 Latency breakdown

We breakdown different sources of latencies present in DAG-protocols, and analyze their respective costs in Bullshark and Shoal. We denote as the e2e consensus latency of a transaction the time between a replica first receives the transaction, and the time that the transaction is ordered. We separate the e2e latency of certified DAG-BFT protocols into three components:

1. **Queuing latency:** the time it takes for a transaction to be proposed in a DAG node.
2. **Anchoring latency:** the time it takes for a non-anchor node's proposal to be "picked up" by an eventually committing anchor.
3. **Anchor Commit latency:** the time it takes to commit an anchor proposal.

Queuing Latency. The DAG advances in a series of synchronized rounds, and each replica may issue at most one proposal (containing a batch of transactions) per round. Each round requires a Reliable Broadcast to certify a proposal, for a total of 3md. Consequently, a transaction that narrowly misses inclusion in the current round proposal must, in the worst-case wait up to 3md before being broadcast; assuming transactions arrive at a uniform rate, the average Queuing Latency is 1.5md.

Anchoring latency. A proposal becomes "anchored" once it is referenced by an eventually committing anchor. Anchoring latency depends on two factors: (i) the frequency of anchor candidates, and (ii) the reliability with which said anchor candidates are committed (i.e. not skipped).[2] Bullshark has anchor candidates in every odd numbered round (i.e., only every second round). Consequently, the best-case inclusion latency of non-anchor proposals in even rounds is 1 round (3md), and 2 rounds (6md) in odd rounds; a best-case average of 4.5md. Shoal increases candidate frequency to every round, and thus improves the inclusion latency of non-anchor proposals to one round for all (3md). The practical reliability of commitment cannot be easily quantified by pen-and-paper, and is best left to empiric analysis as it depends on network configuration and message delivery patterns. Shoal demonstrates that the use of a reputation can aid significantly in ensuring that anchor candidates are reliably included [34].

Anchor Commit latency. In both Bullshark and Shoal, an anchor becomes committed upon either (i) being references by $f+1$ nodes in the next round (Direct Commit Rule), (ii) or being subsumed in the causal history of future committed anchor (Indirect Commit Rule). The Direct Commit Rule requires at least 2 rounds: one to certify the anchor proposal, and one to certify $f+1$ proposals that reference it, for a total of at least 6md. The latency of indirectly committed anchors can be broken down into their own Anchoring Latency to a future anchor, plus the future anchors Commit Latency.

Putting all latency stages together results in a best case average latency of 12md (1.5md + 4.5md + 6md) for Bullshark, and 10.5 for Shoal (1.5md + 3md + 6md).

3.3 A note on uncertified DAGs

A class of recent DAG-based designs [12, 27, 29] proposes to replace DAG nodes with uncertified proposals that are disseminated with best-effort. At first glance, such designs holds promise by reducing latencies across the stack. However, removing certificates makes the DAG brittle: it can introduce a significant amount of unwanted synchronization on the critical path and increase susceptibility to timeout violations [23]. Edges to uncertified, yet locally unavailable proposals must be validated by fetching missing data (at least 2 additional md per round) *before* advancing to avoid losing liveness in the face of Byzantine fabrications or unstable networks. In practice, we find that even occasional message drops can significantly degrade overall latency (§8.3). In contrast, Shoal++ demonstrates that certification is *not* the root-cause of high latencies, and that latency can be significantly reduced without sacrificing robustness.

[2]The proposals of non-anchor candidates are subject to both factors; the proposals of anchors candidates themselves are subject only to the latter.

4 Shoal++ Overview

Shoal++ strives to reduce latency across each of the three latency stages; it employs three different ideas that respectively address Queuing, Anchoring, and Anchor Commit Latency. Together, Shoal++ is able to reduce total e2e expected latency close to an average of 4.5md. We briefly outline each idea in sequence.

Faster anchors. The first observation Shoal++ makes is that the Bullshark Direct Commit rule can be optimized when anchors are referenced by a supermajority of uncertified nodes. Intuitively, a replica may eagerly commit an anchor as soon as $2f+1$ *proposals* that link to an anchor node are observed, as this implicitly guarantees that eventually the Direct Commit Rule is satisfied. Since this is an even stricter condition that may not be fulfilled, Shoal++ (i) retains the existing Direct Commit Rule as backup, and (ii) leverages Shoal's reputation mechanism to reliably trigger the Fast Direct Commit Rule. This allows Shoal++, in practice, to reduce Anchor Commit latency to 4md.

More anchors. Next, Shoal++ takes Shoal's idea of increased anchor frequency one step further, and tries to turn as many nodes into anchors as possible. Doing so requires striking balance. On the one hand, increasing the number of anchors increases the opportunities to avoid incurring the anchoring latency. On the other hand, it introduces also opportunity for more uncommitted anchors, which can stall progress. Adopting the reputation scheme proposed by Shoal can help identify a reliable subset of candidates, but is not enough to streamline progress in practice. Shoal++ (i) slows down round advancement to put the DAG into lockstep and avoid missed anchors, and (ii) dynamically re-interprets the tentative anchor schedule to skip anchors that are no longer needed.

More DAGs. Finally, Shoal++ minimizes Queuing latency by operating not one, but multiple DAGs in parallel. DAGs are staggered using a small offset, allowing waiting transactions that miss a round to quickly be included in the *next* DAG.

This simultaneously improves throughput via better resource utilization: proposals are sent more frequently, batches are smaller and can be sent and processed more efficiently (in a streaming fashion) than less frequent, large batches.

5 Shoal++ In-Depth

Shoal++ uses as starting points the DAG-core of Narwahl [19], and Consensus-core of Bullshark [35] outlined in §3. For the sake of brevity, we do not duplicate the pseudocode that is available in previous works and, in particular, omit a detailed description of the DAG construction (i.e., node broadcasts, transaction submission, node fetching, garbage collection, and so on). We defer to the original papers for an in-depth discussion and correctness proofs.

Algorithm 1 summarizes the DAG API and node structure. We abstract away the node certification process, and the commit logic. PROCESS_NODE and PROCESS_CERTIFIED_NODE respectively involve checking that a node proposal is valid and voting for it, and adding a new confirmed node to the DAG, triggering commit rules. RUN_BULLSHARK performs an instance of the Bullshark commit logic to determine (given a starting point) the first anchor to be ordered (§3.1.1).

Upon committing an anchor, all of its (yet-to-be-ordered) causal history is deterministically ordered as well. Helper functions CAUSAL_HISTORY and GET_ANCHORS aid, respectively, in identifying the causal history of a node (necessary for both commit and ordering logic), and the anchor candidate schedule. Shoal++ adopts and extends Shoal's [34] reputation logic; GET_ANCHORS returns, for each round, a vector of eligible anchors.

Algorithm 1 DAG interface

```
DAG API:
    PROCESS_NODE(v: Node)
    PROCESS_CERTIFIED_NODE(v: Node)
    CAUSAL_HISTORY(v: Node) → Vec<Node>
    RUN_BULLSHARK(anchor: Node) → Node
    GET_ANCHORS(round: int) -> Vec<Node>

struct Node n:
    n.round          ▷ the associated round in the DAG
    n.source         ▷ the replica proposing n
    n.parents        ▷ n−f nodes from round v.round−1
```

Algorithm 2 summarizes the core protocol additions made by Shoal++ in an effort to reduce Anchor Commit, and Anchoring latency. We now discuss both in detail in turn.

5.1 A faster Direct Commit rule

As described in §3.2 it currently takes, in the best case, 6 message delays to commit an anchor using the Bullshark Direct Commit Rule. One DAG round is required to certify the anchor node itself, and a second DAG round is required

Algorithm 2 Core Shoal++ utilities

```
    Local variables:
        dag: DAG                              ▷ Initially empty
        weak_votes: Vec< Vec<int> >           ▷ Initially empty
        round: int                            ▷ Initially 0
        anchors: Vec<Node>                    ▷ Initially empty

 1: procedure NEXT_ORDERED_NODES()
 2:     while true do
 3:         if anchors.IS_EMPTY() then
 4:             round++
 5:             anchors ← dag.GET_ANCHORS(round)
 6:         anchor ← anchors.POP()
 7:         anchor_to_order ← select
 8:                             FAST_COMMIT(anchor)
 9:                             dag.RUN_BULLSHARK(anchor)
10:         if anchor_to_order ≠ anchor then
11:             SKIP_TO(anchor_to_order)
12:         return dag.CAUSAL_HISTORY(anchor_to_order)

13: procedure SKIP_TO(anchor: Node)
14:     round ← anchor.round
15:     anchors ← dag.GET_ANCHORS(round)
16:     anchors.REMOVE(anchor)

17: procedure FAST_COMMIT(n: Node)
18:     while true do
19:         if weak_votes[n.round][n.source] ≥ 2f+1 then
20:             return n

21: upon receiving node n do
22:     r ← n.round
23:     s ← n.source
24:     if n is the first node revived from s in round r then
25:         for each parent ∈ n.parents do
26:             weak_votes[r][parent]++
27:         dag.PROCESS_NODE(n)

28: upon receiving certified node cn do
29:     dag.PROCESS_CERTIFIED_NODE(cn)
```

to collect sufficient votes. A replica can directly commit an anchor upon observing $f+1$ certified nodes that link to the anchor (i.e. include the anchor in their parent set).

Shoal++ makes the following simple observation: if sufficiently many DAG *proposals* link to the anchor, the anchors fate is already set in stone, and we need not wait for round certification to complete. Recall that the certification process consists of three message exchanges, a proposal step, a voting step, and a certificate forwarding step (§3). Shoal++ allows replicas to FAST_COMMIT (Alg. 2) as soon as it observes $2f+1$ proposals that link to an anchor (Fig. 3). To do so, replicas simply additionally keep track of *weak votes* for each round that correspond to the proposals observed. Weak votes are not guaranteed to survive certification. For instance, a faulty proposer may equivocate, and produce a node certificate containing a different proposal that does not link to the anchor. However, out of any $2f+1$ weak-votes, at least $f+1$ must be cast by correct proposers that will never equivocate. Consequently, the presence of $2f+1$ weak-votes guarantees that eventually

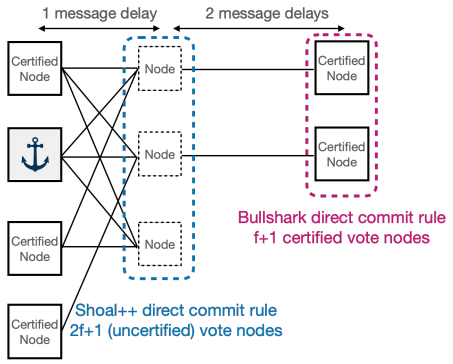


Figure 3: Bullsharks's direct commit rule requires f+1 certified vote nodes, while Shoal++'s direct commit rule requires $2f+1$ (uncertified) node proposals.

$f+1$ certificates linking to the anchor must be formed, thus ensuring safety. Using the Direct Fast Commit rule allows replicas to commit an anchor in only 4md: 3md to certify the anchor, and 1md to broadcast and receive proposals.

We note, however, that this rule places a *stronger* requirement for liveness; f additional proposals are (tentatively) required to link to an anchor. We thus retain the existing Direct Commit rule as backup, and allow replicas to commit using whichever rule is satisfied first. In some cases, it may be faster (and more reliable) to commit using $f+1$ certified nodes. For instance, when the network is unstable or point latencies are asymmetric, the fastest $f+1$ replicas might advance much more quickly than the fastest $2f+1$.

In our empirical evaluation, we find that this case is rare (§8) and that replicas are, most of the time, able take advantage of the Fast Direct Commit rule. Shoal's reputation scheme ensures that eligible anchor candidates are typically well connected, making it exceedingly commonplace for future round proposals to link to an anchor.

5.2 Increasing anchor frequency

In order to reduce Anchoring latency, Shoal++ tries to dynamically designate as many nodes as possible as anchors. Intuitively, if all nodes were anchors that become committed, no node would experience any Anchoring latency.[3] Safely operating more than one anchor per round requires additional orchestration. To ensure a consistently ordered log across all correct replicas, parallel anchors must be committed in a predefined order. One easy way to visualize parallel orchestration is by mapping anchors to fixed slots; much like parallel proposals in traditional BFT protocols such as PBFT [15]. Figure 4 shows a simple example in which there are four anchors in one round (all n nodes are anchors). Each anchor employs its *own* consensus instance that may proceed in parallel, and out of order, but anchors can be committed only in sequence.

[3]We note that such an approach is briefly discussed in the closing remarks of Shoal [34], but is neither implemented nor carefully evaluated for practical considerations.

Each consensus instance operates a one-shot Bullshark instance, augmented with the Fast Direct Commit Rule. We remark that each Bullshark instance operates its own tentative anchor schedule (e.g. the yellow anchor in Fig. 4 is considered the next anchor in the instance started by the red anchor), but terminates as soon as the initial anchor is resolved.

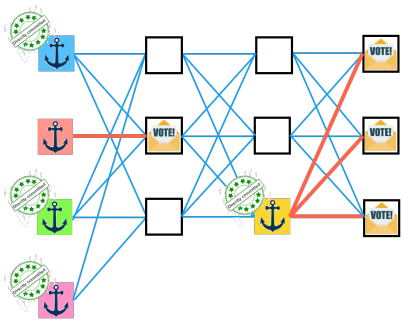


Figure 4: An illustration with four anchors in round 1, using the pre-assigned order: blue, red, green, pink. The blue anchor commits directly; it has $\geq f+1$ references from nodes in round 2. The red anchor is only indirectly committed upon directly committing the yellow anchor in round 4. The green and pink anchors must wait to be ordered until the red anchor is resolved.

While simple, a straightforward instantiation of this design is not robust, and may, in practice negate (or even hurt) any latency benefit. By treating every node as an anchor, progress is, in particular, bottlenecked on the *slowest* anchor. We illustrate such an unwelcome scenario in Figure 4: although the green and pink anchors have sufficient support to commit, they must wait until the preceding red anchor becomes committed or skipped. Unfortunately, the red anchor is slow and does not have any immediate links; it is indirectly committed only two rounds later via the next anchor in its consensus instance (yellow anchor). In worse circumstances, an anchor might never commit. In this case, we must wait for confirmation that the anchor is skipped, and fill its assigned slot with a no-op. Such scenarios are not fringe outliers, but expected by design: since DAG nodes include only $2f+1$ edges, rounds are privy to advancing based on the fastest $2f+1$ nodes, and leaving the remainder behind. Thus, with high probability, *at least* one anchor per round will not receive sufficient direct votes.

Adopting the reputation mechanism proposed by Shoal can help ameliorate, but not remedy the concern. We leverage reputation and modify GET_ANCHORS(r) to limit the set of eligible anchors per round only to those associated with historically fast and well-connected replicas. Unfortunately, we find that in practice leveraging reputation alone is insufficient, and does not reliably enough exclude slow replicas. While past performance is a helpful indicator, network instabilities in geo-distributed systems make it hard to accurately predict *future* performance, and we observe that the set of "slow" replicas changes very dynamically. To improve robustness further, we add two additional techniques that, respectively, try to (i) avoid unnecessarily classifying nodes as slow, and (ii) dynamically skip likely obsolete anchors.

Round Timeouts. Reputation alone is overly harsh in categorizing nodes as slow. Since DAGs advance with the fastest $2f+1$ replicas, to maximize the likelihood of anchors that commit directly, reputation *pessimistically* caps the eligible anchors per round to include only the (estimated) fastest $2f+1$ nodes. (i.e. $\frac{2}{3}$ of total nodes). In reality, however, ineligible nodes may only be fractionally slower, and arrive imminently. To avoid discriminating against such nodes, Shoal++ makes the conscious choice to, in each round, additionally wait for a small timeout beyond the first $2f+1$ nodes observed, and include any momentarily received nodes as additional edges. While this slightly slows down individual rounds, it encourages replicas to operate in lockstep, resulting in a more densely connected DAG. This allows us, in turn, to optimistically increase the set of eligible anchors back to include all n nodes, thus reducing *overall* latency.

We note that timeouts are not necessary for liveness, and are purely optimizing performance. They can thus be configured very aggressively, and are, in practice, negligible compared to the 3 network delays required to certify a node.

Skipping Anchor Candidates. Although round timeouts increase the reliability of the Direct Commit Rule, they cannot guarantee its success. Occasionally, replicas may fail or experience high networking delays, resulting in missing, or unconnected nodes. In such cases, all subsequently pre-ordered anchors must wait until the current anchor is resolved. Recall that an anchor $A1$ in round r is only resolved only upon committing, in its local consensus instance, an anchor $A2$ in a future round $r' > r$: $A1$ is committed if it is present in the causal history of $A2$ (Indirect Commit), or skipped if it is not. In practice, we expect indirectly committed anchors to be rare. Anchors that are merely slow, are likely to be committed thanks to the round timeout. Crashes, in turn, predominantly result in skips. Byzantine abuse can unfortunately not be avoided, and we must rely on the reputation scheme to avoid such anchors; fortunately, such faults are exceedingly rare in most practical deployments.[4]

We note that, even though $A2$ has been committed, it cannot be ordered before all preceding anchors in the pre-defined schedule have resolved their respective consensus instances (i.e. commit or skip). This is undesirable: such anchors are (i) likely already included in the causal history of $A2$, and (ii) may be subject to further latency delays. Unfortunately we cannot avoid this scenario if $A1$ is committed; another replica may have committed it directly, and resolved (and applied) the output of the next scheduled anchor $B1$. We can, however, sidestep the resolution of preceding anchors' Bullshark instances if $A1$ is skipped. For any given consensus instance, all replicas necessarily agree on the order of *all* committed and skipped anchors. Thus, in particular, they agree on the *first* committed anchor (in this case $A2$, since $A1$ is skipped). Let the round of this anchor be r'. Shoal++ opts to *skip* all tentative anchor candidates in rounds $< r'$.

Instead of pre-assigning anchors to fixed slots, Shoal++ considers all but the first anchor in each round *virtual*, and dynamically materializes them as it sees fit. At any given time, there is only a single materialized consensus instance (with an anchor every other round); upon resolving the current instance, Shoal++ re-evaluates where to sensibly place the next anchor. Only then it begins evaluating the associated consensus instance. Since all replicas resolve each instance identically, each locally re-interprets and materializes the same anchor schedule. Consider a slight modification of the example in Figure 4, in which the yellow anchor does **not** extend the red anchor. The blue anchor commits directly, and Shoal++ dynamically materialized the red node as the next anchor to evaluate. The red anchor, however, will be resolved as skipped upon committing the yellow anchor. Shoal++ thus skips evaluating the virtual green and pink anchors, and instead tries to materialize as anchor the first node *after* the yellow anchor.

5.3 Operating multiple DAGs in parallel

Finally, in an effort to minimize Queuing latency, Shoal++ chooses to operate k concurrent DAG instances, and interleave their outputs to construct a single, totally ordered log. Intuitively, staggering multiple DAGs, in a pipline-like manner, allows transactions to be proposed quicker. Instead of waiting (up to 3md) for the next DAG round to begin, a transaction simply joins the next round of the *next DAG*. This reduces the expected Queuing latency from 1.5md to $1.5/k$ md. In practice, we deploy Shoal++ with three DAGs, each offset by one message delay. Since each DAG round requires 3md to certify a node proposal, this ensures that a some DAG's proposal is available every 1md. We find that 3 DAGs offer a good sweetspot between theoretical Queuing latency and batch sizes; additional DAGs offer diminishing returns as proposals carry increasingly smaller batches.

Algorithm 3 summarizes the algorithm. Each DAG instance runs as a black-box, isolated from the others. In order to create a consistent total order, we must interleave their outputs. Each time a DAG commits an anchor, it outputs a new log segment.Shoal++ rotates across DAGs in round-robin fashion, and appends, in each round, exactly one available segment to the log. If, for whatever reason, one DAG commits more frequently, then its excess available segments must wait to be ordered until the other DAGs to make progress. Importantly, the individual DAGs themselves never block, and progress unimpeded. We note that segments also may not (and need not) be of equal size; each segment corresponds to a newly committed anchor, each of which may order varying sizes of causal histories.

Operating additional DAGs incurs additional messages and associated processing overhead. However, we find that in practice, this does not negatively affect performance.

[4] Blockchain systems [42, 44] report observing zero Byzantine faults in over a year of deployment. Reported faults are typically benign crashes.

Algorithm 3 Shoal++

```
Local variables:
    d1,d2,d3: DAG instances                          ▷ Initially empty
1:  loop
2:      for i=1 to 3 do
3:          output di.NEXT_ORDERED_NODES()
```

In fact, throughput increases because smaller batches are proposed more frequently which helps amortize bandwidth and processing, resulting in better resource utilization. In theory, messages from different DAGs can be overlayed in order to share signatures. For instance, a node-proposal of DAG $i \in \{0,1,2\}$ can easily be combined with the certified-node-broadcast message of DAG $(i+1)\%3$.[5] This too, we find not to be worth it in practice as (i) signatures are not the bottleneck, and (ii) message stages have assymetric latencies, and artificially aligning them increases Queuing latency.

5.4 Discussion

The combination of the above techniques allows Shoal++ to, in the common, fault-free case, reduce average e2e latency to only 4.5md: 4md to commit anchors, and 0.5md queuing latency. In the presence of faults or bad network behaviors, additional anchoring and commit latency will be incurred to resolve anchors via the Indirect Commit/Skip Rule.

We note that Shoal++ uses a *linear* star-based communication pattern to ceritfy nodes, while protocols such as PBFT [15] leverages *quadratic* all-to-all communication. If desired, Shoal++ can adopt all-to-all communication, reducing latency by 1md (to 3.5md total) at the cost of increased message complexity. We also remark that although Shoal++ incurs a small additional queuing latency, every replica acts as proposer, and clients need only contact their *local* replica (in absence of faults). Single leader designs, in contrast, require clients to contact possibly remote leaders, introducing additional "queuing" latency. Finally, we note that Shoal++ consumes more resources—such as CPU, memory, and disk space—than Shoal due to the multiple DAGs technique. However, this approach enables Shoal++ to reduce queuing latency and makes better use of the available network bandwidth.

6 Correctness

We now prove the Safety and Liveness of Shoal++. We separately prove each augmentation Shoal++ introduces through a reduction to the Safety and Liveness proofs of Bullshark [35] and Shoal [34]. To formalize the arguments, our proof relies on the following three properties:

Property 1. *For any node n, the call* CAUSAL_HISTORY*(n) returns the same vector of nodes to all replicas.*

[5] In theory, votes can also be overlayed, though typically it is favorable to send them as fast as possible, rather than synchronizing.

Property 2. *All replicas commit the same anchors, and in the same order.*

Property 3. *For any given round r the leader reputation mechanism ensures that* GET_ANCHORS*(r) return the same vector of nodes at all replicas.*

Property 1 holds true for all Narwhal-based DAGs, and follows directly from the fact that all nodes in the DAG are certified; Properties 2 and 3 follow, respectively, from the safety proofs of Bullshark and Shoal.

Fast Direct Commit Rule. Since Shoal++ can commit using both the Fast Direct Commit Rule and the existing (Bullshark) Direct Commit Rule, Liveness follows directly from Bullshark. We prove that the Fast Direct Commit rule is safe:

Lemma 1. *For any anchor a, the procedures* FAST_COMMIT*(a) and* RUN_BULLSHARK*(a) never return contradictory values.*

Proof. By the code, FAST_COMMIT(a) can only return a. Thus we need to prove that if FAST_COMMIT(a) triggers, then RUN_BULLSHARK(a) also returns a. FAST_COMMIT(a) implies that the replica received $2f+1$ (uncertified) node proposals from different replicas that reference a. Since there are at most f faulty replicas, at least $f+1$ of these proposals were broadcasted by correct replicas and will eventually be certified. Since $f+1$ such certified nodes exist, some replicas may directly commit the anchor a via the Bullshark logic. It follows from P.2 that in this case, the RUN_BULLSHARK(a) call returns a to all replicas. □

Multiple Anchors per Round. Liveness follows directly from Bullshark. We show that Shoal++'s dynamic anchor mechanism retains Safety.

Lemma 2. *We assume all correct replicas are initialized with the same state (*round *and* anchors*). Then, all correct replicas' invocations of* NEXT_ORDERED_NODES *(Alg 2) return the same result.*

Proof. By the lemma assumption and Property 3, anchors.POP() returns the same anchor in the first iteration of the while loop to all replicas (Line 6). By Lemma 1, all correct replicas agree on which anchor to order (Line 7). It follows, from the determinism of NEXT_ORDERED_NODES and P.1 that all replicas return the same vector of nodes to be ordered in their first NEXT_ORDERED_NODES call (line 12). In addition, by the determinism of NEXT_ORDERED_NODES, all replicas end up with the same state (*round* and *anchors*) after the calls return. Therefore, by applying the argument above inductively, it is easy to show that all replicas also return the same vector of nodes in all future calls. □

Theorem 1. *The Shoal++ protocol Algorithm 3 satisfies Safety and Liveness.*

The Liveness and Safety proof of each DAG is given above. Since DAG outputs are interleaved deterministically, Safety and Liveness of the composed system follow immediately.

7 Practical Considerations

We briefly discuss some implementation considerations.

Inline data streaming. Narwahl [19] proposes, in addition to the DAG construction, to further decouple, and horizontally scale data dissemination using an additional "worker"-layer. Workers disseminate batches optimistically a-priori, and DAG proposals contain only hash references. This construction is extremely throughput-friendly in the best case, as data is streamed continuously, and consensus carries only metadata. In practice, however, much like with uncertified DAG-BFT protocols [12, 27], other replicas might not have the data behind the hashes (or the references in the case of uncertified DAGs) when the proposal is received and thus additional 2md latency is required for fetching. Shoal++ opts to forgo the worker layer in order to avoid this latency, and chooses to disseminate transaction batches inline with DAG proposals. Operating staggered DAGs allows Shoal++ to nonetheless achieve throughput competitive to that of Narwahl (with a single worker), as smaller batches are sent more frequently, thus amortizing bandwidth close to a streaming design.[6] Finally, we note that inline data streaming simplifies data fetching as it avoids an additional indirection needed to dereference digests.

Efficient fetching. DAG edges in Shoal++ contain only certified nodes, allowing Shoal++ to efficiently fetch missing nodes and resolve inconsistencies across replicas. Replicas whose DAG views are out of sync, may validate and vote on proposals *without* locally observing the proposed node's causal history. This streamlines the certification process, and ensures that the DAG operates at predictable rates. Any missing causal histories can be inferred asynchronously, off the critical path. We remark that this is not possible in uncertified DAGs; to ensure liveness, replicas must fetch all missing data *before* processing a new node. This results in unsteady processing times, and may significantly increase e2e latency. Certified edges, additionally allow us to balance data fetching load: since at least $f+1$ correct replicas must have observed any certified node, replicas missing the node may request it from different replicas to balance load.

Distance-based priority broadcast. We observe that in systems with large n, message broadcasting consumes considerable CPU times. When sending messages in a fixed order (e.g. first to $R1$, then to $R2$, ...) can unintentionally cause replicas to consistently receive messages with delay, making their nodes subject to falling behind in the DAG. To circumvent this issue, Shoal++, periodically records point latencies between replicas, and adjust broadcast send-orders to prioritize farther away replicas. This results in a more balanced message distribution, increasing the homogeneity of node certification pace.

[6] Scaling throughput by employing more workers is typically not worth it in existing BFT deployments, as single-worker ordering throughput in existing protocols already far exceeds execution bandwidth.

8 Evaluation

Our evaluation seeks to answer three questions:

1. **Latency reduction:** How much does Shoal++ reduce Latency compared to state-of-the-art alternatives?
2. **Latency breakdown:** How much does each of the augmentations introduced in Shoal++ contribute to the latency reduction?
3. **Robustness:** How well does Shoal++ tolerate faults?

We implemented Shoal++ into an open-source code base of Aptos, one of the major blockchain projects. Our prototype is written in Rust and utilizes the Tokio [10] asynchronous runtime. It uses BLS [14] implemented over BLS12-381 curves for signatures, RocksDB [9] for persistent storage of consensus data, and Noise [8] for authentication. We disable the execution and ledger storage components of the original blockchain in order to isolate consensus performance. Each data point represents the 50 percentile (median) with error bars representing the 25, and 75th percentiles.

Baselines. We compare Shoal++ against three popular high throughput DAG-BFT consensus protocols, Bullshark [36], Shoal [34], and Mysticeti [12], as well as one traditional low-latency BFT protocol, Jolteon [22]. Bullshark and Shoal represent state-of-the-art *certified* DAG-based protocols, while Mysticeti [12] is a concurrent work to Shoal++ that proposes an *uncertified* design to reduce latency. Jolteon [22] is a state-of-the-art leader-based BFT protocol, which improves Hotstuff [47] latency by 50%. A variant of Jolteon is currently deployed on Aptos [42]. Sui [44] recently replaced Bullshark with a Mysticeti deployment. For an apples-to-apples comparison, we re-implemented Bullshark, Shoal, and Jolteon according to the papers' description in the same codebase as Shoal++. For Mysticeti we run the publicly available source code [7] referred to in the latest version of the paper; its prototype too is written in Rust, using Tokio, but forgoes writing consensus data to persistent storage, making it less production-realistic. To minimize latency, we opt to disable the Narwhal [19] worker layer for both Bullshark and Shoal. However, for a fair throughput comparison with Shoal++, we additionally augment both systems with Shoal++'s proposed parallel-DAG strategy ("Bullshark/Shoal More DAGs", Fig. 5).

Experimental setup. We use the Google Cloud Platform [4] to mimic the deployment of a globally decentralized network. Our testbed consists of 100 replicas spread evenly across 10 regions around the world: we choose two regions in the US (us-west1 and us-east1), two in Europe (europe-west4

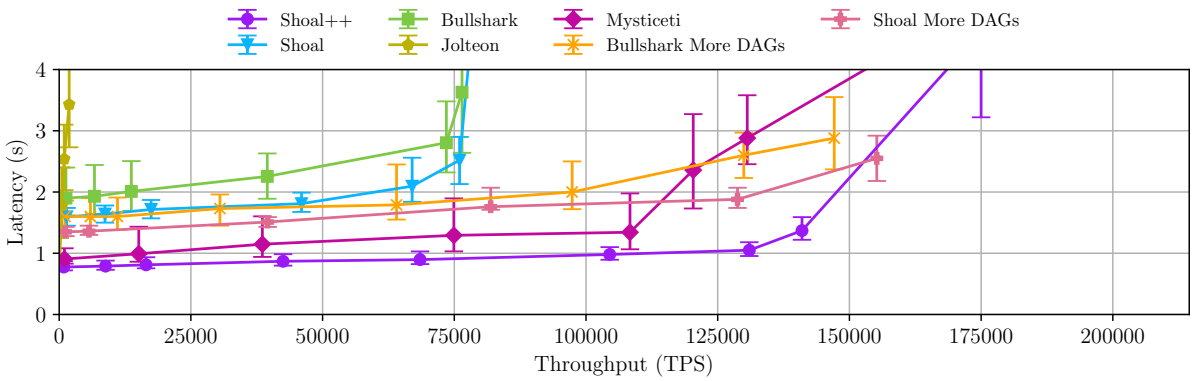


Figure 5: No failures with 100 replicas.

and europe-southwest1), three in Asia (asia-northeast3, asia-southeast1, and asia-south1), and, respectively, one each in South America (southamerica-east1), South Africa (africa-south1) and Australia (australia-southeast1). The round-trip times between these regions range between 25ms and 317ms. We use `n2d-standard-64` virtual machines, containing 64 vCPUs and 256 GB of memory [3]. We provisioned a 2TB network attached disk to each machine to guarantee enough IOPS for persistence. This spec is similar to those used by production blockchains and qualifies as commodity grade.

Clients connect to a single (local) replica and issue a continuous stream of dummy transactions (310 random bytes). As is standard in blockchain deployments, we measure consensus latency as the time between when a transaction first arrives at a replica, and when the transaction is ordered and ready to be executed. We use a batch size of 500 transactions across all systems. In all DAG-based systems, each node proposal contains one batch; Jolteon proposals may contain up to 100 batches. We configure Jolteon to use a 1.5s timeout for view changes (the standard in existing production systems [42]), and use a round timeout of 600ms for Bullshark, Shoal and Shoal++; Mysticeti by default uses a 1s round timeout. We remark that timeouts in Bullshark, Shoal and Shoal++ serve different functions. Bullshark requires them for liveness, while Shoal manages to sidestep using timeouts for liveness all but edge cases; Shoal++ follows Shoal, but re-introduces a timeout only for improved performance. We note that the timeout is started at the *beginning* of a round, and thus typically introduces only a small additional wait past observing the first $2f+1$ certified nodes.

8.1 Failure-free case

Figure 5 illustrates throughput and latency under failure-free conditions. Shoal++ is the only system to sustain sub-second latency for throughput of 100k transactions per second (tps).

Jolteon offers low latency (900ms) at low load, but is unable to scale throughput beyond 2100 tps as it is bottlenecked by the leader's network bandwidth. Bullshark and Shoal support significantly higher throughput (up to 75k tps) but incur high latency: 1.9s and 1.45s, respectively, at low load, and 2.4s and 1.7s, respectively at 50k tps. This follows directly from their high average number of required message delays (§3.2). Throughput scalability in both systems is bottlenecked by network utilization.

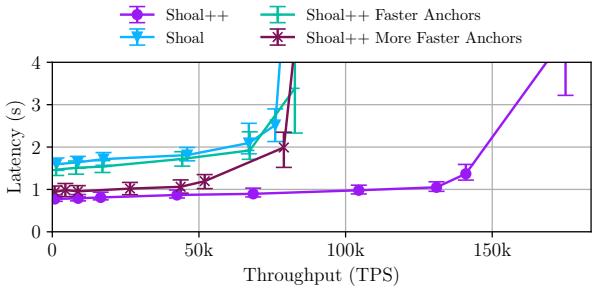


Figure 6: Shoal++ Breakdown. No failures with 100 geo-distributed replicas.

Shoal++ significantly reduces latency (775ms at low load), while simultaneously improving throughput scalability (up to ca. 140k tps). The parallel DAG construction allows Shoal++ to better utilize network and storage resources: rather than broadcasting and processing a large batch once per round, it broadcasts and processes three smaller batches at the same time. Applying the same technique to Bullshark and Shoal ("More DAGs") allows both systems to match the throughput of Shoal++ (while simultaneously improving queuing latency), without incurring the latency penalty of a Narwahl-style worker layer.

Mysticeti matches the throughput of Shoal++, but fails to match its latency, even at moderate loads. At high loads (upwards of 100k tps) Mysticeti notably incurs a signficant latency penalty as replicas may be temporarily out of sync, and must fetch missing data on the critical path of consensus.

8.2 Latency improvement breakdown

Figure 6 isolates the respective latency improvement of each Shoal++ augmentation compared to the baseline, Shoal.

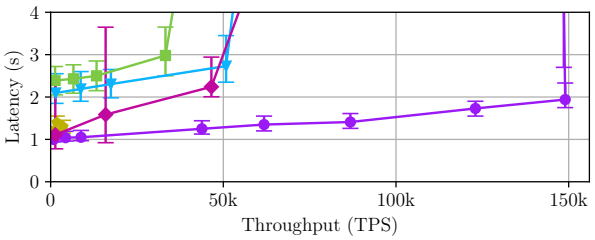


Figure 7: Latency/Throughput graph with 33 out of 100 crashed replicas.

Shoal++ *Faster Anchors* shows Shoal augmented only with the Fast Direct Commit Rule while Shoal++ *More Faster Anchors* additionally applies the multiple anchor augmentation. Finally, Shoal++ includes also the parallel DAG optimization.

The Fast Commit rule, in theory, improves anchor commit latency by up to 2md. In real, asymmetric network settings, however, it is not always 2md faster than the Direct Commit rule, and thus the practical latency improvement is smaller. The multiple anchor augmentation, in turn, improves latency significantly: (i) it saves, on average, 3md by eliminating the anchoring latency for most nodes (those that become anchors), and (ii) ensures that the anchoring latency for non-anchor nodes is typically only one DAG round because (with overwhelming probability) nodes are referenced by at least one of the (many) anchors in the next round. The parallel DAG framework improves latency further, by reducing queuing latency, but more importantly improves throughput scalability by approaching a data "streaming" effect; crucially, and unlike Narwhal-based worker designs, it does so *without* incurring additional latency.

8.3 Failure case

To quantify the robustness of Shoal++, relative to the baseline systems, we evaluate performance under two types of disruptions. First, we measure the performance of all systems while simulating crash failures of 33 (out of 100) replicas (Fig. 7). The performance of Jolteon remains largely unaffected, as its leader reputation mechanism quickly detects failures and subsequently elects only alive replicas. Shoal and Shoal++, likewise, swiftly adjust and elect only live anchors; latency, however, increases (up to ca. 2x at high load) as quorums must span more regions, increasing tail latency. Bullshark and Mysticeti, in contrast, suffer drastically increased latency since they do not employ a leader reputation mechanism to elect suitable anchor candidates, and thus must "wait" to commit until a non-faulty replica is elected as anchor.

Second, we measure the impact of sporadic message drops for Shoal++ and Mysticeti (Fig. 8). We inject network layer message drops for 1% of egress traffic in 5 nodes (out of 100 total), beginning at 60 seconds (red line). Under normal network conditions and a low-moderate load (18k tps) both Shoal++ and Mysticeti offer 700ms median consensus latency. Upon failure injection the latency of Mysticeti rises

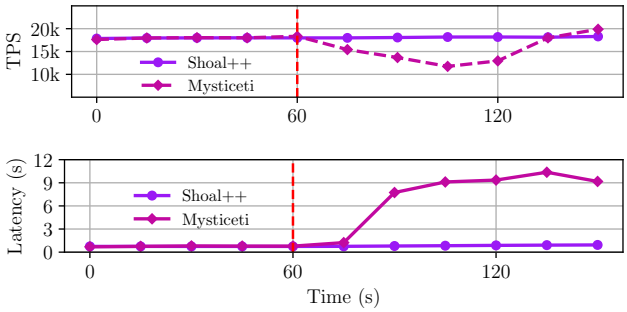


Figure 8: Impact of message drops in Certified vs Uncertified DAGs.

sharply (by a factor of 10x) as replicas scramble to perform critical-path synchronization on missing data (resulting, at times, in timeouts). As latency spikes, throughput initially drops as well, but recovers once the backlog of disseminated DAG proposals commit. Shoal++, in contrast, remains largely unaffected by message drops (latency rises to at most 1.3x). Because all nodes are *certified*, the DAG construction process in Shoal++ proceeds smoothly; any required synchronization is asynchronous and off the critical path.

9 Related work

Latency optimal BFT. Spurred by the seminal PBFT [15] protocol, a long line of systems [22, 24, 28, 30, 39] optimize for consensus latency. PBFT-based protocols designate a single leader to act as sequencer, and replace it only sporadically (e.g upon failure, or eagerly for fairness). This approach enjoys excellent latency. PBFT requires, when led by a correct replica, only 3 message delays (md); the optimum achievable latency in presence of faults, and optimal resilience of $n = 3f + 1$. Several followup works suggest optimistic Fast Paths [24, 28, 30, 39] that improve latency to only 2md in absence of faults, or by weakening resilience to $n = 5f + 1$. Unfortunately, single leader-based designs are fundamentally bottlenecked in throughput by the bandwidth and processing capacity of a single replica.

High throughput BFT. The key idea to scale beyond this bottleneck was realizing that networking resources must be fully utilized. Some works propose to decouple data dissemination from the consensus logic [11, 19, 46], while others [25, 37, 38] propose multi-log frameworks: these systems partition the request space and operate multiple parallel black-box instances of consensus protocols (e.g. PBFT) – each led by a different replica –, and carefully intertwine their outputs to construct a single, totally ordered log. This approach can achieve high throughput but requires complex coordination to deal with failures and re-configuration.

DAG BFT. More recently, a new class of so-called DAG-BFT protocols have emerged as the popular choice for high throughput BFT. At a high level, deployed DAG-BFT

protocols propose to explicitly separate data dissemination from consensus, and utilizing only a single common DAG data structure to implement consensus (§3).

HashGraph [13] was the first to propose a DAG-based consensus structure, and Danezis et al. [32] showed that any deterministic BFT protocol can, in theory, be embedded on a DAG. Aleph [21] and DAG-Rider [26] were the first to introduce additional rigidity to the DAG construction, leveraging a round-based structure to design asynchronous BFT protocols. Blockmania [18] was the first attempt to implement a DAG-BFT system, but fell short of adoption as it inherited a complex view change procedure, notorious to traditional BFT protocols such as PBFT.

Narwhal and Tusk [19] catapulted DAG-BFT to popularity by demonstrating that (i) (asynchronous) consensus can be embedded onto DAG-structures without the need for explicit view changes (Tusk), and (ii) throughput can be scaled nearly linearly through the addition of a horizontally scalable worker layer (Narwahl Mempool). Bullshark [35] improved upon Tusk by adopting partial synchrony and introducing a common case fast path during synchronous intervals; in a future iteration [36] Bullshark removed the asynchronous fallback, and established itself as the de facto partially synchronous DAG-BFT protocol.

Bullshark designates, at regular intervals (every two rounds), DAG-nodes to simulate a leader (anchor) akin to traditional BFT protocols, and commits them upon observing DAG patterns that ensure durable agreement of the leader. We refer to §3 for an overview of the Narwahl core DAG, and the partially synchronous Bullshark consensus protocol. Shoal improves the latency of Bullshark by (i) increasing anchor frequency to every round, (ii) utilizing a deterministic reputation scheme to select as anchors only the fastest, and best-connected replicas, and (iii) eliminating leader/anchor timeouts in all but extremely unlikely scenarios. Shoal, in a closing remark, briefly discusses the idea of multiple anchor per round; however, it does not provide detailed consideration.

Uncertified DAGs. A recent group of protocols propose the design of uncertified DAGs. Such constructions can, in theory, reduce latency, but are increasingly brittle as they are prone to data fetching on the critical path (§3.2). Cordial Miners [27] replaces the Reliable Broadcast (RB) in each round with best-effort broadcast (BEB), allowing anchors to (in the best case) commit within 3md; anchors are placed every other round, for an average latency of 4.5md. Similarly, BBCA-chain [29] replaces non-anchor nodes with BEB, and strengthens anchor nodes to single-shot pbft instances. Unfortunately, neither proposal has an implementation, and thus it is impossible to empirically evaluate practical performance.

Concurrent Latency reduction efforts. In concurrent efforts to Shoal++, several recent (non-peer reviewed) protocols suggest latency improvements, underlining the demand for high throughput systems with lower latency.

Sailfish [33] introduces an anchor commit rule that resembles the Fast Direct Commit Rule proposed in Shoal++. However, unlike Shoal++, it does not leverage the Bullshark Direct Commit rule as a potentially faster alternative. Furthermore, Sailfish does not employ the "more anchors" and "more DAGs" techniques, which enable Shoal++ to further reduce latency.

Mysticeti improves upon Bullshark's best case latency by transitioning to an uncertified DAG and implementing the Cordial Miners [27] consensus protocol. Mysticeti, like Shoal++, adopts Shoals high-level proposal to optimistically employ multiple anchors per round. Our evaluation demonstrates that Mysticeti almost matches Shoal++'s latency in the common case. However, as shown in §8, and independently corroborated by Autobahn [23], uncertified DAG constructions are prone to data fetching on the critical path, which negates the suggested latency benefits. A single Byzantine (or slow) replica, for instance, can impose at least 2 additional message delays *per* round in Mysticeti by not fully (or timely) disseminating its node proposals. In §8 we observed 10x latency degradation when 5 out of 100 nodes experienced 1% message drop.

Autobahn [23] introduces a promising DAG-*free* consensus approach in which replicas efficiently and reliably disseminate data independently in parallel "data lanes", and consensus commits only snapshot cuts of the lane states.

Leader reputation in BFT. To the best of our knowledge, leader reputation for BFT systems was first implemented in DiemBFT [43], and formalized by Carousel [17]. Shoal [34] and later Hammerhead [45] adopt leader reputation to DAG-BFT in an effort to exclude underperforming leaders from anchor candidacy. Shoal++ extends the approach to provide, in each round, a set of anchor candidates.

10 Conclusion

This work introduces Shoal++, a DAG-based BFT consensus protocol that offers high throughput, while significantly lowering e2e latency compared to current state of the art DAG-BFT protocols. Shoal++ takes as its starting point existing state of the art DAG-BFT protocols, and augments them in three key ways: (i) it reduces, in the common case, the commit latency of leaders (anchors), (ii) dynamically reinterprets anchor schedules to increase anchor frequency, and (iii) operates multiple parallel DAG instances to increase proposal frequency. This, in sum, allows Shoal++ to reduce the fault-free average e2e latency from 10.5 message exchanges (Shoal [34]) to only 4.5 message exchanges, within reach of traditional latency optimal (but throughput-inefficient) BFT protocols such as PBFT.

We note that the parallel DAG instances technique might be of independent interest, as this idea can be applied to any BFT algorithm to reduce its queuing latency.

References

[1] [n. d.]. Accolade Gathers $202 Million for New Blockchain Fund. https://arc.net/l/quote/srjmeosq Accessed: March 2025.

[2] [n. d.]. The Digital Euro. https://www.ecb.europa.eu/euro/digital_euro/html/index.en.html Accessed: May 2024.

[3] [n. d.]. GCP n2d machines. https://cloud.google.com/compute/docs/general-purpose-machines Accessed: May 2024.

[4] [n. d.]. Google Cloud Platform. https://cloud.google.com/ Accessed: May 2024.

[5] [n. d.]. Janus Henderson to follow BlackRock and Fidelity into tokenisation. https://arc.net/l/quote/gvxzbiuv Accessed: March 2025.

[6] [n. d.]. Microsoft Confidential Consortium Framework. https://www.microsoft.com/en-us/research/project/confidential-consortium-framework/ Accessed: May 2024.

[7] [n. d.]. Mysticeti codebase. https://github.com/asonnino/mysticeti/tree/paper Commit 96fd831.

[8] [n. d.]. Noise Cyrpto Framework. https://github.com/noiseprotocol/noise_spec Accessed: May 2024.

[9] [n. d.]. RocksDB. https://rocksdb.org Accessed: May 2024.

[10] [n. d.]. Tokio: Rust Asynchronous Runtime. Accessed: May 2024.

[11] Balaji Arun and Binoy Ravindran. 2022. Scalable Byzantine Fault Tolerance via Partial Decentralization. *Proc. VLDB Endow.* 15, 9 (May 2022), 1739–1752. https://doi.org/10.14778/3538598.3538599

[12] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. *arXiv preprint arXiv:2310.14821* (2023).

[13] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* (2016).

[14] Dan Boneh, Benjamin Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *Advances in Cryptology - CRYPTO 2001 (Lecture Notes in Computer Science, Vol. 2139)*. Springer, 514–532. https://doi.org/10.1007/3-540-44647-8_32

[15] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

[16] Jing Chen and Silvio Micali. 2019. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science* 777 (2019), 155–183.

[17] Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. 2022. Be aware of your leaders. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 279–295.

[18] George Danezis and David Hrycyszyn. 2018. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620* (2018).

[19] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.

[20] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[21] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 214–228.

[22] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 296–315.

[23] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (2024).

[24] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.

[25] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. Rcc: Resilient concurrent consensus for high-throughput secure transaction processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1392–1403.

[26] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. In *Proceedings of the 40th Symposium on Principles of Distributed Computing (PODC '21)*.

[27] Idit Keidar, Oded Naor, and Ehud Shapiro. 2022. Cordial Miners: A Family of Simple, Efficient and Self-Contained Consensus Protocols for Every Eventuality. *arXiv preprint arXiv:2205.09174* (2022).

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[29] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2023. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. *arXiv preprint arXiv:2310.06335* (2023).

[30] J-P Martin and Lorenzo Alvisi. 2006. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.

[31] Team Rocket. [n. d.]. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. ([n. d.]).

[32] Maria A Schett and George Danezis. 2021. Embedding a deterministic BFT protocol in a block DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 177–186.

[33] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2024. Sailfish: Towards Improving Latency of DAG-based BFT. Cryptology ePrint Archive, Paper 2024/472. https://eprint.iacr.org/2024/472.

[34] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2024. Shoal: Improving DAG-BFT latency and robustness. In *International Conference on Financial Cryptography and Data Security*, Jeremy Clark and Elaine Shi (Eds.). Springer.

[35] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.

[36] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The Partially Synchronous Version. *arXiv preprint arXiv:2209.05633* (2022).

[37] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. *CoRR* abs/1906.05552 (2019). arXiv:1906.05552 http://arxiv.org/abs/1906.05552

[38] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 17–33.

[39] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.

[40] The Algorand Team. 2019. Algorand Networks. https://www.algorand.foundation/ Accessed: April 2024.

[41] The Avalanche Team. 2020. Avalanche Networks. https://www.avax.network/ Accessed: May 2024.

[42] The Aptos Team. 2022. Aptos Networks. https://aptosfoundation.org/ Accessed: May 2024.

[43] The Diem Team. 2021. DiemBFTv4. "https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf" Accessed: May 2023.

[44] The Sui Team. 2023. Sui Networks. https://sui.io/ Accessed: May 2023.

[45] Giorgos Tsimos, Anastasios Kichidis, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2023. Hammerhead: Leader reputation for dynamic scheduling. *arXiv preprint arXiv:2309.12713* (2023).

[46] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. 2022. {DispersedLedger}:{High-Throughput} Byzantine Consensus on Variable Bandwidth Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[47] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, Peter Robinson and Faith Ellen (Eds.). ACM, 347–356.

Learning Production-Optimized Congestion Control Selection for Alibaba Cloud CDN

Xuan Zeng[1*], Haoran Xu[2*], Chen Chen[1*], Xumiao Zhang[1], Xiaoxi Zhang[2], Xu Chen[2], Guihai Chen[3]
Yubing Qiu[1], Yiping Zhang[1], Chong Hao[1], Ennan Zhai[1]
[1]*Alibaba Cloud* [2]*Sun Yat-sen University* [3]*Nanjing University*

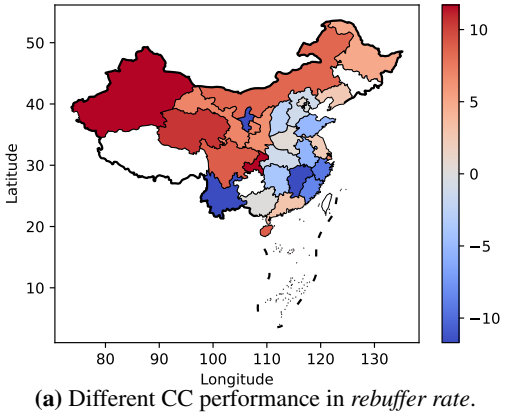
Abstract

Today's content delivery networks (CDNs) typically use static congestion control (CC) configurations, yet the diverse network environments preclude a universally optimal CC for all geographical regions, as evidenced by our extensive measurements. Current CC algorithms, limited by narrow applicability or high maintenance costs, struggle in large-scale CDNs. This work introduces ALICCS, the first CC Selection (CCS) approach tailored for production CDN, integrating fine-grained domain knowledge for learning to choose the best CC from existing, well-established ones. Through an over-one-year real-world deployment in Alibaba Cloud CDN, ALICCS has enhanced the Quality-of-Experience (QoE) by up to 9.31%, surpassing the competitive margin in the CDN market, and significantly reduced the retransmission rate by 25.51% to 174.36% across all provinces of China, leading to cost savings over 10 million US dollars. We also share key insights and experiences from deploying ALICCS at scale, highlighting traffic patterns in Alibaba Cloud CDN.

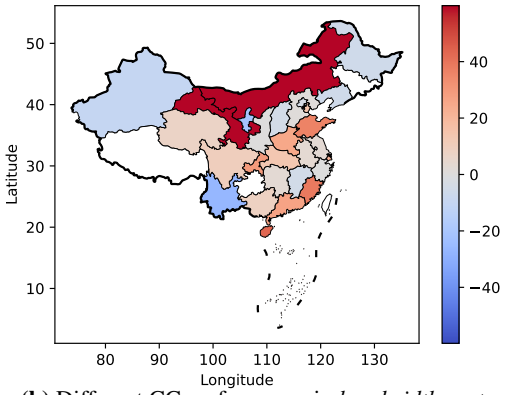
1 Introduction

Content Delivery Networks (CDNs) serve as the invisible backbone of the Internet, responsible for carrying over 70% of global traffic [4]. Enhancing Congestion Control (CC) in CDNs can significantly improve user experience by directly tackling network efficiency and reliability. It also achieves considerable cost savings for CDN operators, by offering a simpler, more resource-efficient alternative, compared to network or application layer modifications. Therefore, as a CDN provider, who has the greatest control and clearest visibility at the transport layer where CC is central, optimizing CC enables us to make more impactful changes that ISPs or application providers might not readily achieve (§2).

Currently, most CDNs still adopt the one-size-fits-all CC configuration [16, 32] across their geographically distributed regions, yielding a suboptimal overall QoE due to the vast diversity of CDN network environments. Conventional approaches either *fine-tune the parameters* of one type of CC or *invent a new CC*, targeting only specific network scenarios. Our experience shows that leading CC solutions exhibit significant variations in user QoE for content delivery services. No single CC scheme can always be the optimal choice across all regions of our CDN network, *e.g.,* ranging from rebuffer rate (Figure 1a) to bandwidth cost (Figure 1b)[1]. Given this complexity and variability, we raise a question: *How can we accurately select the best CC among well-established ones for distinct CDN nodes under dynamic network conditions?*



(a) Different CC performance in *rebuffer rate*.



(b) Different CC performance in *bandwidth cost*.

Figure 1: Performance comparison across 29 provinces in China. Red (Blue) means CUBIC (BBR) achieves better performance. A darker color indicates a bigger performance gap.

Machine learning (ML), with its widely proven proficiency in learning from diverse data, is particularly suited to address the dynamic and heterogeneous nature of CC Selection (CCS). It can adapt to varying CC performance across different factors. Despite this, extensive research on modifying CC, including learning-based approaches [14, 26, 45, 47, 49, 52], have not fully met the unique and stringent needs of production CDNs. From our experience, applying ML to CCS for large-scale production CDN presents the following key challenges, directly stemming from critical CDN requirements:

[*]These authors contributed equally to this work.

[1]Details on the fair A/B test of CUBIC and BBR are in Appendix §A.

Table 1: Comparison of ALICCS with state-of-the-art ML-based Congestion Control Selection methods.

	Optimization Goal	Granularity	Satisfy CDN req.[I]	Scales of CDN nodes, users/requests
Sage [49]	Throughput/latency related	N/A[II]	No	N/A
Disco [47]	Throughput/latency related	Per-connection	No	N/A (<10 servers and clients emulated)
Configanator [32]	Web performance (PLT)	Per-Network-Class	Yes	3 Global CDN PoPs, 8.2M requests
ALICCS	Rebuffer rate, latency	Per-connection	Yes	>1000 CDN nodes, 500M requests

[I] CDN requirements refer to those mentioned in §1: 1) scalability & generalizability, 2) interpretability, 3) managing inference overhead.
[II] N/A stands for Not Available.

- **Scalability and generalizability challenges.** Our CDN spans various geographical locations and network environments, necessitating distinct models for different nodes. However, managing individual models for the thousands of nodes in our global CDN is impractical due to extensive maintenance requirements. Moreover, most ML algorithms assume independent and identically distributed (i.i.d.) data between training and test domains, which may not hold true in practice. Developing a unified ML model that can efficiently scale and generalize across diverse data environments remains a crucial yet challenging goal.
- **Enhancing interpretability and accountability for deploying ML-based CCS.** Standard statistical tools alone do not suffice to evaluate ML models accurately due to potential sample selection bias and confounding factors [35]. It is therefore essential to integrate deep domain knowledge from our CDN with these tools to thoroughly assess the model's effectiveness and increase its interpretability before a full-scale CCS deployment in our production CDN.
- **Managing ML inference overhead without compromising CCS performance.** High performance of ML models often requires complex inference logic, leading to significant processing delays. Our real-world measurements reveal that non-optimized per-connection ML inference, *executed serially*, incurs significant overhead and undermines the queries per second (QPS) performance (refer to Table 3). Enhancing run-time efficiency with reduced ML inference delays is therefore vital to cost-effective CCS.

These challenges are not fully addressed by existing learning-based CCS efforts, summarized in Table 1. They do not incorporate the requirements and domain expertise of realistic large-scale CDN networks. Moreover, their reliance on methods like reinforcement learning [47,49] or online learning [32] necessitates handcrafted reward functions and immediate QoE feedback, both of which are impractical for the short video services supported by our CDN based on our experience.

To address the above challenges simultaneously, we introduce ALICCS, the first domain-specific, ML-based CCS scheme, tailored to *short video delivery*, a dominant workload in Alibaba Cloud CDN. ALICCS dynamically selects the most effective CC for each connection[2], aiming to (1) boost short video QoE, by reducing rebuffer rates and start-up delays, and (2) lower bandwidth cost, *e.g.,* retransmission ratio.

In developing ALICCS, we make the following contributions:

Firstly, it is crucial to define the right features and labels from usable data for ML models that predict key factors to optimize CCS, scale to numerous CDN nodes, and generalize to new environments. We reveal an observation (detailed in Figures 3 and 4) that network types decisively influence the best CCS. Therefore, our goal is to effectively learn a mapping from end-to-end TCP statistics to network types. However, this relation is *unstable* due to hidden states [35] like underlying network conditions. We then propose a decomposed Causal Graphical Model (CGM) using Generative Adversarial Networks (GAN) and domain knowledge, extracting features *that are invariant to the hidden states*. Additionally, we develop a hierarchical clustering algorithm to group a vast number of IP prefixes for model scalability improvement.

Secondly, to enhance the model's transparency and interpretability, we employ knowledge distillation to transfer insights from our complex, well-trained deep neural network (DNN) into a compact decision tree. This tree, developed using training data enhanced by the predictions of our DNN model, effectively replicates the DNN's decisions in a format that is easier to interpret. When we encounter performance degradation during online CCS inference, we substitute our large DNN model with this decision tree. This allows for precise tracking of any suboptimal predictions by identifying the specific features contributing to the inefficiency, providing better interpretability of our ML-produced CCS solutions.

Thirdly, we significantly reduce model inference overhead by constructing an IP-prefix based cache that leverages temporal and IP-prefix-specific similarities in the domain of the best CCS and network types. ALICCS has achieved 64.30× improvement in latency and 2.42× improvement in QPS for online CCS inferences. Moreover, we realize fine-grained CCS by implementing it in a per-connection manner. This also bypasses the complexity and QoE sensitivity issues in selecting the grouping granularity, which is inherent in existing CC control methods that rely on request grouping [32].

Deployment and experience. ALICCS has served Alibaba Cloud CDN's production for over one year. We train our model on labeled data from App #1[3], one of the leading short video companies in the world, obtained from about 30% of the CDN nodes supporting its services. The well-trained model serves online CCS inference in an over-one-year in-production deployment for both App #1 and App #2,

[2] Multiple requests can multiplex on each CDN connection.

[3] Partner apps are anonymized due to NDA of our company.

two major providers in the global short video market. ALICCS achieves 95.8%–99.0% Wi-Fi/4G prediction accuracy in about 400 nodes of three main ISPs in China and Southeast Asia, covering almost all provinces in China, demonstrating our ML model's generalizability. Combining our model's predicted network access type and the mapping to CC choices, we reduce the rebuffer rate by 4.76% and retransmission bandwidth by 25.51%–174.36%. In terms of run-time efficiency, ALICCS achieves CPU usage below 0.79% (at most two cores per CDN node) and maintains the minimum memory consumption below 2.90 GB (see Figure 11). In addition, it achieves 64.32× processing delay reduction and 2.42× maximum QPS improvement (see Table 3).

2 Background and Motivation

In this section, we detail our motivation for employing CCS in Alibaba Cloud CDN, focusing on short video services. We then share observations that drive ALICCS's design choices.

2.1 Overview of CDN at Alibaba Cloud

Alibaba's CDN spans over 70 countries with a total egress bandwidth exceeding 180 Tbps that serves over 120 EB (*i.e.,* 120×2^{60} Bytes) of data every year. In particular, it has 2800+ nodes in China and South Asia, covering a wide range of geographical regions and ISPs, where ALICCS is deployed. By caching content at edge servers, the CDN significantly reduces back-to-origin bandwidth and enhances user proximity. It offers diverse services, ranging from large file transfers to media streaming services, among which short video services take the most significant share of traffic demands.

CDN providers aim to deliver high-quality services cost-effectively, focusing on improving *user QoE* and reliability to stand out from competitors. *Bandwidth costs* constitute a major portion of operational expenses, which means even slight and relative reductions in data transfer can lead to significant absolute savings at scale.

2.2 Motivation: CCS for Short Video Services

Short videos, *e.g.,* supported by App #1 and App #2, have become a major workload of Alibaba Cloud CDN. The CDN should then be optimized to deliver short videos with high user QoEs while incurring minimum bandwidth costs.

QoE of short video delivery. Efficient content delivery for short videos in CDNs necessitates quick video start times and smooth playback. Traditional video streaming focuses on metrics like bitrate, resolution, and loading appropriate initial buffering video segments (typically 2-10 seconds) to ensure high-quality, continuous playback over longer durations. In contrast, short video services emphasize minimizing the rebuffer rate to reduce stalls and the initial buffering delay, typically less than 1 second for the first frame to load.

Why adjust CC? Extensive efforts have been made to improve video delivery QoE by refining application layer strategies like Adaptive Bitrate (ABR) algorithms and content pre-fetching. However, these are beyond the control of CDN providers and thus not considered in this work. In large-scale production CDNs, optimizing CC directly targets network bottlenecks, leading to immediate improvements in content delivery, especially for short video services. Besides, adjusting CC offers a straightforward and resource-efficient alternative to overhauling application layer configurations, including fine-tuning ABR [7, 30, 43] and content pre-fetching [17, 29] settings, or transitioning from HTTP/2 to HTTP/3 [5, 42].

Table 2: Performance comparison of different CCs for 4G networks (*CUBIC Agg.* means CUBIC tuned to aggressively trigger retransmissions to lower rebuffer rates).

CC Algorithm	CUBIC	CUBIC Agg.	BBR
Video Rebuffer Rate	1.13	1.05	1.0
Retransmission Rate	1.41	1.53	1.0

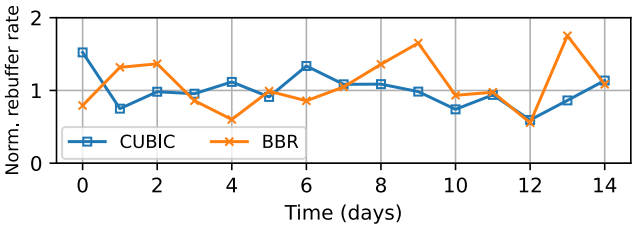


Figure 2: A two-week A/B test comparing BBR and CUBIC, serving short videos for App #1 in one CDN node.

Why CCS? While conventional approaches generally focus on parameter fine-tuning (*e.g.,* Bayesian Optimization) for a given CC [8], our experience has shown that the benefits of such tuning can be marginal in certain scenarios. This is because each CC algorithm has inherent limitations in how it identifies and reacts to congestion, which may fall short in situations violating their assumptions. For instance, in tuning CUBIC for 4G networks, a substantial increase in sensitivity to trigger retransmissions is necessary to match BBR's rebuffer rate in 4G. This adjustment, however, leads to an increase in the retransmission rate from 1.41 to 1.53 (see Table 2). Compared to the alternative of *inventing new CC algorithms*, our approach of selecting the best among long-living CC schemes, known for their effectiveness in variable scenarios, *provides robust worst-case performance*. It reduces potential revenue loss associated with inadequately tested CCs.

Why ML for CCS? Due to dynamic network conditions and user traffic in CDNs, *rule-based* CCS approaches are generally impractical. This is supported by our measurements showing volatile patterns of CC performances across both regions (Figure 1)[4] and time (Figure 2). Moreover, our choice of using ML for CCS is driven by our observations and domain knowledge, based on which CCS can be reduced to a classification problem, elaborated in §2.3. This problem reduction directly informs our model design, where we integrate supervised

[4]We find regional proximity does not yield network metrics' similarity, likely due to diverse traffic policies (*e.g.,* rate-limiting) in nearby regions.

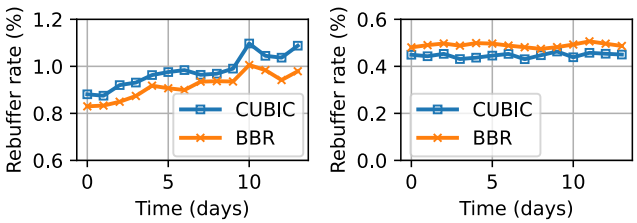


Figure 3: Contrasting advantages of BBR and CUBIC varying network types, *i.e.,* 4G (left) and Wi-Fi (right).

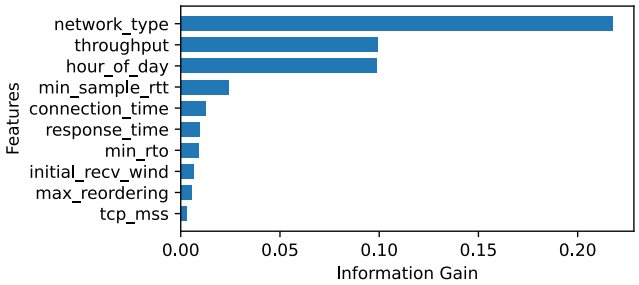


Figure 4: Top 10 features on our CDN with the highest information gain for predicting the best CC candidate. These features span four categories: transport layer (TCP) statistics (*e.g.,* retransmission rate and throughput), application layer statistics (*e.g.,* response time), link layer statistics (*i.e.,* network type) and timing (*i.e.,* hour of the day).

learning with fine-grained domain knowledge. Consequently, other ML choices, such as reinforcement learning [47, 49, 52] and no-regret (or online) learning [14, 32, 50] do not align well with our needs. They are in general tailored to sequential processes where data are gradually available, adding unnecessary complexity and inefficiency to CCS.

2.3 Key Observations

Our goal is to develop an ML system that selects the best CC algorithm from a set of candidates to enhance user QoE and reduce bandwidth costs for short video services across extensive CDN nodes.

Observation I: CCs' advantages vary regionally and temporally. Despite extensive efforts to optimize the performance of individual CC algorithms, we find that no single algorithm consistently outperforms others in all scenarios. Our Figures 1a and 2, illustrate that the superiority of BBR and CUBIC in rebuffer rate varies across regions and fluctuates over time, demonstrating that using a fixed CCS rule is generally impractical. This observation aligns with recent findings [49], which report similar fluctuations in the CC superiority.

Observation II: Network types decisively influence CC advantages. We compared short video rebuffer rates between BBR and CUBIC over 14 days, presented in Figure 3. Both groups, represented by the blue and orange curves respectively, use the same network type, *i.e.,* 4G for the left figure and Wi-Fi for the right. This reveals a strong correlation between network access and the best CC choices: BBR outperforms in 4G connections (left) while CUBIC wins under Wi-Fi (right). Other CCs like Vegas [11], Reno [23], and Westwood [40], exhibited significantly inferior performances (*e.g.,* 30%–40% higher rebuffer rate than CUBIC and BBR) for short video delivery in our production CDN, based on comprehensive pre-deployment measurements. To ensure performance reliability and stability, we choose CUBIC and BBR as the CC candidates. Moreover, we examine the impact of different features on CC selection. Figure 4 shows that network type has the highest information gain [3], vastly outweighing other features like retransmission rate and hour-of-the-day with information gain of 0.001 and 0.005 respectively[5]. This observation leads us to reduce the CCS problem by focusing on predicting the network type for each TCP connection, as it provides the most substantial predictive power of the best CC and keeps system complexity manageable. We then deterministically choose CUBIC for Wi-Fi and BBR for 4G to optimize short video QoE.

Observation III: ISP rate-limiting impacts CCs differently. Based on Observation II, we hypothesize that the contrasting performance of CUBIC and BBR varying network types is attributed to the ISP's traffic policies, particularly token bucket rate-limiting that may cause premature packet losses[6], which CUBIC misinterprets as congestion signals, significantly reducing throughput. To confirm this, we first utilize BBR's bandwidth probing mechanism to assess rate-limiting strength over months and then examine how CUBIC and BBR respond. We observe that 4G's rate-limiting generally increases from the start towards the month's end. During this, CUBIC's average throughput drops to less than 10% of that without rate-limiting (1472 Kbps vs. 8728 Kbps), and its average retransmission rate surges to 73× higher than its non-rate-limited performance. It highlights that CUBIC struggles in 4G rate-limiting conditions, while BBR's bandwidth probing technique can instead effectively identify the bandwidth constraints. Regarding BBR's inferior performance on Wi-Fi, it is primarily due to Wi-Fi's frame aggregation mechanism, which undermines BBR's bandwidth probing capabilities [18].

Observation IV: Network types need to be predicted dynamically. Obtaining network type via client reporting is impeded by software upgrade overhead and unreliable client data. Notably, labeled data from client accounts for only 5%–10% of total requests (see labeled data acquisition in §3). Additionally, private IP database solutions like MaxMind [2] offer limited coverage and accuracy of network type labels for our CDN network. Moreover, we observe that with 90% probability, a /24 IP prefix will significantly shift its proportion of dominant type (Wi-Fi or 4G) in 15 hours, necessitating dynamic network type prediction.

[5] Features like retransmission rate are analyzed but not shown in the figure, as they ranked outside the top 10.

[6] Packet losses can be categorized as transmission, congestion (from link contention), and rate-limiting loss. While the first two vary over time within the same network, the rate-limiting loss is consistently bound to 4G networks.

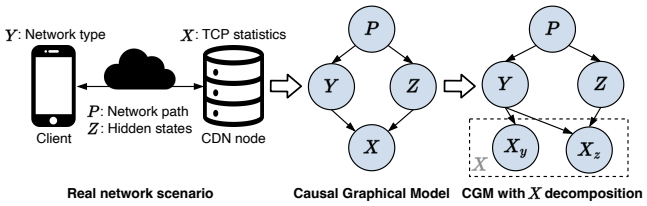


Figure 5: Illustration of decomposing features via CGM.

3 Design and Implementation

In this section, we outline the design of a unified ML model for all CDN nodes, enabling stable mapping from TCP statistics (additional features are listed in Appendix §B) to network types. One naïve solution would be to collect training data from all regions and train a comprehensive model. However, this method raises two significant issues: (1) The QoE performance in worst-case scenarios is not guaranteed, which is critical for production CDNs. (2) Regions that are underrepresented in the training data can impede model inference performance due to limited generalization. Essentially, weak network conditions and regions with fewer requests are often insufficiently represented in the training set. Training the model naïvely without eliminating the effects of these diverse network conditions, which are represented as underlying network characteristics, may lead to overfitting on good conditions or regions with ample requests. This, in turn, degrades model accuracy in poor network conditions or small regions, as evidenced in §5.4, undermining its capability to provide worst-case guarantees. Another straightforward CCS approach is to directly adopt the network type labels from datasets and then choose CUBIC for Wi-Fi and BBR for 4G. However, network types in the datasets are often unreliable and thus need to be predicted dynamically (refer to Observation IV).

3.1 Model Design and Training Method

Prior research on applying ML to networked systems often overlooks model generalizability under volatile network environments. In contrast, our model design aims to learn causal relationships between TCP data and network types[7], guided by our key observations (§2.3). This approach enhances prediction accuracy in new environments, crucial for generalizability. However, our measurements reveal that the relationship between TCP statistics and network type is generally *unstable*. This instability likely stems from end-to-end TCP statistics being influenced by network types as well as varying path-specific characteristics, such as buffer sizes and the number of competing flows across CDN nodes and IP prefixes.

Design overview. To tackle the instability in learning, our idea is to decompose the raw data features, which are TCP statistics ($\triangleq X$), into two parts, denoted by X_y and X_z respectively in Figure 5: one influenced solely by network types ($\triangleq Y$) and the other influenced by both Y and hidden state ($\triangleq Z$). Our goal is to create a model based only on features X_y for model stability. To do so, we construct a decomposed Causal Graphical Model (CGM) and tease apart the impact of network types Y and hidden states Z on TCP statistics X, by leveraging the one-to-one mapping between network paths ($\triangleq P$) and IP address prefixes observable on CDN servers. This ensures a stable model that maps X to Y while remaining *causally invariant to* Z. Finally, we incorporate domain knowledge as constraints and regularization terms, to ensure model scalability and reduce the noise.

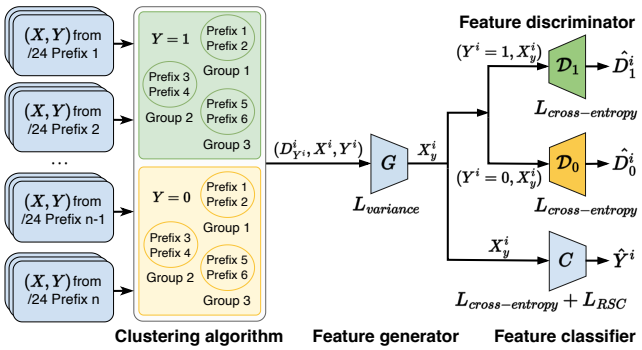


Figure 6: Overview of our ML model design.

Decomposing features via CGM. Ideally, to develop a generalized model for the diverse CDN nodes, we need to identify features for any TCP connection that have a stable relationship with our target Y. However, directly using X, the measured TCP statistics (*e.g.,* maximum congestion window size), is insufficient due to their dependence on both the network type Y and varying hidden factors Z (*e.g.,* path capacity), as depicted in the middle CGM in Figure 5. Different CDN nodes involve various paths P (interchangeable with connections) and thus different hidden factors Z, causing instability in the X and Y relationship. To address this, we leverage our decomposed CGM with X_y and X_z, the decomposed features of X (refer to Figure 5), and a condition satisfied by X_y: $Pr(X_y|Y,P) = Pr(X_y|Y)$. Therefore, our goal is *to identify features* X_y *for each connection that maintain a constant conditional probability distribution* $Pr(X_y|Y)$ *across all connections*. By training a classifier with these feature-target pairs $\{X_y^i, Y^i\}_{i=0}^m$, with a total of m samples, we can create a model that performs reliably on every CDN node. This is realized by integrating this decomposed CGM into a GAN, where a generator is trained to obscure the discriminator's ability of predicting the path from which the sample originates.

Learning path-invariant features via GAN. Through the above mathematical insights rooted in our decomposed CGM, learning a stable mapping from X to Y is converted to learning a path-invariant feature representation X_y given Y. Furthermore, leveraging the one-to-one mapping relationship between IP prefix and Internet path[8], the goal becomes *learning*

[7] The minor case of non-1:1 mapping between connection and network type is not yet addressed. Potential solutions are outlined in Appendix §C.

[8] The association between IP prefixes and Internet paths stays stable for several hours, despite occasional changes due to load balancers.

an IP-prefix-invariant feature representation for any specific network type. To achieve this, we adopt the GAN framework and construct two discriminators, marked by $\mathcal{D}_1$ and $\mathcal{D}_0$ in Figure 6, each outputting which /24-prefix group the sample comes from, for Wi-Fi ($Y=1$) and 4G ($Y=0$), respectively We then formulate two regularizers, R_{D_0} and R_{D_1}, based on the discriminators' predictions $\widehat{D}_0$ and $\widehat{D}_1$, corresponding to the ground-truth group labels D_0 and D_1, respectively[9]. These regularizers represent the classical cross-entropy-based classification loss of discriminators and are formulated as:

$$R_{D_0} = L_{cross-entropy}(D_0, \widehat{D}_0) \tag{1}$$

$$R_{D_1} = L_{cross-entropy}(D_1, \widehat{D}_1) \tag{2}$$

Scalability with domain knowledge-assisted clustering. Ideally, the outputs D_0 and D_1 correspond to the vectors indexing IP address prefixes. However, a scalability challenge arises: the discriminators need to distinguish which IP prefix each sample originates, among tens of thousands of candidates at each CDN node, causing intractable training complexity. To tackle this, we modify the discriminators' original outputs to be IP-prefix *groups* and introduce a domain knowledge-assisted clustering algorithm. As shown in Figure 6, we first separate the training data based on whether Y equals 1 or 0 and then adopt K-means clustering to group the IP /24 prefixes, with Euclidean distance adopted as the distance metric.

Labeled data acquisition. We collect labeled data through collaboration and coordination with one of the short video service's client end. The label of network type is piggybacked via http header extensions from client side. Note that due to the overhead associated with software development and client upgrades, the volume of labeled data obtainable from client side is limited between 5% and 10% of its total requests with limited region coverage in a limited period of time.

Feature selection. Our CDN spans nearly all provinces in China, handling numerous TCP connection features from user requests, requiring efficient feature selection due to significant data scale and diversity. We choose information gain as the selection criteria for its intuitiveness and broad applicability. However, a one-shot feature does not meet our production-level accuracy requirements. Therefore, we also periodically diagnose ALICCS's model performance by combining the knowledge-distilled decision tree and Shapley value analysis to tweak the feature sets, which will be elaborated in §3.2.

While the core of our ML model to predict network types, integrating a decomposed CGM with domain-specific GAN and clustering, it also demands meticulous handling of numerous corner cases. Below, we outline two additional techniques used in our production CDN for effective model training.

[9] In practice, to eliminate CC algorithm bias and obtain IP-prefix-and-CC-invariant features, we can instead cluster data by the combination of CC and IP-prefix, and re-define the outputs of $\mathcal{D}_1$ and $\mathcal{D}_0$ to be the combinations. Or, we can add another discriminator for it.

Diversified features for smaller ISPs. Through training with CDN data, we have learned that, while a universal feature representation generally works well, significant deviations in feature distribution exist, in particular for smaller ISPs compared to major ones. This discrepancy can cause a model to underperform with smaller operators. By identifying and transforming alternative features from smaller ISPs to align with the broader distribution, we can ensure a more robust and effective model. To enhance generalizability and avoid over-reliance on dominant features, our model employs the Representation Self-Challenging (RSC) method [22]. Unlike Dropout, which randomly deactivates features, RSC uses gradients to selectively mute predictive features, encouraging the model to utilize a broader range of features. Therefore, we integrate an RSC-based regularizer into our training loss, denoted as L_{RSC} (used later in Equation 5). This approach is particularly effective for data from smaller-scale ISPs, diversifying the model's focus and ensuring robustness across various scenarios that deviate from larger operators.

Noise reduction with domain knowledge. To reduce feature noises, we again utilize the domain knowledge that requests with the same IP /24 prefix usually follow identical IP paths. Thus, the extracted features from the same IP /24 prefix should align closely, sharing the same path-specific hidden states X_z and the desired X_y. We translate this insight into a regularizer $L_{variance}$. Let $\mathcal{S}_{prefix}$ denote the set of those X^j sharing identical IP /24 prefixes and $\overline{X}_{prefix}$ denote the average of $X^j \in \mathcal{S}_{prefix}$. We formulate $L_{variance}$, the sum of the variance of extracted features of each /24 prefix group, as follows:

$$L_{variance} = \sum_{prefix=i}^{n} (X^j - \overline{X}_{prefix})^2_{X^j \in \mathcal{S}_{prefix}}. \tag{3}$$

Training method. Utilizing Equations 1, 2, and 3, we construct the loss functions for training the discriminators, generator (G), and the classifier (C) according to the following:

$$L_D = R_{D_0} + R_{D_1}, \tag{4}$$

$$L_C = L_{cross-entropy}(Y, \widehat{Y}) + L_{RSC}, \tag{5}$$

$$L_G = L_{variance}, \tag{6}$$

$$L_{all} = L_C - \lambda_1 * L_D + \lambda_2 * L_G, \tag{7}$$

where L_D is the discriminator's classification loss, L_C is the classifier's classification loss, and L_{RSC} is a term inspired by RSC regularization. L_G represents the regularization term applied on the generator G. L_{all} is the overall training loss, with hyperparameters λ_1 and λ_2 balancing the trade-off between model fitting and generalization. Further details on the design trade-off can be found in Appendix §D. Adhering to the standard GAN training method, we alternate between two training phases. We initially train the discriminators using loss L_D, with fixed generator and classifier parameters. Subsequently, we train the generator and classifier using the combined loss L_{all}, while keeping the discriminators fixed.

3.2 Model Interpretability Enhancement

In addition to generalizability and scalability, it is essential to establish trust before deployment and understand the root cause of prediction errors. This is crucial for iterating and refining the model, especially during incremental deployment in a production CDN. Model interpretability entails two requirements: First, its behavior must be easily verifiable by operational engineers using their domain expertise. Second, it should accurately reflect the ML model's behavior. To this end, we have chosen a tree-based model, which satisfies these requirements and has proven to be particularly well-suited for networking problems that span local and global control [31].

Knowledge distillation with augmented features. We distill the knowledge of our GAN-based deep learning (DL) model into a decision tree-based model for performance diagnosis. We first input all features $\{X^i\}_{i=0}^m$ from the training set to the trained DL model to obtain the prediction probabilities $\{Y_0^i, Y_1^i\}_{i=0}^m$. Then, we combine them to create a distilled version of the training set, $\{X^i, Y_0^i, Y_1^i\}_{i=0}^m$. Next, we construct a multi-output regression decision tree and train it using the distilled training set. Each output in the tree corresponds to the predicted probability of one CC selection solution, *i.e.,* Wi-Fi and 4G. The utilization of feature-probability tuples (X^i, Y_0^i, Y_1^i) rather than the standard feature-label pairs (X^i, Y^i) reflects our domain-specific GAN-based model design, where we construct two discriminators for Wi-Fi and 4G network types respectively, leading to higher interpretability.

Using our decision tree in practice. The tree-based models enhance the robustness of ML models by enabling performance issue debugging and helping understand the model predictions to build trust. For instance, during the initial deployment phase, when encountering a low accuracy within a specific /16 IP prefix, we applied Shapley analysis [28] to the output of the decision tree for those examples. This combination allows us to quantify feature importance for predictions on subset of samples and understand the model predictions for certain IP prefixes. We then found that the model excessively relied on the TCP maximum segment size (MSS) feature from that specific /16 IP prefix. Therefore, we adjusted the model to reduce its reliance on MSS and broaden its feature usage, effectively boosting the accuracy. Beyond debugging, decision tree models also assist the production team in comprehending and gaining confidence in our ML model, which is essential for its deployment in production. It is crucial to note that the distilled decision tree is employed solely for performance diagnosis rather than online inference, for two primary reasons. First, our caching mechanism, discussed later in §3.3, effectively mitigates DL model's inference costs. Second, empirical observations reveal that the distilled model's accuracy is much lower than that of our DL model. Nevertheless, despite a 5%–7% accuracy drop, we find the distilled tree model still captures error-related features, by aligning its prediction with the DL model for most samples. For subsets with significant prediction discrepancies, we increase the tree depth and retrain the tree until the alignment passes a threshold.

3.3 Inference Pipeline at CDN Nodes

Integrating our ML model (§3.1) into the networking stack of CDN nodes poses two technical challenges: (1) The model inference time must be negligible compared to the normal request processing time to maintain high request processing rates (*e.g.,* approximately 10k QPS) for any CDN node. (2) Hardware costs need to be considered, as excessive use of additional CPU or GPU cores per CDN node can be impractical and costly, potentially outweighing the benefits. We develop an efficient inference pipeline to tackle these challenges.

Centralized vs. distributed inference deployment. To implement automatic and intelligent CCS across CDN nodes using a well-trained DL model, we evaluated two approaches: centralized model inference services and distributed model replication on each node. Empirically, the centralized approach offers more abundant hardware resources in a cluster to deploy the model, facilitating faster ML model inference. However, the network latency between the centralized cluster and geo-distributed CDN nodes can be prohibitively high, potentially causing millisecond-scale delays for serving each batch of requests. Therefore, we opted for the distributed approach, replicating and performing inference locally on each CDN node to achieve greater responsiveness and efficiency.

This decision to adopt a distributed approach leads us to an efficient inference pipeline, grounded by two key observations drawn from comprehensive empirical studies:

- **Temporal stability of the best CCS and IP-based caching.** Consistent with a previous study [47], we find that the best CC choices, influenced by network path characteristics, remain stable for several hours. Specifically, our measurement reveals that a /24 IP prefix stays dominated by either Wi-Fi or 4G type (purity over 90%) for more than 2 hours with 87% probability[10]. And it can remain dominated by one of these types for 15 hours or more with 10% probability. This stability allows us to cache inference results for requests from specific paths, enabling their reuse for subsequent requests from the same path and significantly reducing inference costs over time.
- **Consistent CC choices based on IP prefix similarity.** IP networks are typically organized in subnets, and IP routing is based on the longest prefix matching. As a result, requests from the same IP prefix often undergo similar path decisions at a given time. As shown in Figure 7, among the top 10K /24 prefixes with the most requests, 90% of them exhibit a *purity* (percentage of connections with the same network type) higher than 91%, and these prefixes account for 84% of the total requests seen by our CDN. This allows us to not only cache results for the last few hours but also aggregate the results of our predicted CC choices

[10] We infer these dynamics may stem from ISP's load balancing and management, but we lack visibility into ISP's network to validate such hypothesis.

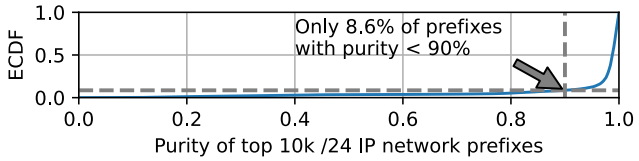


Figure 7: CDF of purity distribution of /24 prefixes.

under the same IP prefix into a single entry, leveraging their homogeneity in CC selection to save memory.

Inference pipeline. Figure 8 depicts the pipeline developed utilizing the above insights. We minimize the inference delay in the request processing pipeline by decoupling the process into online cache lookups (shown as the "Online processing pipeline" box) and periodic offline updates (shown as the "Offline processing pipeline" box). The latter records pairs of requests' IP addresses and the CC choices chosen given the predicted network types. The update is performed hourly to save computations, and also match the temporal stability of the best CCS. Every time a request server encounters a new client connection, it queries the mapping cache for the CC decision. This reduces the CCS latency from running a complex DL model inference to a lightweight cache lookup.

Efficient IP lookup cache. We implement an IP lookup cache using the Trie data structure [10,19,34,38,48], which provides O(1) lookup complexity with special hardware requirements. Instead of constructing a full Trie, which has high space complexity, we aggregate IP prefixes based on our previous observation on purity. We first construct a Trie at the /16 prefix level and examine the purity of each leaf node. Expansion stops if a node's purity surpasses a set threshold; otherwise, we expand until the node satisfies our purity standards or reaches the /32 prefix limit. A detailed algorithm is provided in Appendix §F, with further insights into threshold tuning available in Appendix §D. Even under /32 prefix granularity, requests of different network types are still observed. We speculate that the CDN only sees the public IP, which may be dynamically binded to various networks.

Empirical purity tuning. Tuning the aforementioned purity thresholds is crucial to model accuracy and space complexity. We adopt an empirical approach, starting with gathering online purity measurements, as shown in Figure 7. By analyzing the requests from the top 28 CDN nodes over one week, we find that only 8.6% of /24 prefixes have purity lower than 90%. By fine-tuning the threshold to aggregate IP addresses based on this observation and the available resources in each CDN node, the number of entries in the mapping cache is brought down from hundreds of thousands (*i.e.,* 10^5, the number of IP addresses seen by a CDN node in the last hour) to hundreds (*i.e.,* the number of /16 or /24 IP prefixes), resulting in a 614× memory saving without sacrificing prediction accuracy.

4 Experiences and Discussion

In this section, we discuss our deployment experiences and lessons learned in our real-world production CDN.

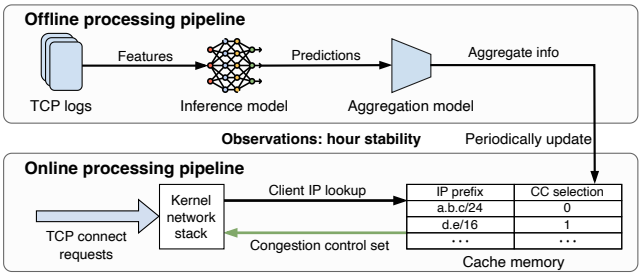


Figure 8: Inference pipeline at CDN nodes.

4.1 Deployment Experiences

ALICCS has served Alibaba Cloud CDN's production for over one year, supporting short video services for two major providers, spanning nearly all provinces in China. It reaches high network type prediction accuracy in around four hundred nodes of three main ISPs in China and Southeast Asia. We share some experiences in deploying ALICCS at scale.

Service-aware CCS. CDN requires tailored CCS strategies for different customers and services. This means that CCS can not be operated only in the kernel space where we are not aware of the service types and can not make customized decisions for different services. To better integrate with information from services and maintain flexibility, we decide to move AI processing mostly to user space.

TCPe (TCP extension) is our modular and customized TCP protocol stack implemented in the kernel space. Upon receiving each connection request from a client, it sends a query for the right CC decision to the mapping cache lookup using the client's IP address and then applies the query result of the CC algorithm accordingly in the kernel space. It also acquires TCP statistics from kernel space at a fine granularity, provided to the Log Server, which is responsible for collecting TCP statistics of CDN requests and send them to AI model for inference. We use the standard TCP socket's programming interface (*e.g.,* setsockopt) to set the CC algorithm.

AI Server, where the ML model resides, is not required to provide low latency, as it is decoupled from the online CCS service due to our designed Mapping Cache. We export the model trained in Pytorch in PMML format and load it into the AI server using the high-performance CPMML library. It continuously outputs predictions of network types using TCP logs from the Log Server, converts the predictions into CCs, and forwards them to the Aggregation Module.

4.2 Lessons Learned

We now share our lessons learned in deploying ALICCS.

What influences the QoE disparity of short video services under various CCs? Traditional CC design typically emphasizes maximizing throughput; however, our studies indicate that preventing frequent speed fluctuations is more critical, especially for short video services. Thus, while most CC mechanisms offer adequate throughput, optimizing *throughput stability*, a critical determinant of rebuffer rate perfor-

mance, emerges as essential for short video services. Beyond traditional network metrics such as RTT, loss, and bandwidth, our research also highlights the significant impact of network operators' traffic policies, such as rate limiting (refer to Observation III). Taking these policies into consideration is the key to optimal CC algorithm design. Furthermore, CCs customized for specific network types such as Wi-Fi and 4G, considering their unique physical characteristics, greatly enhance short video QoE (refer to Observation II).

Data cleaning. The massive number of CDN nodes requires careful data pre-processing via automatic data cleaning tools, augmented with our expertise in the patterns of TCP statistics and the prediction target, *i.e.,* network types. In our experience, data like RTT often incurs long-tail outliers that are extremely high but occur infrequently, thus violating the Gaussian assumption explicitly assumed by most deep-learning approaches, and impacting the efficiency of data normalization. Some abnormal RTT values can even skew the average RTT to seconds. Our current solution is to compress the data range by applying a logarithmic transformation (log), followed by standardization. This approach avoids the need for manually deciding the threshold to filter out outliers and significantly aids in accelerating model training.

Advantages of distributed inference for CDN networks. Our application of ML to CDNs indicates a distributed inference architecture is more beneficial for CDN networks. This approach offers two benefits: First, it enhances scalability by enabling edge nodes to independently serve requests, which alleviates bottlenecks and increases the network's capacity to manage high traffic volumes. Second, since CDN central servers often reserve processing power for video transcoding, offloading inference tasks to edges with surplus CPU cycles for peak demand is more cost-effective. In contrast, centralizing these tasks requires allocating an additional 2 CPU cores and 4 GB of memory to each of thousands of CDN nodes, resulting in significant expenses for CDN based on tests.

Enhanced gains of customized network services for IPv6. Our deployment experience shows that customizing network services using IP prefix aggregation reduces average rebuffer rates by 60% more in IPv6 networks than in IPv4 (7.6% vs 4.76%). This arises from two factors: First, IPv6 enhances predictability by reducing reliance on NAT and similar technologies used in IPv4. IPv6 network tends to assign a fixed IP prefix to a subnet over time, improving history-based predictions. Conversely, IPv4 with NAT may frequently change the mapping of IP prefix to underlying subnet, undermining these predictions. Second, the disparity in network characteristics between Wi-Fi and 4G is more pronounced in IPv6, with a ratio between mean retransmission rate of Wi-Fi over 4G being 2.1× in IPv4 and 2.9× in IPv6. These facts underscore the need for more tailored services to manage Wi-Fi and 4G disparities in IPv6 networks.

Deploying a fallback strategy for ML in CDN networks. When integrating ML system with CDNs, it is crucial to establish a robust fallback strategy when ML predictions are inaccurate. In practice, no ML system can perfectly address every scenario or maintain reliability in all situations. Therefore, it's essential to revert to a default strategy when ML predictions are uncertain. We employ a threshold-based scheme: if our ML-predicted network type has a logit exceeding 80% for Wi-Fi type and 60% for 4G type, it is trusted; otherwise, the system defaults to its original protocol set-up. The higher threshold for Wi-Fi reflects more cautious decision-making due to its large request proportions compared to 4G. This safeguard ensures that performance does not drop below the default configuration, thus enhancing reliability.

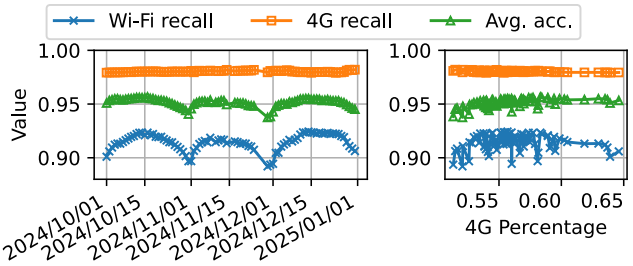


Figure 9: Model prediction accuracy varying date (left) and percentage of 4G traffic (right).

5 Evaluation

To ensure a smooth transition to full-scale deployment, we have focused our efforts on short video services. ALICCS has been applied to two major short video services on Alibaba Cloud CDN, with one of them contributing training data over 30% of our CDN nodes. We evaluate ALICCS's performance by addressing three key questions:

Q1: How accurate is our ML model in predicting the network types for large-scale, geo-distributed requests?

Q2: How efficient is our inference pipeline, particularly with the cache-based, online-offline decoupled design?

Q3: What's the QoE improvement for short video services using our production CDN?

5.1 Machine Learning Model Accuracy

Model prediction effectiveness. We validate our model's prediction accuracy across numerous CDN nodes handling high volumes of short video requests from major ISPs in China. To ensure statistical reliability, nodes with less than 5% of requests from 4G or Wi-Fi connections are excluded. Our ML model achieves an accuracy ranging from 95.8% to 99.0%, averaging an improvement of 1.58% and peaking at 3.4% on Wi-Fi networks of ISP #2. Remarkably, this is accomplished using only 30% CDN nodes for model training, confirming our model's generalizability.

Online validation of the ML model. Figure 9 presents our model's prediction accuracy in serving a large volume of short video requests over 3 months on a CDN node. The left figure depicts that our ML model achieves a remarkably high value of recall (*i.e.,* true positive rate of predicting each network

type), varying slightly between 89% to 93% for Wi-Fi, 97% to 98% for 4G, and an overall average ranging from 94% to 96%. Interestingly, the Wi-Fi recall exhibits a monthly cyclic pattern and remains stable over time. This phenomenon likely stems from ISP-enforced monthly 4G data caps, leading to a reduction in 4G usage as users hit their data limits towards the end of the month and a corresponding drop in Wi-Fi recall at the same times. This dependency is due to our model being trained on a balanced dataset (*i.e.,* 50% of the data is 4G), not reflecting the actual distribution of 4G usage at the beginning and end of the month. Previous studies, such as in reference [39], have shown that when the proportions of category (like 4G) shift, classification accuracies decrease. The impact on Wi-Fi recall is more noticeable due to its smaller absolute number. Additionally, our model prediction accuracies for both Wi-Fi and 4G are robust to varying percentages of 4G connections, as demonstrated in Figure 9 (right), confirming ALICCS's generalizability. We continuously monitored the model's performance by checking a domain name whose requests have been labeled by App #1 short video service provider. We see that the model maintains a high recall rate of over 90% for each network type for at least 6 months, from March 2024 to August 2024.

ML model prediction comparison. To evaluate the effectiveness of our ML model design, we conduct ablation studies, evaluating ALICCS's different components, and compare with Configanator, an advanced learning-based CCS scheme, and a baseline named Vanilla ML, where we directly feed training data into an MLP (multi-layer perceptron) network without extra generalization optimization. As shown in Figure 10, Configanator shows similar average performance as Vanilla ML but exhibits significantly more performance variance. This is due to Configanator's method of aggregating samples by network class, resulting in overfitting when the number of samples is small. Our GAN-based design significantly reduces variance across CDN nodes, an essential feature for large-scale deployments that demand reliability in worst-case scenarios. This is potentially due to our tailored regularizers in the loss functions, which leverage domain knowledge to boost model robustness by promoting learning from a universal set of features. For instance, incorporating RSC improves average performance by encouraging diverse feature use, and our regularizer $L_{variance}$ reduces feature noises, thereby improving average prediction accuracies.

5.2 Implementation Framework Efficiency

Superiority of our offline-online decoupled inference. In Table 3, we compare our decoupled online-offline inference method with the baseline, which executes *serial per-request online inference*. In the experiment, we set 32 Nginx workers to fully utilize available CPUs of 32 cores on the machine, emulating a CDN node at maximum QPS. The baseline method incurs a 10417ns processing delay and handles a maximum of 7.6k QPS only, while our system significantly reduces the processing delay to 162 ns and achieves 18.4k maximum QPS, representing 64.30× and 2.42× improvements respectively. Consequently, our decoupled framework drastically cuts AI inference delays and substantially increases QPS capacity, which is critical for managing high user demand efficiently.

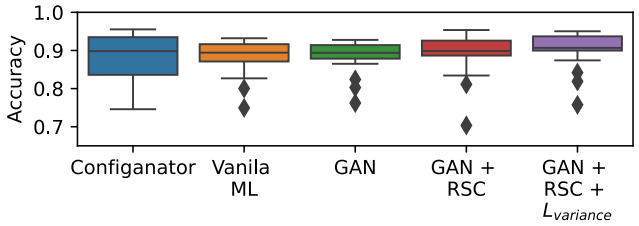


Figure 10: Ablation studies for evaluating different components of ALICCS and comparison with Configanator [32].

Table 3: Comparison with non-optimized online inference.

	Processing delay	Max. QPS
Baseline: online inference	10417 ns	7.6k
Online-offline decoupled	162 ns	18.4k
Improvement	64.30×	2.42×

CPU and Memory usage patterns. As Shown in Figure 11, the CPU overhead of ALICCS grows slowly with the number of connection requests, reaching a maximum of 200% at 17k QPS (refer to the left figure). This indicates our system requires no more than 2 cores, which is less than 1% (2 out of 256 cores) of the total CPU resources per CDN node, to handle peak traffic. Additionally, ALICCS's memory usage remains conservative, never exceeding 2.9GB (refer to the right figure), over a seven-day period in our measurements. These results underscore ALICCS's minimal resource impact on the overall CDN resource availability and its significant run-time efficiency. Additionally, our results also reveal interesting usage patterns in short video service: People tend to watch more short videos at noon, evidenced by the CPU usage being proportional to request volume and the 2 minor daily peak patterns; After a slight decrease early in the afternoon, viewship steadily climbs until reaching the peak at 20 o'clock.

5.3 QoE and Retransmission Improvement

In this section, we present the improvement in both QoE (rebuffer rate) and bandwidth cost (retransmission rate) achieved by ALICCS, serving the two major short video services, App #1 and App #2, in our Alibaba Cloud CDN.

Evaluation setup. To demonstrate the QoE improvements from deploying ALICCS, we conduct a month-long Randomized Control Test (RCT), comparing rebuffer rates with and without CCS. To approximate the random assignment of client connections to both versions, for each connection request, we apply the Cyclic Redundancy Check (CRC) [36] hash function on the client's IP address. Depending on whether the hash code is odd or even, the request is assigned to the system with our CCS strategy or the default system without CCS.

Baseline. The baseline system (default) uses CUBIC for all

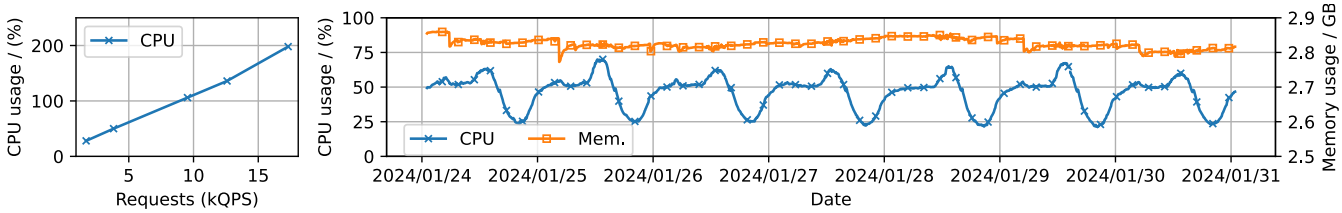


Figure 11: ALICCS's CPU overhead varying kQPS (left) and its seven-day usages of CPU and memory (right).

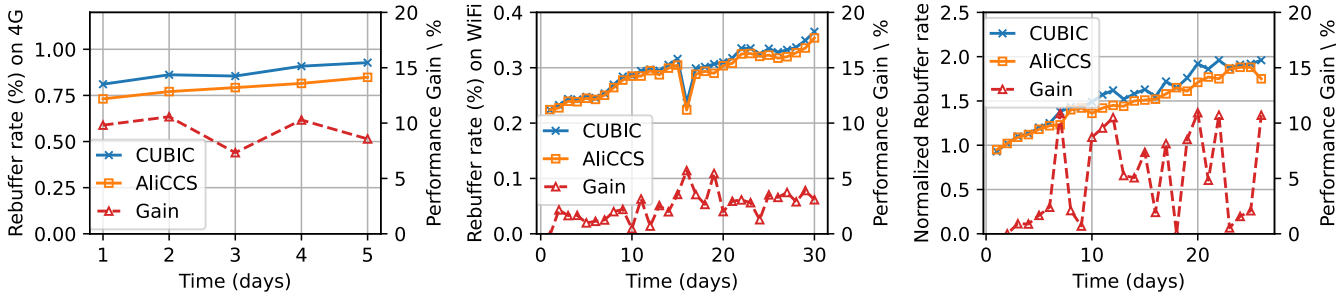


Figure 12: End-to-end rebuffer rate comparison of CUBIC and ALICCS.

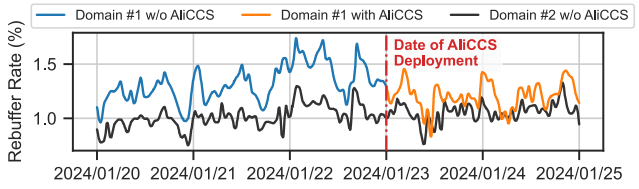


Figure 13: Rebuffer rates over time for two domain names with the same CDN nodes. The blue and orange curves represent the rebuffer rate for Domain #1 before and after ALICCS was applied. The black curve, associated with Domain #2, confirms stable CDN network conditions in this period of time.

requests given Wi-Fi's dominance in our CDN connections.

Real-world rebuffer rate improvement. As shown in Figure 12, ALICCS consistently achieves a lower end-to-end rebuffer rate, with improvements of 9.31% for 4G and 2.51% for Wi-Fi, along with a 4.76% gain across all connections *w.r.t.* deployment prior to ALICCS with all requests using CUBIC. This 5% reduction can boost our market edge in the fiercely competitive CDN industry, where even a 2%–3% difference in service quality matters, ensuring customer retention. Figure 13 zooms in on the results around the time when ALICCS was deployed, demonstrating the rebuffer rate reduction is attributed to the deployment of ALICCS.

Real-world retransmission rate improvement. As demonstrated in Figure 14, ALICCS achieves 59.24% improvement on App #1 and 61.28% on App #2, which translates into tens of million US dollars savings per year in terms of bandwidth cost for Alibaba Cloud CDN. To better understand the geographical diversity, we also investigated the performance gains across each province in China, as shown on the right side of Figure 14. ALICCS consistently achieves significantly lower retransmission rate in all provinces, with minimum, maximum, average, and standard deviation gains of 25.51%, 174.36%, 62.76%, and 36.28%, respectively. The results also lead to several interesting findings: Henan province tops the chart, possibly due to its high population density and heavy reliance on 4G networks; Performance gains in Guangdong province and Northeastern China exceed the average, in line with our observed widespread habit of streaming short videos on mobile devices in these areas. The smallest gain is seen in Qinghai province, likely attributed to inadequate 4G connectivity and lower population density.

5.4 Trace-driven Evaluation

Emulation setup. We also conduct an emulation study using short video traces collected over a day from one of Alibaba Cloud's highest-traffic CDN nodes. We compare ALICCS with two model-based CCS methods, Configanator [32] and Disco [47], both of which utilize the decision tree, as well as Pytheas [26], an online learning-based CCS (The rationale for the baseline selections is detailed in Appendix §E). We select CUBIC and BBR as CC candidates for evaluation due to their extensive testing, deployment, and worst-case assurance. Additionally, we construct the *best CCS* based on our trace, named Oracle, as a benchmark for CCS algorithm comparisons. We also define an aggregate QoE function, defined as a linear combination of rebuffer and retransmission rates, *i.e.*, $QoE = 1-\alpha \cdot rebuf - (1-\alpha) \cdot retrans$. Given our primary focus is on rebuffer rate over transmission costs, we set $\alpha = 0.8$ to prioritize minimizing stalls during short video streaming. Our emulation study employs a time-window-based mechanism [25, 27], where we construct unbiased estimates of CC's *outcomes*, *i.e.*, *QoE*, rebuffer rate, and retransmission rate metrics. For each /24 IP prefix group, we assign a time window for each CC to collect its corresponding outcomes. Oracle selects the CC with the best average *QoE* at the end of each window for each /24 IP prefix group. To simulate any

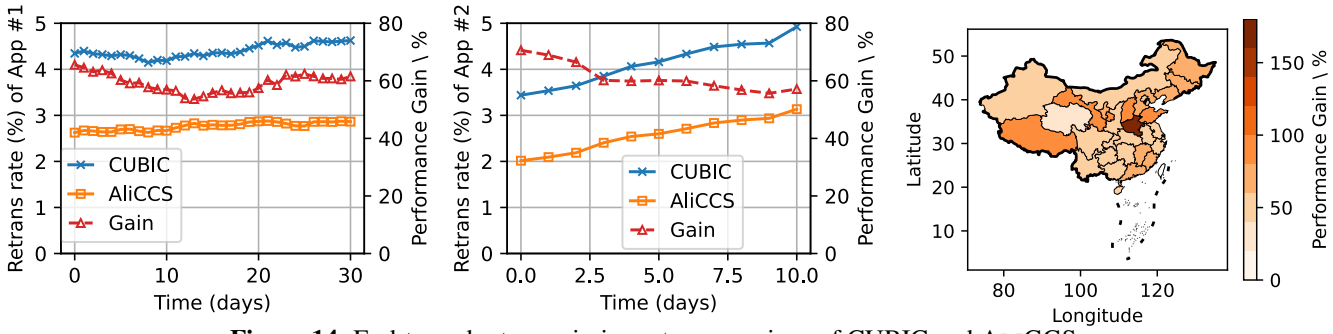
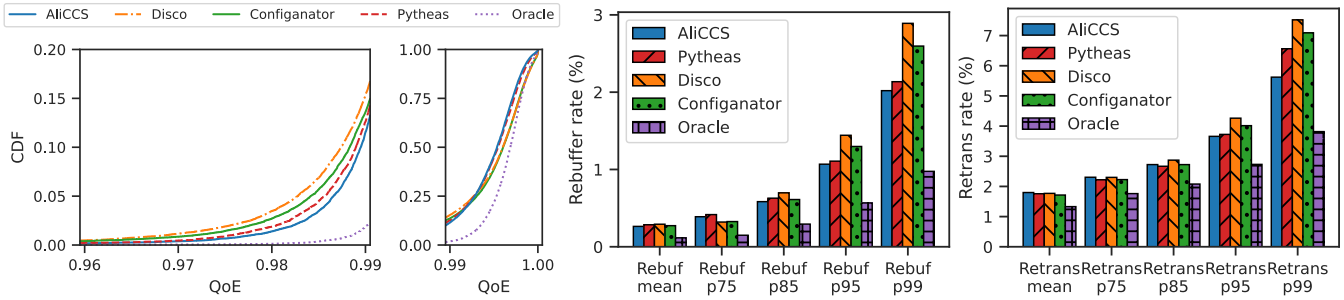


Figure 14: End-to-end retransmission rate comparison of CUBIC and ALICCS.



(a) QoE in good (right) and poor (left) network conditions. **(b)** Rebuffer rate. **(c)** Retransmission rate.

Figure 15: End-to-end improvement achieved by ALICCS in the trace-driven emulation.

algorithm's outcomes for CCs, including those that are not actually adopted by the real-world policy at a given timestamp in our trace, we use the average outcomes at historical times where the same CC is chosen under the same IP prefix group and network type.

Comparison with CCS state-of-the-arts. As shown in Figure 15a, all algorithms perform similarly in high QoE regions (right), *i.e.,* good network conditions where CC choice has minimal impact. In contrast, ALICCS performs notably well in low QoE regions (left), *i.e.,* impaired conditions. For QoE metrics like rebuffer rate, samples from poor network conditions disproportionately affect the average, underscoring ALICCS's advantage in low QoE regions. We then compare baseline performance under poor network conditions. Figure 15b shows that ALICCS *achieves a 4.54% to 30.80% lower rebuffer rate in the 85th percentile and beyond.* In contrast, Configanator and Disco exhibit higher rebuffer rates, as they overfit to good network conditions, neglecting the importance of edge cases with poor network scenarios. Meanwhile, Pytheas performs sub-optimally due to the inherent exploration required during network probing. Figure 15c shows the retransmission rate results. While ALICCS slightly increases retransmissions to reduce rebuffering, it outperforms Configanator, Disco, and Pytheas in the long-tail distribution, achieving a 1.74% to 25.24% lower retransmission rate. While Pytheas achieves comparable performance to ALICCS, estimating its reward function (*e.g.,* rebuffer rates) in real-time poses significant challenges and could incur additional computational overhead compared to ALICCS.

6 Related Work

Congestion control (CC) is a longstanding and significant topic, featuring a multitude of meticulously designed CCs, including CUBIC [20], BBR [12], and Reno [23].

Automatic CC Configuration. Recent studies propose machine-generated CCs and automatic parameter adjustments, highlighting that handwritten CC policies are effective only in limited contexts. Remy [45] utilizes offline optimization to generate a new CC by searching from a certain design space, followed by Copa [9] with more theoretical insights. CherryPick [8] employs Bayesian Optimization, while CFA [25] exploits domain-specific insights. Such methods fail to leverage large-scale data available on CDNs, potentially limiting their accuracy and generalization performance.

Learning-based CC parameter tuning. To increase generalizability, ML has been adopted to adjust CC parameters. For instance, Orca [6], Aurora [24], Libra [15], and TCP-RL [33] adopts RL with handcrafted reward functions for learning parameters like congestion windows, sending rates, etc. A few others adopt online learning but generally favor scenarios with i.i.d. network and/or CC characteristics. For example, Pytheas [26] applies multi-armed bandits (MAB), Configanator [32] integrates MAB with decision trees, and PCC vivace [14] utilizes gradient-based online learning. Recent works also adopt imitation learning, *e.g.,* DuGu [21] and Muses [51]. Such methods often overlook crucial worst-case scenarios in production CDN. ALICCS, instead, uses GAN-based model to balance performance across scenarios and

tackles the often-overlooked inference latency.

Learning-based CCS. Recent works have advocated CCS, *i.e.,* selecting the best CC among existing schemes. Specifically, Rein [13] uses data-driven methods to mitigate the CC transitioning overhead. Disco [47] and Antelope [52] leverage RL-inspired reward functions, demonstrating the advantage of CCS over CC tuning. OPSBC [50] combines traditional flow scheduling methods with MAB-based CCS. Sage [49] employs offline DRL to learn a new CC from traces, despite the complexity in feedback collection and high training cost due to a large state-action space. DPO [1] from Akamai represents an industrial effort to optimize CCS. Unlike ALICCS, these works do not present large-scale network experimental results, nor do they incorporate CDN domain knowledge.

Domain adaptation aims to preserve model performance on a target domain with limited data by applying knowledge from a source domain, and has been applied in various networking problems. For instance, Genet [46] applies curriculum learning for network adaptation. AWARE [37] uses meta-learning and bootstrapping for cloud auto-scaling. Other works leverage knowledge from simulation settings [41] or smaller-scale networks [44]. These works shed light on our work, albeit from a high-level perspective. We instead adopt GAN and Causal Graphical Model (CGM), tailoring our design specifically for short video workloads and production CDNs. Our solution is deeply rooted in CDN-specific domain knowledge.

7 Discussion

Do the observations driving ALICCS' design remain valuable beyond the scope of short video services? We believe most of our measurement insights extend beyond short videos and can be significant for the broader community. Notably, Observations I, III, and IV in §2.3—highlighting the variability in CCs' advantages, the impact of rate limiting, and the dynamics of network types—apply in general and inform further research on intelligent CCS design for diverse services. Additionally, our lessons in §4.2 on data cleaning, distributed inference, IPv6 benefits, and the fallback strategy provide insights for designing ML-based CDN systems. However, Observation II on correlation between network types and the best CCS and the lesson on throughput stability, are more specific to short video contexts, which is likely tied to the low traffic volume of short videos and the dependency on rebuffer rate for QoE.

Can ALICCS' design be effectively applied outside of short video domain or Alibaba Cloud's CDN? Our GAN-based model (§3.1), which learns uniform feature representations across IP prefixes, can be applied to CCS prediction for any service, provided that there is sufficient A/B test data with relevant labels (*i.e.,* best CCS under an IP prefix). The interpretation technique (§3.2) can aid model debugging of any classification task. Moreover, the distributed inference pipeline (§3.3) should be widely applicable to many ML-based CDN systems to reduce latency with a few assumptions: the input/output features of ML model are temporally stable and similar under the IP prefix, which are commonly adopted in existing works [26, 32]. However, it is noteworthy that, as rate limiting affects CCS (§2.3), ISP policy differences may impact ALICCS' performance gains in other networks.

Does network contention among clients degrade ALICCS' prediction accuracy? ALICCS uses the GAN model to achieve balanced accuracy across diverse conditions, including contention. In fact, ALICCS outperforms existing works by reducing accuracy variance and 95th percentile rebuffer rate, likely associated with impaired conditions (see Figures 10 and 15). However, ALICCS does not strictly optimize accuracy uniformity, as we find this could degrade overall accuracy significantly. Instead, we fine-tune λ_1 in the training loss to balance accuracy uniformity and overall accuracy.

ALICCS's reliance on pre-deployment test and domain knowledge: The success of framing CCS as an ML classification task relies heavily on comprehensive pre-deployment tests and domain knowledge to identify a strong link between network types and best CCS. This framework eliminates the need for acquiring short video QoE training data directly from content providers. However, it is not always straightforward, for a new scenario, to establish a clear connection between model predictions and desired outcomes. Our approach, despite avoiding continuous observations and handcrafted reward functions like in RL approaches, demands deep domain knowledge and insightful observations.

8 Conclusion

In this paper, we introduce ALICCS, the first production-optimized CCS approach that has served Alibaba Cloud CDN for over one year. ALICCS exploits our insights from large-scale measurements that reveal contrasting CC performances for short video services in varying network access types. It employs a domain-specific ML model for scalability and generalizability across numerous CDN nodes and changing network conditions, with an efficient inference pipeline for runtime performance. Extensive results from our production CDN demonstrate that ALICCS achieves significant retransmission rate reduction and enhanced QoE for short video services, outperforming advanced ML approaches and the default rule-based CC. We plan to explore combining CCS with protocol parameter tuning in the future and assess whether this integration can yield further QoE improvements.

Acknowledgments

We thank our shepherd, Soheil Abbasloo, and the anonymous reviewers for their insightful comments. This work is supported by NSFC under grant 62472460, Guangdong Basic and Applied Basic Research Foundation under grants 2024A1515010161 and 2023A1515012982, Young Outstanding Award under the Zhujiang Talent Plan of Guangdong Province, and the Alibaba Research Intern Program. Xiaoxi Zhang and Ennan Zhai are the co-corresponding authors.

References

[1] DPO from Akamai. https://www.linkedin.com/pulse/need-dynamic-protocol-optimization-ant-bao, 2020.

[2] MaxMind GeoIP database. https://www.maxmind.com/en/home, 2020.

[3] information Gain from wiki. https://en.wikipedia.org/wiki/Information_gain_(decision_tree), 2024.

[4] The Evolution of CDN Technology in 2024. https://blog.blazingcdn.com/en-us/cdn/the-evolution-of-cdn-technology-in-2024, 2024.

[5] What are QUIC and HTTP/3? https://www.f5.com/glossary/quic-http3, 2024.

[6] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 632–647, 2020.

[7] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: Auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 44–58, 2018.

[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, volume 2, pages 4–2, 2017.

[9] Venkat Arun and Hari Balakrishnan. Copa: Practical {Delay-Based} congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, 2018.

[10] Hirochika Asai and Yasuhiro Ohara. Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup. *ACM SIGCOMM Computer Communication Review*, 45(4):57–70, 2015.

[11] Lawrence S Brakmo, Sean W O'malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[12] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

[13] Kefan Chen, Danfeng Shan, Xiaohui Luo, Tong Zhang, Yajun Yang, and Fengyuan Ren. One rein to rule them all: A framework for datacenter-to-user congestion control. In *4th Asia-Pacific Workshop on Networking*, pages 44–51, 2020.

[14] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. {PCC} vivace:{Online-Learning} congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.

[15] Zhuoxuan Du, Jiaqi Zheng, Hebin Yu, Lingtao Kong, and Guihai Chen. A unified congestion control framework for diverse application preferences and network conditions. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 282–296, 2021.

[16] Sishuai Gong, Usama Naseer, and Theophilus A Benson. Inspector gadget: A framework for inferring tcp congestion control algorithms and protocol configurations. In *Network Traffic Measurement and Analysis Conference*, 2020.

[17] Ali Gouta, David Hausheer, Anne-Marie Kermarrec, Christian Koch, Yannick Lelouedec, and Julius Rückert. Cpsys: A system for mobile video prefetching. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 188–197. IEEE, 2015.

[18] Carlo Augusto Grazia, Natale Patriciello, Toke Høiland-Jørgensen, Martin Klapez, and Maurizio Casoni. Aggregating without bloating: Hard times for tcp on wi-fi. *IEEE/ACM Transactions on Networking*, 30(5):2359–2373, 2022.

[19] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[20] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[21] Tianchi Huang, Chao Zhou, Lianchen Jia, Rui-Xiao Zhang, and Lifeng Sun. Learned internet congestion control for short video uploading. In *Proceedings of the 30th ACM International Conference on Multimedia*, pages 3064–3075, 2022.

[22] Zeyi Huang, Haohan Wang, Eric P Xing, and Dong Huang. Self-challenging improves cross-domain generalization. In *Computer vision–ECCV 2020: 16th European conference, Glasgow, UK, August 23–28, 2020, proceedings, part II 16*, pages 124–140. Springer, 2020.

[23] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329, 1988.

[24] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059. PMLR, 2019.

[25] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. {CFA}: A practical prediction system for video qoe optimization. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 137–150, 2016.

[26] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *NSDI*, volume 1, page 3, 2017.

[27] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[28] Stan Lipovetsky and Michael Conklin. Analysis of regression in game theory approach. *Applied Stochastic Models in Business and Industry*, 17(4):319–330, 2001.

[29] Ge Ma, Zhi Wang, Minghua Chen, and Wenwu Zhu. Aprank: Joint mobility and preference-based mobile video prefetching. In *2017 IEEE International Conference on Multimedia and Expo (ICME)*, pages 7–12. IEEE, 2017.

[30] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[31] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

[32] Usama Naseer and Theophilus A Benson. Configanator: A data-driven approach to improving cdn performance. In *19th USENIX NSDI Symposium*, 2022.

[33] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[34] Stefan Nilsson and Gunnar Karlsson. Ip-address lookup using lc-tries. *IEEE Journal on selected Areas in Communications*, 17(6):1083–1092, 1999.

[35] J Pearl. Causality: Models, reasoning, and inference 47cambridge university presscambridge, united kingdom. pearl, j. 2000. *Causality: models, reasoning, and inference*, 47, 2000.

[36] William Wesley Peterson and Daniel T Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[37] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. {AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 387–402, 2023.

[38] Miguel Á Ruiz-Sánchez, Ernst W Biersack, and Walid Dabbous. Survey and taxonomy of ip address lookup algorithms. *IEEE network*, 15(2):8–23, 2001.

[39] Marco Saerens, Patrice Latinne, and Christine Decaestecker. Adjusting the outputs of a classifier to new a priori probabilities: a simple procedure. *Neural computation*, 14(1):21–41, 2002.

[40] MY Sanadidi, Claudio Casetti, Mario Gerla, Saverio Mascolo, and Ren Wang. Tcp westwood: End-to-end congestion control for wired/wireless networks. *Wireless Networks*, 8(5):467–479, 2002.

[41] Junyang Shi, Mo Sha, and Xi Peng. Adapting wireless mesh network configuration from simulation to reality via deep learning based domain adaptation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 887–901, 2021.

[42] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. Evaluating quic performance over web, cloud storage, and video workloads. *IEEE Transactions on Network and Service Management*, 19(2):1366–1381, 2021.

[43] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[44] Shun Tobiyama, Bo Hu, Kazunori Kamiya, and Kenji Takahashi. Large-scale network-traffic-identification method with domain adaptation. In *Companion Proceedings of the Web Conference 2020*, pages 109–110, 2020.

[45] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.

[46] Zhengxu Xia, Yajie Zhou, Francis Y Yan, and Junchen Jiang. Genet: automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, 2022.

[47] Furong Yang, Zhenyu Li, Jianer Zhou, Xinyi Zhang, Qinghua Wu, Giovanni Pau, and Gaogang Xie. Disco: A framework for dynamic selection of multipath congestion control algorithms. In *2023 IEEE 31st International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2023.

[48] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. Guarantee ip lookup performance with fib explosion. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 39–50, 2014.

[49] Chen-Yu Yen, Soheil Abbasloo, and H Jonathan Chao. Computers can learn from the heuristic designs and master internet congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 255–274, 2023.

[50] Xiaoxi Zhang, Siqi Chen, Yunfan Zhang, Youngbin Im, Maria Gorlatova, Sangtae Ha, and Carlee Joe-Wong. Optimal network protocol selection for competing flows via online learning. *IEEE Transactions on Mobile Computing*, 22(8):4822–4836, 2023.

[51] Zhiren Zhong, Wei Wang, Yiyang Shao, Zhenyu Li, Heng Pan, Hongtao Guan, Gareth Tyson, Gaogang Xie, and Kai Zheng. Muses: Enabling lightweight learning-based congestion control for mobile devices. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 2208–2217. IEEE, 2022.

[52] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. Antelope: A framework for dynamic selection of congestion control algorithms. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

Appendices

Appendices are supporting material that has not been peer-reviewed.

A Methodology to Conduct A/B Test Comparing CUBIC and BBR

To conduct a fair A/B test to compare performance of CUBIC and BBR in short video services, for each new connection from a specific partner app, we hash the request's IP using CRC algorithm to randomly map it to either the CUBIC or BBR group, ensuring a fair workload and network environment. The measurement period lasted around two weeks, with average data collected on a regional and daily basis. We have checked and confirmed the consistency of the results over this period. To further valid the difference we observe is due to CC's assignment rather than inherent difference between the group, we have also reversed the A/B assignment (IP assigned to group A now assigned to group B and vice versa) for about another two weeks, and the results are consistent with the first test, which validates the fairness of A/B test and the difference we observed is due to CC's assignment.

B Input Features of Our ML Model

Our CDN network collects several dozens of metrics to support a wide range of applications and services. Among these metrics, we carefully pick twelve of them as input features for our ML model, as shown in Table 4.

Table 4: Input Features for the ML Model

Feature	Description
mss	TCP maximum segment size
connection_time	TCP handshake time
init_snd_wnd	receiver initially advertised TCP window
min_rtt	minimum RTT in a TCP connection
max_snd_wnd	TCP receiver advertised maximum window size
min_snd_wnd	TCP receiver advertised minimum window size
min_rtt_variance	min RTT variance
max_rtt_variance	max RTT variance
min_rto	minimum retransmission timeout
min_srtt	max smoothed version of RTT
max_srtt	max smoothed version of RTT
max_cwnd	maximum congestion window of a TCP sender

C Handle Non-1:1 Mapping between Connections and Network Types

When there is a connection migration from one network type to another, there could be non-1:1 mapping between Connections and Network Types. We dicuss potential solutions to it. In case the migration is transparent to CDN server (e.g., the migration is behind a mobile proxy server), we find changing the CC for a live connection is a heavy engineering task requiring significant kernel modifications, and such cases are in minority. So we decide they will not be handled by ALICCS. In case the migration is visible to CDN server, like in case where QUIC or MPTCP is applied, ALICCS can be applied independently for each sub-connection, leveraging two IPs to find the best CC for each sub-connection in its cache.

D Hyper Parameter Tuning and Experiences

We report below some details on the engineering decisions and parameter choices. We explain as well the key considerations and the rationale behind our decisions.

Hyper parameters in the loss function for DL model training. We aim to strike a balance between reducing the variability of prediction performance across CDN nodes and maximizing overall average prediction accuracy. Our experiments show that increasing the parameters λ_1, λ_2 encourages the model to extract more uniform representations, thereby reducing the variability of prediction accuracy across CDN nodes. However, strictly minimizing variability could hurt the over all average accuracy. To address this, we perform a grid search to determine the values of λ_1 and λ_2, achieving minimal variability of accuracy without significantly compromising overall accuracy.

Other hyperparameters in DL model design. While the best hyperparameter configurations provided in existing literature on GANs can serve as a a starting point, we observe that they do not deliver the expected performance boost in our scenario. This likely reflects domain differences between areas like computer vision and networking. Consequently, we rely on performing a grid search to obtain the best-performing hyperparameter set for AliCCS, tailored to its unique requirements.

Inference pipeline at CDN Nodes. To balance computational efficiency and prediction accuracy, we opted to decide update the cache on an hourly basis. For our IP prefix aggregation strategy, we employ differentiated thresholds: For IP prefixes with high request volumes, we set high thresholds to make the conditions more strict, in order to avoid incorrect predictions affecting a large number of requests. For IP prefixes with low request volumes, we apply low thresholds to encourage aggregation, reducing memory usage without significantly impacting prediction accuracy.

E Method Selection Rationale for Trace-driven Evaluation

In our trace-driven evaluation, several state-of-the-art methods were not selected for the following reasons. Adapting Sage [49], originally designed for new CC learning to a CCS scheme, presents challenges due to its inefficiencies for production CDNs. The impracticality stems from the extensive training data required, such as rebuffer rates and bandwidth costs, across various state configurations from short video providers, which is not applicable for us as a CDN provider.

Algorithm 1 Prediction Results Aggregation Algorithm

Input: Pairs of IP and predicted cc: $\{(IP_i, pred_res_i)\}_{i=1}^{m}$
Output: $hashtable : IP_prefix \rightarrow (pred_res, purity)$
Input Parameters: p_thresh_16, p_thresh_24,
// step1: aggregate at prefix /16
Initialize $D \leftarrow \{(IP_i, pred_res_i)\}_{i=1}^{m}$
$H \leftarrow$ AGGREGATE_AT_PREFIX_LEN$(D, 16)$
// step2: for prefix without sufficient purity, expand at prefix /24
$D_expand_24 \leftarrow \emptyset$
for all prefix_16 in $H.keys()$ **do**
    if $H[prefix_16].purity < p_thresh_16$ **then**
        $D_tmp \leftarrow$ { $IP_i, pred_res_i \mid IP_i$ matches prefix_16 }
        $D_expand_24 \leftarrow D_expand_24 \cup D_tmp$
        remove prefix_16 from H
    end if
end for
// step3: for prefix without sufficient purity, expand at prefix /32
$H_24 \leftarrow$ AGGREGATE_AT_PREFIX_LEN$(D_expand_24, 24)$
$D_expand_32 \leftarrow \emptyset$
for all prefix_24 in $H_24.keys()$ **do**
    if $H_24[prefix_24].purity < purity_thresh_24$ **then**
        $D_tmp \leftarrow$ (find all data under prefix_24)
        $D_expand_32 \leftarrow D_expand_32 \cup D_tmp$
        Delete prefix_24 from H_24
    end if
end for
$H_32 \leftarrow$ AGGREGATE_AT_PREFIX_LEN$(D_expand_32, 32)$
// merge the 3 hash tables
$H \leftarrow H \cup H_24 \cup H_32$
return H

procedure AGGREGATE_AT_PREFIX_LEN(D, $prefix_len$)
    Input: D: a subset of Pairs of IP and prediction result
        $prefix_len$: the length of prefix to aggregate at
    Output: $Map : IP_prefix \rightarrow (pred_res, purity)$
    Initialize $H \leftarrow \emptyset$ ▷ a 2-level hashtable
    for all $(IP, pred)$ in D **do**
        $ip_prefix \leftarrow$ COMPUTE_PREFIX$(IP, prefix_len)$
        $H[ip_prefix][pred] \leftarrow H[ip_prefix][pred] + 1$
    end for
    Initialize $H_res \leftarrow \emptyset$
    for all ip_prefix in $H.keys()$ **do**
        $pred_label \leftarrow \arg\max_{pred}(H[ip_prefix][pred])$
        $purity \leftarrow (fraction of pred_label under ip_prefix)$
        $H_res[ip_prefix].pred_res \leftarrow pred_res$
        $H_res[ip_prefix].purity \leftarrow purity$
    end for
    return H_res
end procedure

Both Disco [47] and Antelope [52] employ tree-based ML models for CCS prediction, with Antelope permitting mid-flow CC switching. However, in our trace-driven emulation where CCS is allowed only on a per-flow basis, the two approaches become nearly identical. Therefore, we implement only Disco as a representative model. Rein [13] was excluded from our comparative study due to its closed-source nature and its heuristic rules not being tailored for our short video service context.

F Prefix Aggregation Algorithm

We illustrate the prefix aggregation algorithm with a flow chart shown in Figure 16 as well as with Algorithm 1. Initially, we aggregate IP prefixes at the /16 level using a Trie and then assess the purity of this aggregation. Purity is defined as the ratio of the number of IP addresses with the most common label to the total number of IP addresses within the range. A high purity indicates that using the most common label to represent all IPs within the range results in minimal accuracy loss. If the purity is insufficient, we further refine the aggregation to /24 and /32 prefixes, as detailed in Steps 2 and 3 of the algorithm.

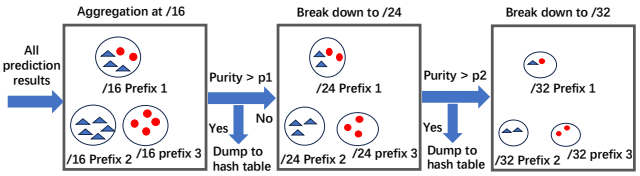
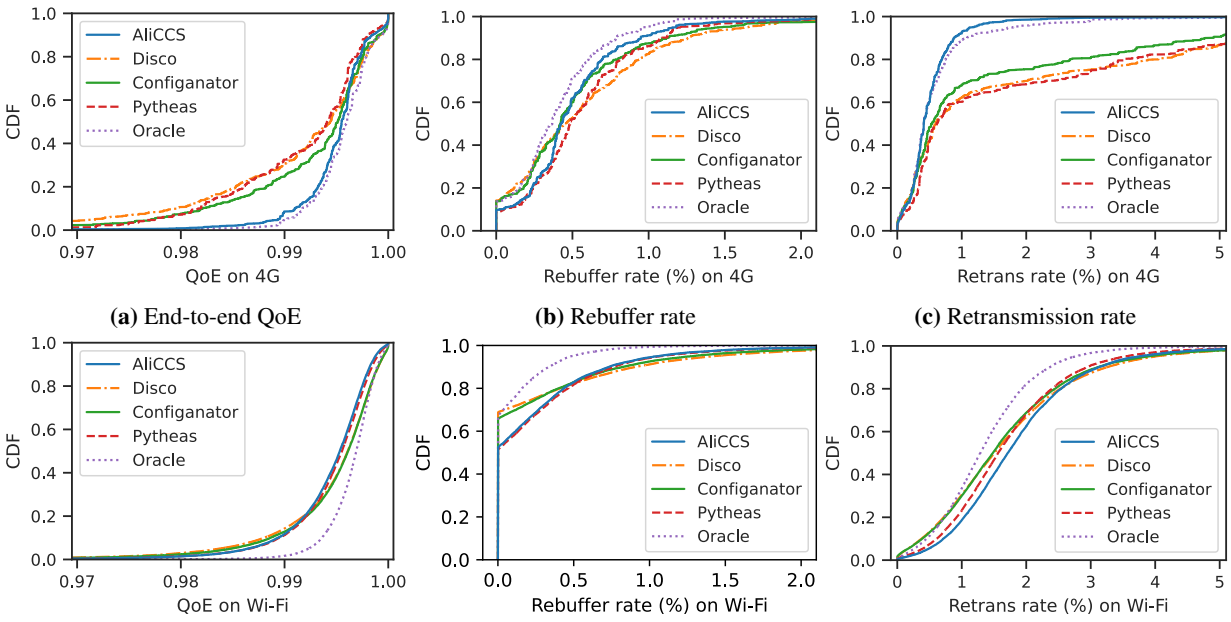


Figure 16: Trie-based prefix aggregation algorithm.

G End-to-End Performance

Figure 17 shows the end-to-end performance comparison of WiFi connections and 4G connections in terms of QoE, rebuffer rate, and retransmission rate.



(a) End-to-end QoE **(b)** Rebuffer rate **(c)** Retransmission rate

(d) End-to-end QoE **(e)** Rebuffer rate **(f)** Retransmission rate

Figure 17: The enhancement of ALICCS in trace-driven emulation is shown in terms of QoE, rebuffer rate, and retransmission rate. Figures 17a–17c show the results for 4G connections, while Figures 17d–17f show the results for Wi-Fi connections.

GPU-Disaggregated Serving for Deep Learning Recommendation Models at Scale

Lingyun Yang[†], Yongchen Wang, Yinghao Yu, Qizhen Weng[†], Jianbo Dong, Kan Liu,
Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi,
Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, Wei Wang[†]
[†]*Hong Kong University of Science and Technology* *Alibaba Group*

Abstract

Online recommender systems use deep learning recommendation models (DLRMs) to provide accurate, personalized recommendations to improve customer experience. However, efficiently provisioning DLRM services at scale is challenging. DLRMs exhibit distinct resource usage patterns: they require a large number of CPU cores and a tremendous amount of memory, but only a small number of GPUs. Running them in multi-GPU servers quickly exhausts the servers' CPU and memory resources, leaving a large number of unallocated GPUs stranded, unable to utilize by other tasks.

This paper describes Prism, a production DLRM serving system that eliminates GPU fragmentation by means of resource disaggregation. In Prism, a fleet of CPU nodes (CNs) interconnect with a cluster of heterogeneous GPU nodes (HNs) through RDMA, leading to two disaggregated resource pools that can independently scale. Prism automatically divides DLRMs into CPU- and GPU-intensive subgraphs and schedules them on CNs and HNs for disaggregated serving. Prism employs various techniques to minimize the latency overhead caused by disaggregation, including optimal graph partitioning, topology-aware resource management, and SLO-aware communication scheduling. Evaluations show that Prism effectively reduces CPU and GPU fragmentation by 53% and 27% in a crowded GPU cluster. During seasonal promotion events, it efficiently enables capacity loaning from training clusters, saving over 90% of GPUs. Prism has been deployed in production clusters for over two years and now runs on over 10k GPUs.

1 Introduction

Personalized recommender systems are the key infrastructure for many user-facing, revenue-generating web services, such as content streaming, e-commerce, social networks, and web search [12, 21, 47, 58]. These systems use deep learning recommendation models (DLRMs) to provide accurate, personalized recommendations to improve customer experience and increase user engagement. DLRM serving consumes the majority of the inference resources in today's AI cloud, with top recommendation models account for more than 79% of AI cycles, according to Meta [26].

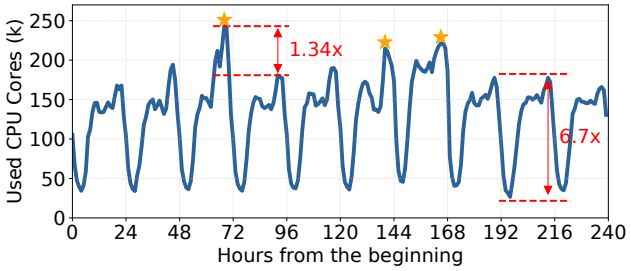


Figure 1: The CPU demands of DLRM services exhibit *daily* and *seasonal* variations; GPU demands follow the same trend. Trace collected from a production cluster, including three (starred) e-commerce promotion events.

However, serving DLRMs at scale faces aggravating challenges. DLRM serving has a stringent latency service-level objective (SLO), usually on the scale of tens of millisecond per request. In the meantime, DLRM serving needs to handle frequent spikes in demand. Meeting the latency SLOs often means provisioning for the peak load, which can be significantly higher than the average [28, 39]. Figure 1 illustrates the resource demands of Alibaba's DLRM services in a production cluster. We observe a distinct diurnal pattern with the peak-to-valley ratio over 6×; during seasonal promotion events, the peak load can be 1.3× higher than the regular peaks, which is in line with the previous report [68]. Provisioning for the peak load at such scale results in significant underutilization, making it economically unviable.

To reduce overprovisioning, a better strategy is to provision for the average load and enable *capacity loaning* during load spikes. Large companies like Alibaba own multiple purpose-specific infrastructures: some for training and the others for inference. When DLRM serving is in peak hours, it can temporarily loan GPU servers from training clusters as training jobs are less latency-sensitive and can tolerate interruptions. However, there is a mismatch between server configuration and DLRM's resource demand. Unlike training tasks that demand extensive GPU cycles, recommendation models exhibit low compute-intensity and are not bottlenecked on GPU. Instead, they perform sparse computations such as embedding [36, 37], which requires abundant memory for storing embedding tables and many CPU cores for table look-up and

pooling operations [26, 32]. As a result, running recommendation models in training servers quickly exhausts the servers' CPU and memory resources, leaving a large number of unallocated GPUs stranded. In our cluster, a typical DLRM service requests 48 CPUs and 1 GPU, while training servers commonly have ⟨96 CPUs, 8 GPUs⟩ (Figure 4). Deploying two DLRM inference instances occupies all CPUs on the host, leaving 6 unallocated GPUs unable to utilize by other tasks.

This paper presents Prism, a production DLRM serving system that addresses the resource mismatch problem with *GPU disaggregation*. Prism runs on a shared infrastructure where a fleet of CPU nodes (CNs) interconnect with a cluster of heterogeneous GPU nodes (HNs) through a high-speed RDMA network. Each CN has a large number of CPU cores and high memory but no GPU, while each HN is a typical training server with multiple GPUs but only a modest amount of CPU and memory resources. This infrastructure breaks down a cluster of monolithic servers with fixed configurations into two *disaggregated resource pools*, where CNs provide rich CPU and memory resources while HNs provide abundant GPUs. The two resource pools can be independently scaled to match the changing demands of dynamic workloads. Given a DLRM, Prism automatically divides its compute graph into two subgraphs, one containing CPU- and memory-intensive operators and the other GPU-intensive. The system then schedules the two subgraphs on the selected CN and HN for disaggregated serving and returns the results to users.

This paper describes the challenges, techniques, and lessons learned in building a disaggregated DLRM system at a production scale. First, the disaggregation needs to be *transparent to model owners*. Manually re-architecting models to a disaggregated version raises the risk of accuracy degradation and requires extra efforts from their owners, thus undesirable. Second, the system should *scale to thousands of servers* to handle excessive load spikes. Given the surging traffic, it should promptly schedule workloads to a large fleet of servers to achieve dramatic total throughput in a short period of time. Third, the system should *meet the stringent latency SLOs* of DLRM serving, in the presence of non-trivial communication overhead between CNs and HNs due to GPU disaggregation. Prism tackles these challenges with three major components: a disaggregation-optimized real-time prediction (RTP) framework that optimally partitions compute graphs between CNs and HNs (§4.1), a topology-aware resource manager that minimizes inter- and intra-server communication overhead (§4.2), and SLO-aware communication scheduling that ensures disaggregated serving within the target latency SLOs (§4.3).

Prism has been pilot deployed in late 2022 as the underlying infrastructure for a small number of production online recommendation services, with serving scale continuously increasing over the past two years. As of January 2025, Prism runs on over 10k GPUs, successfully decoupling the CPU and GPU computation demands of DLRMs without impacting service performance. Evaluations show that in daily high-allocation GPU clusters, Prism effectively reduces CPU fragmentation by 53% and GPU fragmentation by 27%; during seasonal promotion events, Prism efficiently enables capacity loaning from training clusters, saving over 90% of GPUs (§5). Our primary contributions are as follows.

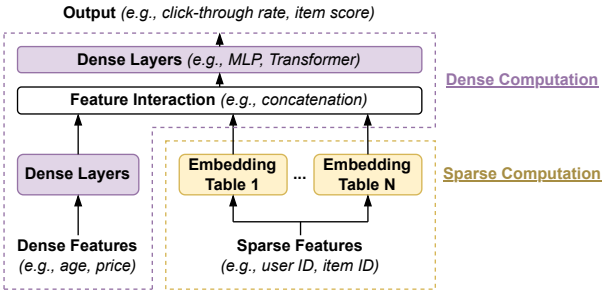


Figure 2: An architecture overview of DLRM.

- We identify the challenges in resource provisioning when deploying elastic DLRM services at a production scale and motivate the need for GPU-disaggregated serving.
- We design and implement Prism to harvest resources from CPU nodes and heterogeneous GPU nodes by means of disaggregated serving, alleviating the resource mismatch between the server configurations and DLRMs' resource demands, while still meeting latency SLOs.
- We evaluate Prism in production and demonstrate that it can effectively reduce resource fragmentation without compromising service performance, enabling efficient capacity loaning during promotion events.

We have released a production DLRM serving trace[1] as part of the Alibaba cluster trace program [3].

2 Background and Motivation

2.1 A Primer on DLRM

Modern recommendation models have a large feature set, separated into dense and sparse categories. Dense features, such as vectors and matrices, are processed by typical deep neural network layers, while sparse features are processed by indexing large embedding tables. Figure 2 illustrates a typical architecture of a DLRM [12, 47, 69]. The input comprises dense continuous features (e.g., user age and item price) and sparse categorical features (e.g., user ID and item ID). Sparse features are transformed into continuous embedding representations via a lookup operation in the embedding tables. These features are then fused and fed into a densely connected deep network, such as a multi-layer perceptron [12] or transformer [57], for prediction, eventually generating the model output, such as click-through rate or item score.

[1] https://github.com/alibaba/clusterdata/tree/master/cluster-trace-gpu-v2025

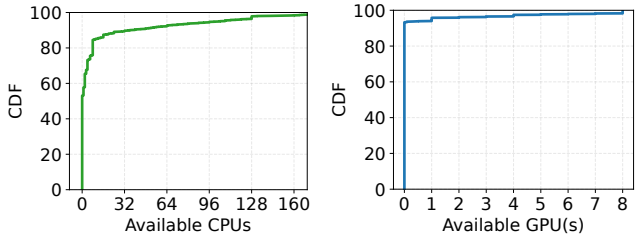


Figure 3: Resource allocation snapshot of a large-scale GPU cluster H with over 4k nodes, 640k CPU cores, and 11k GPUs.

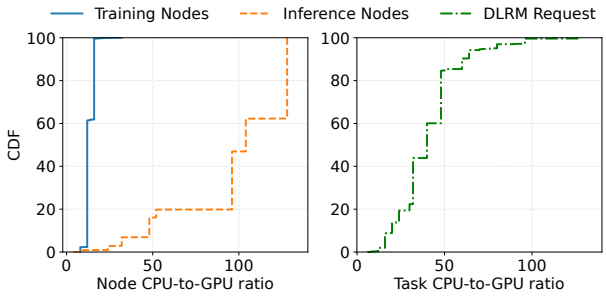


Figure 4: CDF of CPU-to-GPU ratio. The DLRM trace was collected over a period of one week from our production clusters, comprising of more than 25k instances.

DLRMs exhibit distinct computational and memory consumption characteristics compared to convolutional and recurrent neural networks [26, 30]. The embedding operations dominate the run-time of recommendation models. These operations are characterized by large embedding tables (typically on the order of tens to hundreds of gigabytes), low compute-intensity and little to no regularity, making them ill-suited to run on GPUs. In contrast, the model's dense network component is better executed on GPUs, which can achieve 10–44× speedup compared to running on CPUs, as shown in our experiments. Therefore, current DLRM serving systems decouple the sparse embedding lookup operations from dense computations. The former runs on a large number of CPUs, and the latter runs on a GPU.

2.2 Challenges in DLRM Provisioning

DLRM services serve a massive volume of requests with stringent latency SLOs and account for the majority of inference cycles in today's AI cloud [26]. These services feature dynamically changing requests with large *daily* and *seasonal* variations as shown in Figure 1. Instead of serving DLRMs in a dedicated cluster, which requires provisioning for the peak load and is economically unviable, at Alibaba we serve DLRMs along with other machine learning models in *shared clusters* in normal days; we additionally *loan* servers from training clusters to handle large load spikes during seasonal promotion events. However, this approach faces two challenges in practice.

C1: Resource fragmentation for daily DLRM serving. In shared clusters, frequently scaling DLRM provisioning based on the changing load results in severe resource fragmentation [64]. Figure 3 illustrates the distribution of available CPUs and GPUs of servers in a large GPU cluster, where both CPU and GPU allocation rates reach 90%. Despite having sufficient CPU and GPU quotas, DLRM service owners observe prolonged scheduling time or even scheduling failures [64]. From their perspective, the cluster has over 30k fragmented CPUs and more than 200 fragmented GPUs (detailed explanation in §5.4). The severe fragmentation is primarily due to the high CPU-to-GPU ratio of these inference instances (Figure 4, right), which cannot be accommodated using the servers' remaining resources. In practice, scheduling new DLRM instances often requires manually relocating running jobs to reduce fragmentation, which is time-consuming.

C2: Ineffective capacity loaning during seasonal load spikes. Throughout a year, Alibaba hosts several large-scale seasonal shopping festivals [68], during which DLRM services experience significant load spikes, exceeding the daily peak by more than 1.3× (Figure 1). To handle these transient, excessive load spikes, *capacity loaning* is often needed, which we elaborate as follows.

Production AI clouds are built from a large quantity of *heterogeneous servers* with configurations meant to support either training or inference tasks. Figure 4 (left) shows the distribution of the CPU-to-GPU ratio of the server nodes used for training and inference in a production cluster at Alibaba. In general, nodes suitable for training tasks are configured to have multiple advanced GPUs (e.g., 4 or 8 H800 GPUs) with high-speed interconnect (e.g., NVLink) but only a limited number of CPUs (e.g., 16 to 32 cores), thus having a low CPU-to-GPU ratio. In comparison, nodes optimized for inference tasks, especially DLRM serving, have a much higher CPU-to-GPU ratio, typically with a single GPU but a large number of CPU cores (e.g., 128 cores) to accelerate CPU-intensive operations, such as data preprocessing and embedding lookup. The presence of these heterogeneous servers essentially segregates a datacenter into two purpose-specific infrastructures, one for training and the other for inference. To improve utilization and reduce overprovisioning, capacity loaning between the two infrastructures should ideally be enabled. That is, when DLRM serving is in peak hours, it can temporarily loan training servers to handle excessive recommendation queries.

However, the mismatch between server configuration and resource demand renders capacity loaning ineffective. Figure 4 (right) shows the distribution of CPU-to-GPU ratio of the recommendation models deployed in our production clusters. Over 90% of DLRMs have CPU-to-GPU ratio greater

than 20, whereas *all* multi-GPU training nodes have a ratio below 20. As a result, running DLRMs on these training nodes quickly exhausts CPUs, leaving a large number of GPUs stranded, unable to utilize by other tasks.

2.3 GPU-Disaggregated DLRM Serving

Resource disaggregation [18, 23, 25, 53] holds tremendous promise in DLRM serving. This architecture allows decoupled independent scaling-out of CPUs and GPUs, thereby addressing the mismatch between server configuration and resource demands. Resource sharing is also made possible as GPU-intensive AI workloads, such as training, can run on the same infrastructure, together with recommendation models, significantly increasing the system utilization. Our design of a GPU-disaggregated DLRM inference service is motivated by two fundamental observations.

O1: Distinct characteristics of DLRMs. DLRMs exhibit a distinctive computational structure that naturally lends itself to graph partitioning based on operator-level resource affinity. Our analysis reveals that embedding table queries dominate CPU computation, while matrix multiplication accounts for the majority of GPU computational overhead (Figure 8). While DLRM services routinely update model parameters and embedding table values to reflect real-time user interaction, the underlying computational graph remains static. This architectural stability ensures that graph partitioning and disaggregation optimizations, once computed, remain optimal throughout the model's deployment lifecycle, eliminating the need for dynamic repartitioning.

O2: High-performance RDMA network. Modern RDMA network interface cards (RNICs) deliver bandwidth capabilities of 200–400 Gbps, matching or even surpassing that of PCIe 4.0 x16 interconnects (256 Gbps). RDMA's offload capabilities bypass the kernel networking stack, minimizing CPU overhead——a critical factor for inference workloads in disaggregated environments. Optimized RDMA networking can provide scalability, stability, and SLO guarantees for disaggregated inference scenarios (§5).

GPU disaggregation approaches. GPU disaggregation can be achieved using three approaches, as summarized in Figure 5. We compare these approaches and justify our choice.

1) API-level disaggregation: this approach intercepts program calls to CUDA APIs and redirects them to a remote GPU node for execution [2, 15, 16]. It requires only replacing the original CUDA library inside the container without any modification to the serving framework. However, its application agnostic nature offers little optimization opportunities. Future CUDA upgrades would also require considerable engineering effort in forward compatibility and performance tuning. Moreover, supporting heterogeneous AI accelerators (e.g., AMD GPUs) necessitates developing and maintaining distinct remoting layers for each platform.

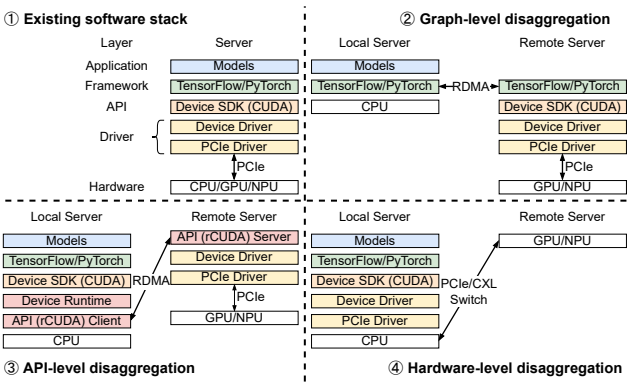


Figure 5: GPU disaggregation at different levels.

2) Hardware-level disaggregation: GPU disaggregation can be ideally enabled with specialized hardware, such as customized multi-hop PCIe switches [29, 42] and CXL technologies [1]. Hardware-level disaggregation requires no modification to software and incurs minimum I/O latency. However, it faces significant deployment barriers as it requires expensive infrastructure upgrade and customization. The disaggregation is also confined to a short distance (e.g., PCIe switches support disaggregation only within a rack).

3) Graph-level disaggregation: A DLRM is represented as a compute graph consisting of multiple operators running on CPU or GPU. Inspired by model parallelism, we can partition the compute graph into a CPU sub-graph and a GPU sub-graph (**O1**). The two sub-graphs can then be scheduled on selected CPU and GPU nodes for disaggregated execution, with communication over high-speed RDMA network (**O2**). Compared with the previous two approaches, graph-level disaggregation requires no specialized hardware and can achieve good performance with optimized partitioning, scheduling, and network transport. We hence consider it as a viable approach and base our system design on it.

3 Prism Overview

In this section, we present Prism, a large-scale DLRM system that enables GPU-disaggregated serving by means of graph partitioning. Prism has been deployed in our production clusters, serving as the underlying infrastructure for core recommendation services. As of January 2025, the system serves over 20k DLRM instances, utilizing more than 10k GPUs and 800k CPUs. As illustrated in Figure 6, Prism operates on a cluster where a fleet of heterogeneous GPU nodes (HNs) interconnects with a number of CPU nodes (CNs) via a high-speed RDMA network. A CN is configured with a large number of CPUs and high memory, while an HN is a multi-GPU node. This results in two disaggregated resource pools, a CPU pool provisioned by CNs and a GPU pool provisioned by HNs. Prism automatically partitions recommendation models for distributed inference on CNs and HNs.

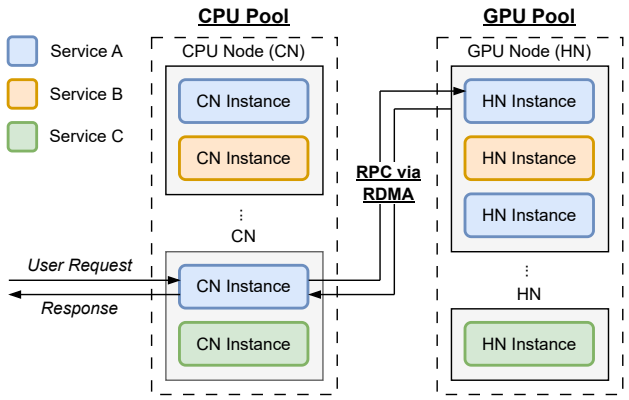


Figure 6: The cluster deployment of DLRM inference instances and request execution flow in Prism.

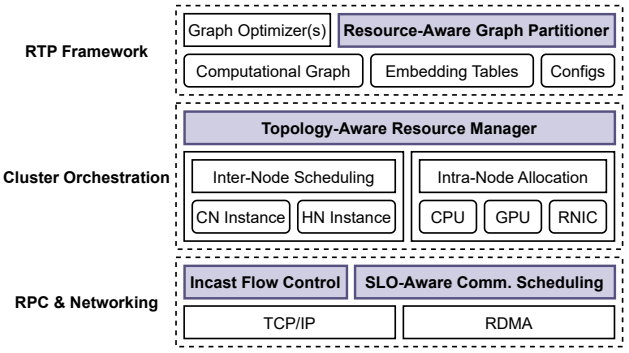
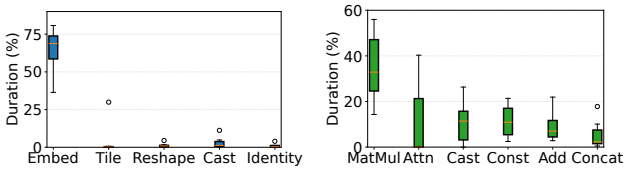


Figure 7: Prism overview and its key designs are in *purple*.

System architecture. Figure 7 provides an overview of Prism. The RTP framework (§4.1) takes input the compute graph, associated embedding tables, and configuration parameters (e.g., enabling/disabling custom optimizers). It divides a monolithic recommendation model into two subgraphs, one consisting of CPU-intensive operations and the other consisting of GPU-efficient operations. The resource manager (§4.2) then packages the two subgraphs into containerized CN and HN instances and places them onto the selected CN and HN nodes for disaggregated execution, by considering the network and node topology. The manager makes resource allocation decisions (e.g., CPUs, GPUs, RNICs) and dynamically scales CN and HN instances in response to the changing load. Once the CN and HN instances are placed on nodes, the communication scheduler establishes RDMA connections and performs incast control and SLO-aware request scheduling to meet latency SLOs (§4.3).

Execution flow. As illustrated in Figure 6, user requests are first routed to a frontend CN instances for embedding lookups and other CPU-intensive computations. Intermediate tensors are then transferred to the remote HN instance on a GPU node through RPCs over the RDMA network. After the HN instance returns results, the CN instance performs post-processing before sending the response to the user.



(a) Major CPU operators. (b) Major GPU operators.

Figure 8: Operator analysis of online DLRM services.

4 Prism Design

Prism is designed to meet three key requirements:

- *Transparency to model development and optimization:* For production deployment, it is important to make disaggregation transparent to model developers, without their cooperation to modify model architecture or implementation. It is hence essential to enable automated graph partitioning to support disaggregated inference for various recommendation models, without affecting users or invalidating original graph optimization strategies (§4.1).
- *Compliance to SLOs:* Disaggregation inevitably introduces performance overheads, necessitating a joint optimization approach across various system components to minimize the impact on service performance. For example, the compute graph must be judiciously partitioned and optimized to effectively separate the resource usage of different operators (§4.1). Furthermore, the node placement and resource allocation of instances require careful consideration to ensure good performance (§4.2).
- *Scalability*. In contrast to traditional recommendation services, GPU disaggregation introduces non-trivial communication overhead. Our system must ensure service performance remains unaffected, even under conditions of multiple server instances and high traffic loads in production environments (§4.3).

4.1 Resource-Aware Graph Partitioner

The most critical *first* step in decoupling CPU and GPU computation is to partition a model's original computation graph into two subgraphs, which are executed respectively by the CN and HN instances.

Retrofit for the existing workflow. In production DLRM systems, the recommendation framework provides model developers with a plethora of optional optimization techniques (e.g., JIT fusion, CUDA Graphs [22]) to enable or disable based on the model characteristics. These optimizers are typically applied in a *sequential manner*, iteratively rewriting the original computation graph and ultimately generating an optimized computation graph tailored for deployment. The

primary objective of our disaggregation approach is to minimize the impact on the service's inference performance. Thus, Prism introduces *graph partitioning* and *disaggregation optimization* as the final stages in the existing DLRM workflow, strategically waiting for all user-specified optimization strategies to be completed before meticulously reconstructing the computation graph for disaggregated serving.

Operator analysis of DLRMs. To gain deeper insights into the computational characteristics of DLRM services, we conducted a comprehensive profiling of the operators, as illustrated in Figure 8. The results unveil two key observations that serve as the foundation for our *resource-aware graph partitioning* algorithm: (1) Embedding lookup operators exhibit a dominant presence, accounting for over 70% of the CPU computation (Figure 8, left); consequently, we designate all embedding lookup-related operators as `CPU-intensive operators`. (2) Matrix multiplication computations emerge as the most significant contributor to GPU computation time. In more advanced models [11] incorporating transformers [57], the computation ratio of attention layers can even reach a staggering 40% (Figure 8, right); we categorize these operators as `GPU-efficient operators`.

Device placement. Leveraging the aforementioned observations, the RTP framework employs a heuristic approach to partition the GPU subgraph that necessitates computation on the HN instance: (1) `GPU-efficient operators` are selected as the initial seeds for the partitioning process. (2) Commencing from the seed operators, a Depth-First Search (DFS) coloring algorithm is simultaneously executed from both upstream and downstream directions, aiming to encompass the maximum number of operators feasible for GPU computation. (3) The DFS coloring process is terminated upon encountering `CPU-intensive operators`, ensuring an optimal balance between CPU and GPU utilization.

Optimization for disaggregation. To enhance the efficiency of distributed graph execution, we introduce two complementary strategies that achieve up to 50% reduction in inter-node data transfer volume: (1) We preserve and optimize constant subgraph execution. While constant operations constitute 10–20% of GPU computational workload (Figure 8, right), their optimization potential is often compromised when fragmented across different servers. By consolidating constant operations onto the HN instance, we maintain the structural integrity of constant subgraphs. This ensures that constant computations are executed exactly once and their results are cached for subsequent operations, eliminating redundant data transfers. (2) In cases where multiple derivative tensors are generated from a common source tensor and subsequently transmitted from the CN instance to the HN instance, we employ a size-aware transfer strategy. When derivative tensors' aggregate size exceeds the ancestor's magnitude, we optimize communication patterns by transmitting only the source tensor, thereby reducing the network bandwidth requirements.

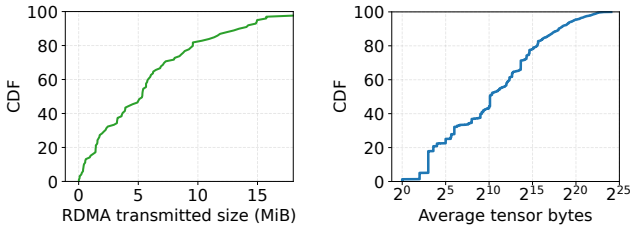


Figure 9: Statistical analysis of data transfer for top 100 disaggregated DLRM inference services. **Left**: Total volume of RDMA data transfers from a CN instance to a HN instance; **Right**: Distribution of transferred tensor sizes.

RPC for remote GPU execution. We develop a unified operator called `FusedGraphOp` for RPCs. This operator aggregates all tensors that require transmission from the CN instance and sends the requests to one of the remote HN instances. Upon completion of the remote GPU computation, the results are returned, marking the conclusion of the entire inference process. To minimize memory copies between the computational framework (e.g., TensorFlow) and the RPC system, we establish a unified memory pool where `FusedGraphOp` flags tensors requiring transmission and relinquishes control to the RPC system. By leveraging GPUDirect RDMA [7], these data are already stored in the RPC system's memory buffer upon creation, realizing defacto *zero-copy data transmission* [27, 67]. Experimental results demonstrate that this approach can yield performance improvements of 19–181%. The RPC system adaptively balances load using real-time processing latency of each HN instance [13].

Production deployment. We deploy Prism to evaluate DLRMs in production. In 80% of recommendation services, the data transfer size via the RDMA network is less than 10 MiB (Figure 9, left). Note that we focus on unidirectional data transfer from the CN instance to the HN instance, as the data sent back is typically *negligible* (e.g., approximately 100 KiB). The results presented in §5 demonstrate that, at this scale of transfer size, disaggregated inference can effectively separate resource requirements without compromising service performance. Interestingly, we observe a *bimodal* distribution in the size of transferred tensors. Only 11% of the tensors exceed 0.25 MiB, while a significant 44% of the tensors are smaller than 1 KiB (Figure 9, right). These findings motivate us to adopt a size-based approach when transferring tensors. For small tensors (e.g., $\leq$ 4 KiB), we directly employ RDMA send/recv operations, whereas for large tensors, we utilize RDMA write operations.

Implementation. Our implementation is built upon an optimized internal version of TensorFlow v1.12 [9], comprising approximately 2k lines of Python code for graph partitioning and disaggregation optimization, along with 3,500 lines of C++ code for the `FusedGraphOp` functionality. Prism rewrites

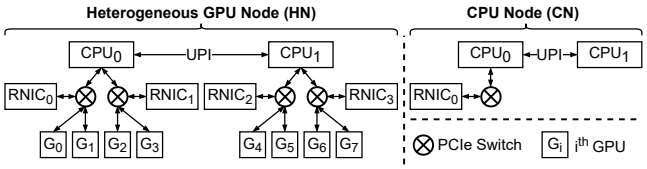


Figure 10: The hardware topology of a typical HN and CN.

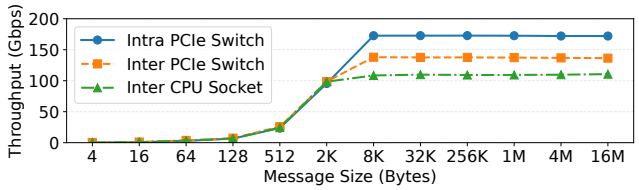


Figure 11: Performance comparison of GPUDirect RDMA across different GPU-RNIC interconnect topologies.

Algorithm 1: Topology-aware resource scheduling

Input: Cluster N, DLRM application id app, CN/HN instance $role$, resource requirements (cpu_num, gpu_num, mem_size, $disk_size$), deployment density d
Output: Selected node $node_id$, allocated resources (cpu_id, gpu_id, $rnic_id$)

1 Initialize node set $S \leftarrow \varnothing$
2 $N_{pod} \leftarrow N$.filter($N.cluster_topology, app$) ▷ Retain all nodes in the pod that runs the same app's existing instances
3 $N_{asw} \leftarrow$ get_asw_nodes(N_{pod}, app) ▷ Get nodes from ASWs with maximum instances
4 **parallel for** *node* $n \in N_{asw}$ **do**
5 **if** (*Insufficient resources* $\|$ $get_density(n, app) \geq d$) **then**
6 Return ▷ Filter out unavailable nodes
7 `// Prefer the shortest CPU/GPU-to-RNIC path`
8 **if** $role$ = *HN* **then**
9 $(cpu_id, gpu_id, rnic_id, topo_score) \leftarrow$ calc($n.topology$, GPU, RNIC, gpu_num)
10 **else if** $role$ = *CN* **then**
11 $(cpu_id, \varnothing, rnic_id, topo_score) \leftarrow$ calc($n.topology$, CPU, RNIC, cpu_num)
12 $S \leftarrow S \cup (n, cpu_id, gpu_id, rnic_id, topo_score)$
13 $node_id$, cpu_id, gpu_id, $rnic_id \leftarrow \arg\max_S topo_score$

the user-submitted model graph, where the CN subgraph receives the original model inputs (e.g., user attributes, candidate item lists), and the HN subgraph receives tensors sent via `FusedGraphOp`. If a tensor requires transfer via the RPC library and subsequent operator consumption, Prism replicates a new operator and delegates control to the RPC library. We also provide support for the PyTorch framework [49]. The subsequent experiments (§5) focus on TensorFlow versions to evaluate overall system performance.

4.2 Topology-Aware Resource Manager

The RTP framework disaggregates the model onto CN and HN instances, with the resource manager selecting nodes and allocating resources. To ensure recommendation service SLOs, the resource manager adheres to *topology-aware* node scheduling and resource allocation principles (Algorithm 1).

Inter-node scheduling. The cluster consists of a fleet of CNs and HNs (Figure 6). These nodes are grouped in a cluster unit called a pod (point-of-delivery), where nodes have one or more dual-port high-performance RNICs. Intra-cluster communication is achieved through a classic two-tier clos network [50]. Prism follows two principles for scheduling all CN and HN instances of the same DLRM service. First, all instances are confined within the same pod (Algorithm 1, Line 2), as *cross-pod RDMA connections induce over 50% performance degradation* in our evaluation. Second, co-locating CN and HN instances within the same ASW (Access Switch) offers superior performance. Therefore, we prefer this affinity and endeavor to pack these instances under the same ASW. If instances are distributed across multiple ASWs, Prism schedules new instances to the ASW with the most existing instances (Algorithm 1, Line 3).

Intra-node resource allocation. The hardware topologies of a typical HN and CN are depicted in Figure 10. The interconnect between GPUs and RNICs has a substantial impact on performance. Figure 11 shows that arbitrary bindings of GPUs and RNICs can induce 21–36% performance loss. Therefore, the resource scheduler prefers to assign RNIC and GPU on the same PCIe switch for each HN instance, mitigating data movement. Regarding CPU and RNIC topology, Prism mainly considers CPU-intensive CN instances. CN typically has only one RNIC, so the scheduler prioritizes the allocation of CPUs under the same PCIe switch connected to the RNIC (Algorithm 1, Line 8–11).

Production deployment. To ensure fault tolerance, online inference services deploy instances across multiple data centers. Simultaneously, to mitigate the effects of traffic peaks, the resource scheduler limits the deployment density of DLRM instances (Algorithm 1, Line 5). This scheduling policy avoids bursty resource consumption on the same node. During peak seasonal traffic periods, the scheduler incorporates additional instance density constraints at the NIC switch layer. These constraints are derived from the aggregate bandwidth capacity of distributed switches and predetermined safety thresholds (e.g., 80%). The scheduler dynamically allocates bandwidth quotas for each instance based on these constraints, thereby preventing network saturation.

Implementation. We develop topology-aware scheduling capabilities as a scoring plugin integrated within Kubernetes [4]. The device plugin deployed on each node and the central scheduler each implemented approximately 1,000 lines of code in Golang. The device plugins on individual nodes manage heterogeneous hardware topologies, reporting available GPUs and virtualized RNICs to the scheduler.

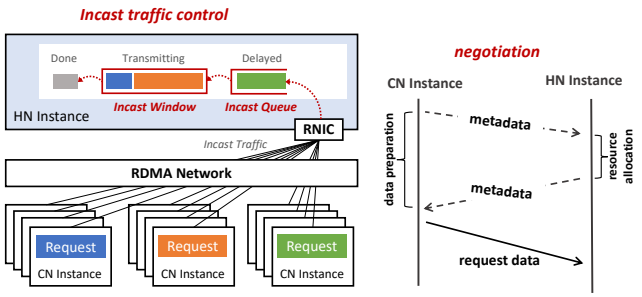


Figure 12: The incast control mechanism comprises of an incast window and queue. Only requests within the window can transmit parameters. The queue stores delayed requests.

4.3 SLO-Aware Communication Scheduler

Online recommendations demand stringent performance and stability guarantees. Empirical evaluations indicate that to adhere to stringent SLOs, the p50 and p99 latencies for transmitting 4 MiB model parameters must be below 3 ms and 6 ms, respectively. Notably, when a substantial number of CN instances concurrently transmit data to a limited number of HN instances, it generates a fan-in traffic pattern, also known as *incast*, as depicted in Figure 12. High-volume incast traffic can degrade overall performance, including long-tail latency and reduced throughput.

Incast flow control. As presented in Table 1, under various incast configurations, failing to impose restrictions on incast flows can result in request failures, which is unacceptable for a reliable online service. To meet the strict SLOs and mitigate the performance degradation caused by incast traffic, we propose an SLO-aware incast control mechanism. Incast control enhances communication performance by reducing the volume of incast traffic. As illustrated in Figure 12, incast control orchestrates the HN and CN instances to throttle incast traffic in a window-based manner. Initially, the CN instance sends a communication request before transmitting model data. Next, the HN instance verifies whether the incast window has the sufficient capacity to receive the model parameters. If space is available, the HN instance sends a communication response to initiate the data transmission. Otherwise, the HN instance defers processing the communication request in the incast queue until the incast window has the adequate capacity.

The *incast window size* and the *processing order of delayed requests* are crucial factors in determining the effectiveness of incast control in meeting SLOs, necessitating meticulous design. An undersized window leads to unnecessary delayed requests, while an oversized window renders incast control ineffective. Moreover, communication requests arrive with varying SLOs, and requests with more stringent SLOs might be impeded by other requests, resulting in SLO violations. We address these challenges through *adaptive incast window* and *SLO-aware communication request scheduling*.

Incast Size	Latency	% of Failed Requests
10	10 ms	≈ 33%
20	40 ms	≈ 50%
100	10 s	≈ 100%

Table 1: Incast statistics from our clusters. Incast size is the number of concurrent messages from different CN instances. Latency refers to the time for a HN instance to process 1,000 requests. The default message size is 8 MiB, typical for our services. Failed requests include timeouts or packet loss.

Algorithm 2: SLO-Aware Scheduling

Input: Newly-arrived communication request r' with its completion requirement $SLO_{r'}$ and message size $M_{r'}$, available network bandwidth B, requests in the incast queue R, current time t_{now}
Output: Scheduled communication requests R'

1 $d_{r'} \leftarrow t_{now} + SLO_{r'} - M_{r'}/B$ `/* calculate deadline */`
2 **foreach** *r in R* **do**
3 **if** $d_{r'} > d_r$ *and* r' *not in* R' **then**
4 R'.append(r')
5 R'.append(r)

Adaptive incast window. In Prism, the incast window size adapts to the congestion level of network links and PCIe links, as demonstrated in Figure 10. For PCIe link congestion, the traffic is stored in the RNIC receiver buffer. For network link congestion, the switch buffer stores the traffic. When the buffer occupancy of RNIC and switch surpasses a predefined threshold, the RNIC notifies the senders of congestion via Congestion Notification Packets (CNPs). Consequently, the number of CNPs serves as an estimator of the congestion level. The HN instance periodically collects the count of sent CNPs and calculate the congestion level by averaging CNP count by the number of f congestion requests. If the congestion level exceeds a predefined threshold T_{high}, the window sizes is increased. Conversely, if the congestion level falls below T_{low}, the window size is decreased. In Prism, the values of T_{high} and T_{low} are tunable based on the online workloads, preventing frequent fluctuations in the window size.

Deadline-aware request scheduling. Intuitively, Prism can process delayed requests held in the incast queue using a first-come-first-serve (FCFS) approach. Although straightforward to implement, FCFS is not SLO-friendly, as early-arriving requests with loose SLOs can obstruct requests that arrive later but have more stringent SLOs. To overcome this issue, the incast queue is designed to pop requests in a SLO-aware manner. Specifically, upon the arrival of a new communication request, Prism reorders the requests in the incast queue to maximize the number of requests meeting their SLOs. Based on this insight, we propose the early-deadline-first scheduling for delayed communication requests. The deadline of a communication request is defined as the latest time to initiate parameter transmission to meet the SLO. Prism calculates the deadline for the arrived request (Algorithm 2, Line 1) by

subtracting parameter transmission overhead ($M_{r'}/B$) from the sum of the current time and SLO. Then, r' is inserted after the requests with smaller deadline (Line 2–5).

Operational experience. Production clusters frequently employ a mixed deployment of online and offline tasks to improve efficiency. In our scenarios, the disaggregated DLRM serving introduces frequent RDMA network communication. We observe that even with exclusive access to the RNIC, RDMA transfer latency can experience a tenfold increase under intense resource contention. The root cause is that under the container overlay network [72], both RDMA and TCP rely on the overlay network scheme for communication. The TCP traffic from mixed workloads affects the flow table logic at the bottom layer of the network card, impacting RDMA traffic. Our current workaround involves monitoring node resource utilization and online service latency, triggering the eviction of offline tasks when the metrics become abnormal. We consider this as an open question for future research.

Implementation. We extend the native RoCEv2 [24] stack and implement an efficient middleware to fully leverage RDMA communication capabilities in a virtualized environment. The implementation consists of 40k lines of C++ code. Standard protocols [71] running in RNICs act as a black box for upper-level system components, which is not suitable for rapid testing and deployment. Thus, we implement an additional software-level control strategy to mitigate the negative impact of incast congestion. We design specialized meta-message packets for negotiation between senders and receivers to restrict the number of busy connections (Figure 12). This complementary software control mechanism is tuned with underlying congestion control protocols to ensure stable performance. To reduce overhead from registering memory regions for RDMA requests, Prism maintains a pre-allocated memory pool and several maps from buffer addresses to the related memory region objects. A memory allocator similar to slab is adopted to prevent buffer fragmentation.

5 Evaluation

In this section, we conduct extensive experiments to evaluate the performance impact and resource efficiency improvements of Prism. Our results demonstrate the following:

- Prism's disaggregated inference can maintain service performance under *high*-traffic scenarios (§5.2).
- Prism can effectively separate CPU and GPU computations, enhancing GPU efficiency in multi-GPU nodes by 5–9× (§5.3).
- Prism has been deployed in production clusters for over two years. In a daily GPU cluster with a high allocation rate (90%), it can reduce CPU fragmentation by 53% and GPU fragmentation by 27%. Additionally, during seasonal peak traffic, Prism can efficiently borrow training nodes to meet the increased demand, saving over 90% of GPUs (§5.4).

Model	Emb Size (Approximate)	RDMA TX (Per Req)	Dense Features
Model-XS	100 GiB	552.96 KiB	338.67 MiB
Model-S	450 GiB	6.84 MiB	57.20 MiB
Model-M	500 GiB	3.87 MiB	21.46 MiB
Model-L	600 GiB	3.69 MiB	20.79 MiB
Model-XL	700 GiB	9.03 MiB	8.73 GiB

Table 2: Models for evaluation.

5.1 Methodology

Production workloads. We use models from our industrial DLRM system, with test data originating from *real-world* user requests. Table 2 outlines the *high-level* characteristics of these models, including the size of their embedding tables, the communication volume per request transferred via RDMA after disaggregation, and memory space required for storing dense features. The models analyzed span a range of scales and use cases, processing *billions* of daily requests for tasks such as video click-through rate prediction, ad conversion rate prediction, and item ranking.

Machine specifications. The hardware topology of the HN and CN is illustrated in Figure 10. By default, model instances are deployed on HNs. Each HN contains 128 CPU cores, 8 A100 [5] GPUs with 80 GiB GPU memory each, and 4 RNICs with a 200 Gbps. After resource disaggregation using Prism, CN instances are deployed on CNs. Each CN is configured with 128 CPU cores and a single 200 Gbps RNIC. All nodes use Intel(R) Xeon(R) Platinum 8369B CPUs, with 1024 GiB memory, interconnected in a pod.

Baselines. To the best of our knowledge, Prism is the first GPU-disaggregated inference system for DLRMs. Our experiments feature three primary baselines for comparison: (i) *Baseline* refers to our highly optimized production DLRM inference system without resource disaggregation, as a comparison for performance overhead; (ii) *Local Disaggregation* splits the original DLRM instance, but the CN instance and HN instance co-locates on the same HN server; (iii) *Prism (Remote Disaggregation)* disaggregates the DLRM inference service by placing CN instance on the CN server and the HN instance on the HN server.

Metrics. We seek to evaluate the DLRM system's processing capabilities under peak traffic loads. To simulate such conditions, we initialize 30 workers to continuously send requests at a predefined frequency. Our key indicators are categorized into two groups:

Performance-level metrics: (i) *goodput*, defined as the number of requests that DLRM instances can process; (ii) *average latency*; and (iii) *p99 latency*.

Resource efficiency metrics: (iv) *CPU usage*: the overall CPU consumption of the DLRM service and the proportion of CPU usage on GPU nodes; (v) *GPU efficiency*: measured by the number of inference requests that can be processed on

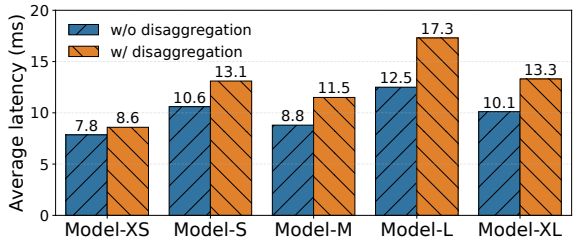


Figure 13: Latency of sequentially executing requests.

a single GPU node; and (vi) *resource fragments*: the amount of unallocated resources that are *expected* to be fragmented because they are too small to meet workload requirements or become stranded from other dimensional resources, quantified using the methodology proposed in [64].

5.2 Prism Performance: End-to-End Results

Overhead of serving a single request. First, we quantify the impact of model disaggregation on inference efficiency in the *worst-case* scenario. We send only *one* request at a time to the DLRM inference system for sequential execution, which indicates *low* resource utilization and throughput. Figure 13 records the average latency of sequentially processing requests in different models, showing a 10–38% increase. In this setup, the performance degradation of inference mainly stems from two aspects: (1) each request processing requires data transmission across the RDMA network, and (2) when executing requests sequentially, only one instance of Prism's CN instance or HN instance is computing at a time, whereas the baseline allows *asynchronous* executions of CPU and GPU operators on the same node.

Performance under varying traffic loads. Figure 14 shows the performance of various baselines under differing levels of requests per second (RPS). While Prism incurs minor overhead from cross-node communication during low RPS inference (e.g., p99 latency of Model-L at 200 RPS), it can *lower latency and increase goodput under higher RPS* compared to other baselines (e.g., p99 latency of Model-L at 800 RPS). Two main factors drive these improvements. First, modern GPUs' first-in-first-out scheduling can result in substantial queue head-of-line blocking [48], where at high RPS, concurrent requests *interleavingly* submit kernels to the GPU will drastically increase the latency. Prism mitigates this by leveraging `FusedGraphOp` to fuse each request's GPU operators, into a single sequential execution rather than interleaving across requests. Second, Prism decreases PCIe volume by converting host-to-device data transfers in DLRM computation to high-bandwidth device-to-device RDMA transfers. *Local disaggregation* also shows performance gains due to the graph optimizations mentioned above, but CPU contention between the CN instance and HN instance results in inferior performance compared to Prism. One special case is Model-XL, which has the largest communication volume (i.e., over 9 MiB) but very short GPU time ($<$ 1 ms, the lowest part of Figure 15). Even under this condition, Prism only suffers at most 6% performance loss. Overall, Prism is capable of maintaining service performance under *high* traffic scenarios in real-world deployments.

Latency breakdown. We perform additional breakdown analysis of elapsed time across stages, divided into CN computation time, data transmission latency via the RDMA network, and HN computation time. We discover that different DLRM inference services have distinct bottlenecks, which can be classified as *CPU-bounded*, *balanced*, or *GPU-bounded*. For example, the increased latency of Model-XS at high RPS is primarily attributed to the HN instance, indicating that the computational capacity (i.e., GPU) of the HN instance becomes the bottleneck. In contrast to the baseline, Prism decouples the resource usage of DLRM inference, allowing for *arbitrary* configuration of the number of CN and HN instances. This *flexibility* enables Prism to better accommodate the diverse resource requirements of different DLRM services (e.g., independently scale out HN instances to alleviate the computational bottleneck, in the case of GPU-bounded services like Model-XS, as shown in Figure 17).

5.3 Prism at Node: Resource Efficiency

Distinct resource separation. Figure 16 compares the CPU usage across different DLRM services before and after resource disaggregation. Prism maintains nearly identical CPU consumption when processing equivalent amounts of goodput, in comparison to the baseline. Notably, Prism effectively separates the computational resources of DLRM, resulting in a substantial reduction of CPU consumption on GPU nodes by 15–84×. These findings suggest that Prism possesses the capability to effectively mitigate GPU fragmentation issues that arise due to insufficient CPU resources.

Better resource efficiency on multi-GPU nodes. We deploy varying numbers of inference instances on a single HN to handle DLRM requests, limiting the service latency to 25 ms, and then compare the total goodput. Figure 17 demonstrates that Prism can deploy more DLRM instances on a single multi-GPU node, increasing inference throughput by 5–9×. Without disaggregation, the baseline can deploy a maximum of two Model-XS inference instances on the HN, fully saturating the CPUs while leaving 6 GPUs completely idle and *stranded*. For Model-XL, due to the memory limitations imposed by its massive embedding tables (i.e., 700 GiB), the baseline can deploy at most one instance. Prism supports flexible configuration of the ratio between CN and HN instances. For CPU-bounded Model-XL, only increasing the number of CN instances from 1 to 4 can yield a 36% improvement. For GPU-bounded Model-XS, if the MIG [8] feature

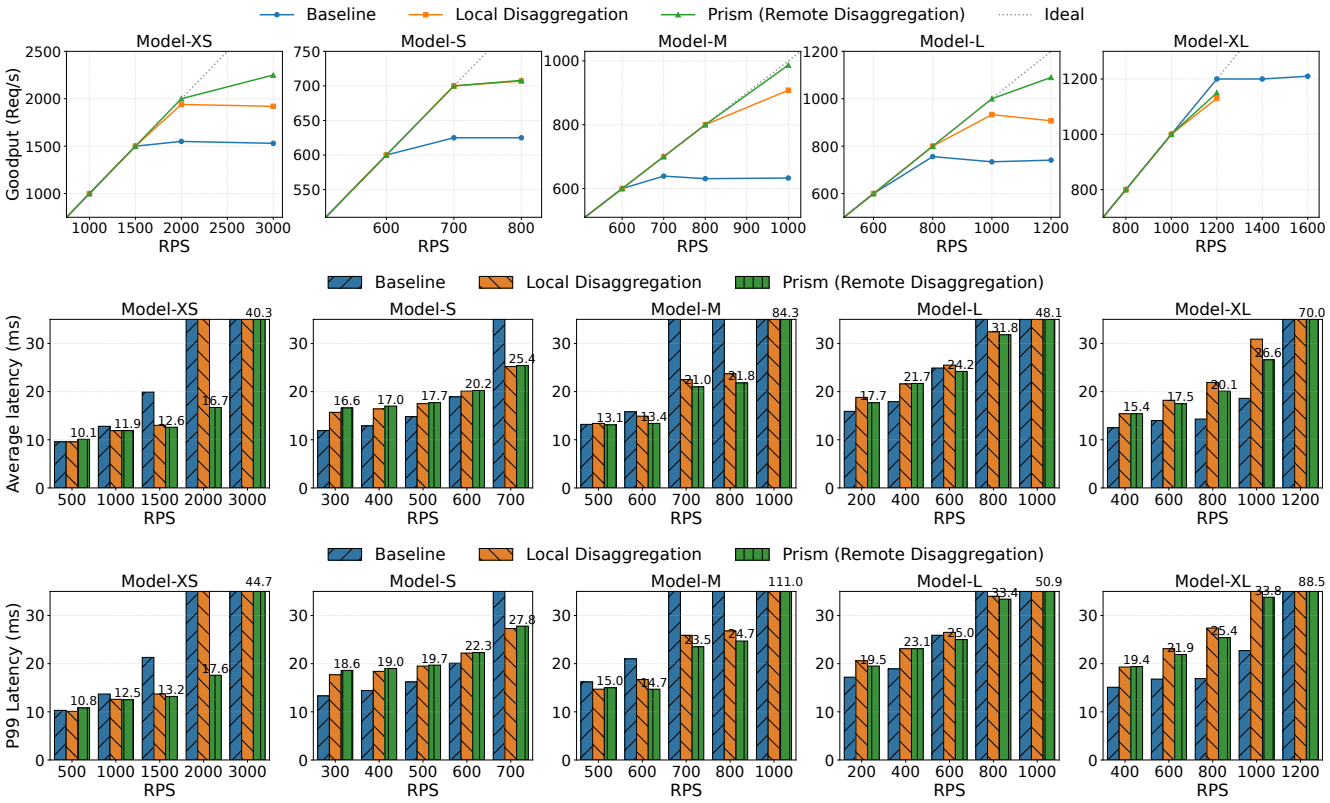


Figure 14: The end-to-end performance under varying traffic loads.

of A100 (i.e., fine-grained GPU allocation) is enabled, up to 24 HN instances can be deployed, boosting throughput to 9× compared to the baseline. This demonstrates that Prism's resource disaggregation allows for more efficient utilization of multi-GPU nodes customized for training workloads.

5.4 Prism at Scale: Production Deployment

Prism has been deployed in our production cluster for over two years, scaling to more than 10k GPUs to date. The system addresses two challenges: (1) resource allocation in GPU clusters with high allocation rates and (2) resource provisioning during seasonal traffic spikes. By leveraging a design that avoids costly infrastructure upgrades, Prism has enabled serving scale to grow continuously throughout this period.

Reduced resource fragmentation in GPU clusters. As discussed in [64], cluster resource fragmentation must consider the distribution of workloads. For tasks with different resource requirements, the cluster resource fragmentation from their perspective varies. Figure 18 compares the changes in resource requirements of DLRM instances before and after our deployment of Prism. Prism reduces the stringent resource requirements of the DLRM inference service. For HN instances that require GPU allocation, their CPU requests are less than 12 cores, and memory requests are below 24 GiB. In contrast, CN instances, which are CPU-intensive and need to load embedding tables, have CPU requests exceeding 48 cores and memory requests greater than 240 GiB. We simulate the deployment of DLRM instances before and after resource disaggregation to cluster H with a high allocation ratio. Figure 19 compares the changes in fragmented cluster resources. As Prism separates the resource requirements of DLRM instances, CN instances can run on nodes where GPUs are exhausted but CPUs remain, while HN instances can run on nodes with insufficient CPUs but available GPUs. Statistical analysis reveals that this approach significantly reduces the cluster's CPU fragments by 53% (18k cores) and GPU fragments by 27% (60 GPUs). These findings suggest that even in clusters with high allocation rates, the disaggregated DLRM service can efficiently utilize fragmented resources to meet deployment requirements.

Efficient resource loans for peak demand. During e-commerce promotional periods, we leverage Prism to borrow a portion of training nodes (equipped with A100 GPUs) to scale out DLRM inference services, meeting the *short-term* yet *high-throughput* traffic peaks. Table 3 outlines the resource requirements of three online services, with each HN instance requiring only one MIG GPU and minimal CPU resources (≤ 4), while the remaining CN instances handle the vast majority of CPU computations. These instances are

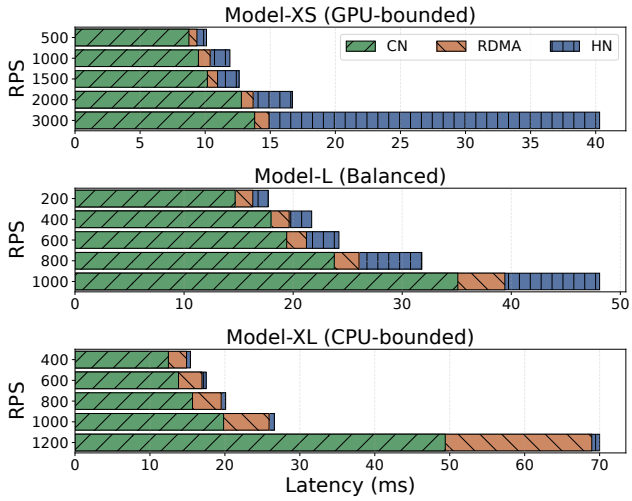


Figure 15: Latency breakdown of disaggregated DLRM inference services, which consists of CN computation time, RDMA transfer latency, and HN computation time. Model-XS, Model-S, and Model-M demonstrate consistent trends.

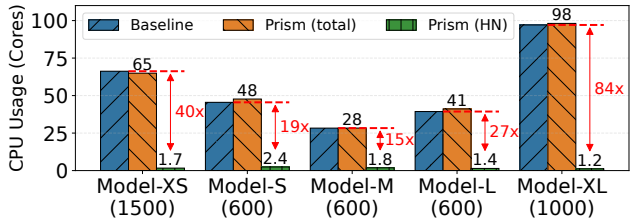


Figure 16: Comparison of CPU utilization. Prism (total) represents the total CPU consumption of the CN instance and HN instance in the disaggregated DLRM inference service, while Prism (HN) denotes the CPU consumed by the HN instance. The x-axis labels include the goodput in parentheses.

distributed across different nodes to ensure high availability. Figure 20 illustrates the QPS and latency of the three services during seasonal promotions, with the peak occurring at the 20th hour. Latency for the online service was not detrimentally impacted—overall latency remained within 25 ms and no request exceeded 35 ms, demonstrating the ability in handling production-level traffic loads. A rough estimation reveals that only 6 A100 nodes are required to satisfy GPU requirements of these services, whereas previously, an A100 node could deploy at most 2 inference instances, requiring up to 70 A100 nodes to meet the same demand. By decoupling resource requirements, Prism enables DLRM services to more efficiently borrow multi-GPU training nodes during promotional periods, saving over 90% of GPUs.

6 Discussion and Related Work

DLRM systems. Existing DLRM systems primarily focus on scaling embedding table capacity for enhanced model accuracy [34,36,46], optimizing embedding lookup operations for accelerated inference [55,59,63,65,66], maximizing resource efficiency during training [60,61], developing embedding table pruning techniques [37], and expediting model parameter updates [54]. Only a few works study DLRM provisioning at the datacenter scale. Hercules [33], developed for monolithic servers, efficiently searches the task scheduling space and dynamically provisions the best-matching heterogeneous resources in the presence of diurnally changing load. To our knowledge, DisaggRec [34] is the only work that advocates resource disaggregation for large-scale DLRM serving. Unlike Prism, DisaggRec is a memory-disaggregated system that decouples the deployment of compute and memory, with the aim of addressing the growing memory demands of large-scale DLRMs. DisaggRec is only a prototype evaluated in an emulated memory-disaggregated infrastructure.

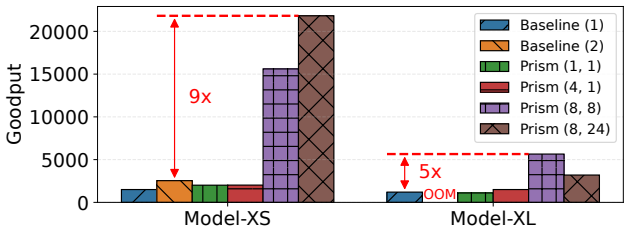


Figure 17: Comparison of HN GPU efficiency, measured by the total goodput. Baseline (count) represents the number of monolithic DLRM instances deployed on the HN, while Prism (CN count, HN count) denotes the number of CN instances and HN instances, respectively.

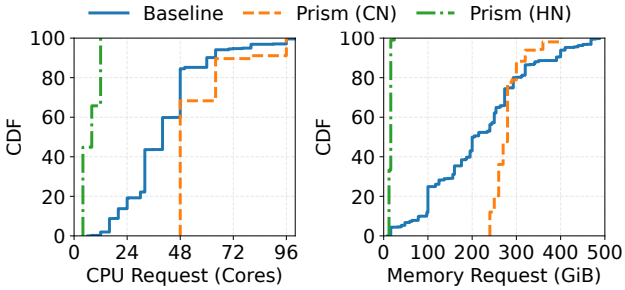


Figure 18: Comparison of resource requirements. Baseline, Prism (CN), and Prism (HN) denote the original monolithic DLRM instances, the disaggregated CN instances, and the disaggregated HN instances respectively. Both the baseline and Prism (HN) instances are allocated one GPU.

Colocation of training and inference workloads. Modern cluster management systems aspire to create a *unified* infrastructure that seamlessly integrate training and inference workloads, optimizing resource multiplexing while minimizing fragmentation. Specifically, Lyra [39] advances this paradigm by leveraging elastic training mechanisms to repurpose low-load inference servers. In this work, Prism reveals fundamen-

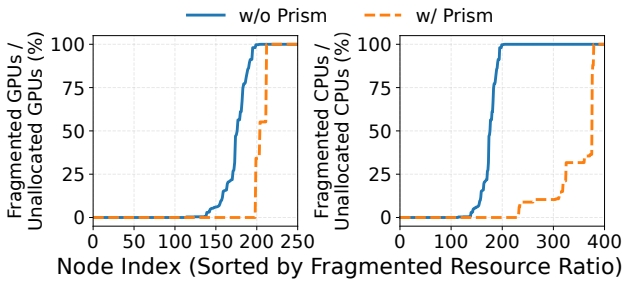


Figure 19: The change in resource fragments of a large-scale GPU cluster *H* before and after resource disaggregation. The original node residual resources are depicted in Figure 3.

Service	Role	# of Instances	CPU	GPU
Product-A	CN	25	48	-
	HN	45	4	MIG 2g.20gb
Product-B	CN	15	48	-
	HN	40	4	MIG 2g.20gb
Product-C	CN	15	48	-
	HN	55	2	MIG 2g.20gb

Table 3: The resource specifications for three online recommendation services. Each GPU mentioned here corresponds to one GPU instance on A100 [5], which includes two GPU compute slices and 20 GiB of GPU memory.

tal inefficiencies in repurposing training clusters for DLRM inference workloads, eliminating resource mismatches through resource disaggregation.

Resource disaggregation. Resource disaggregation holds tremendous promise in datacenters, with many prototype implementations demonstrating benefits of independent scaling of compute and memory resources, improved reliability, and cost-efficient hardware deployment. Existing works span multiple domains, encompassing API and framework innovations [15, 51, 70], operating system and network architectures [18, 53], and hardware design [25, 29, 38]. Prism specifically targets DLRMs, which are characterized by their substantial memory footprints and intensive CPU usage. By disaggregating CPU and GPU provisioning, Prism optimizes resource utilization in heterogeneous GPU clusters. This approach can potentially generalize to other models exhibiting distinct resource usage patterns, including graph neural networks [52] and large language models with retrieval augmented generation [20].

Model parallelism. The idea of model parallelism has been long introduced [35], yet the partitioning of computation graphs across available devices continues to evolve for different model architectures [17, 31, 44, 45, 56]. Prism focuses on DLRMs, which are distinct in their large embedding tables [36, 37]. Moreover, we aim to optimize the inference rather than the training, thus more concerned with the per-request latency rather than long-term throughput.

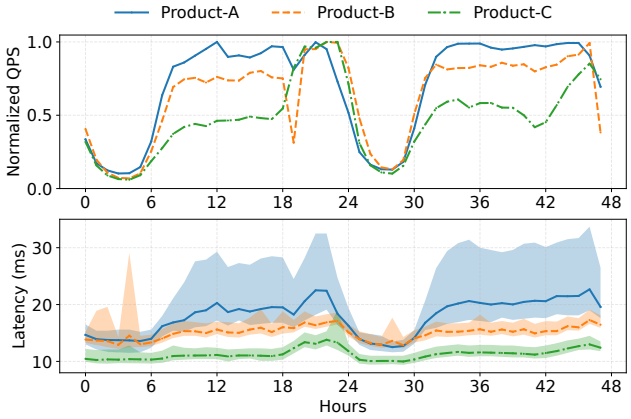


Figure 20: The statistics of three production services (Table 3) during a tremendous traffic promotional period.

Communication optimization with RDMA. RDMA technology has revolutionized datacenter applications across domains, including remote storage systems [10, 19, 43, 62], distributed training frameworks [6, 7, 14], and cluster monitoring solutions [41]. Specifically, previous work [24, 40, 71] attempts to implement high performance and stable RDMA communications from the protocol level. Prism adopts software-level flow control and scheduling strategies, which are more customizable for online services. Prism is complementary to the above RDMA-optimizing techniques.

7 Conclusion

In this work, we propose Prism, the first GPU-disaggregated DLRM serving system to efficiently provision resources at scale. Prism divides DLRMs into CPU- and GPU- intensive subgraph and offloads them to CPU and GPU servers. Prism employs various techniques to minimize the latency overhead, including optimal graph partitioning, topology-aware resource scheduling, and RDMA network optimization. Experimental results demonstrate that Prism effectively separates CPU and GPU computations while maintaining service performance. Prism effectively reduces CPU fragmentation by 53% and GPU fragmentation by 27% in a GPU cluster with a high allocation rate. During seasonal promotional events in e-commerce platforms, Prism can efficiently borrow GPU servers from training clusters to meet peak traffic demands, resulting in up to 90% of GPU savings.

Acknowledgement

We thank our shepherd Seo Jin Park and anonymous reviewers for their valuable comments. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC GRF Grants 16217124 and 16210822, RGC CRF Grants C6015-23G and C7004-22G, and NSFC/RGC Collaborative Research Scheme under the contract of CRS_HKUST601/24.

References

[1] Compute Express Link 3.0. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0_white-paper_FINAL.pdf, 2023.

[2] VMware vSphere Bitfusion. https://docs.vmware.com/en/VMware-vSphere-Bitfusion/index.html, 2023.

[3] Alibaba cluster trace program. https://github.com/alibaba/clusterdata, 2025.

[4] Kubernetes. https://kubernetes.io/, 2025.

[5] NVIDIA A100. https://www.nvidia.com/en-us/data-center/a100/, 2025.

[6] NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl, 2025.

[7] NVIDIA GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2025.

[8] NVIDIA Multi-Instance GPU (MIG). https://docs.nvidia.com/datacenter/tesla/mig-user-guide, 2025.

[9] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proc. USENIX OSDI*, 2016.

[10] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure storage with RDMA. In *Proc. USENIX NSDI*, 2023.

[11] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. Behavior sequence transformer for E-commerce recommendation in Alibaba. In *Proc. ACM DLP-KDD*, 2019.

[12] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. *arXiv preprint arXiv:1606.07792*, 2016.

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

[14] Jianbo Dong, Shaochuang Wang, Fei Feng, Zheng Cao, Heng Pan, Lingbo Tang, Pengcheng Li, Hao Li, Qianyuan Ran, Yiqun Guo, Shanyuan Gao, Xin Long, Jie Zhang, Yong Li, Zhisheng Xia, Liuyihan Song, Yingya Zhang, Pan Pan, Guohui Wang, and Xiaowei Jiang. ACCL: Architecting highly scalable distributed training systems with highly-efficient collective communication library. *IEEE Micro*, 41(5):85–92, 2021.

[15] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. HPCS*, 2010.

[16] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. DGSF: Disaggregated GPUs for serverless functions. In *Proc. IEEE IPDPS*, 2022.

[17] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *Proc. USENIX OSDI*, 2021.

[18] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proc. USENIX OSDI*, 2016.

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

[20] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2024.

[21] Carlos A. Gomez-Uribe and Neil Hunt. The Netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manag. Inf. Syst.*, 6(4), 2016.

[22] Alan Gray. Getting started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with Infiniswap. In *Proc. USENIX NSDI*, 2017.

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

[25] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proc. ACM ASPLOS*, 2022.

[26] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's DNN-based personalized recommendation. In *Proc. IEEE HPCA*, 2020.

[27] Walid A. Hanafy, Limin Wang, Hyunseok Chang, Sarit Mukherjee, T. V. Lakshman, and Prashant Shenoy. Understanding the benefits of hardware-accelerated communication in model-serving applications. In *Proc. IEEE/ACM IWQoS*, 2023.

[28] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. IEEE HPCA*, 2018.

[29] Bowen He, Xiao Zheng, Yuan Chen, Weinan Li, Yajin Zhou, Xin Long, Pengcheng Zhang, Xiaowei Lu, Linquan Jiang, Qiang Liu, Dennis Cai, and Xiantao Zhang. DxPU: Large-scale disaggregated GPU pools in the datacenter. *ACM Trans. Archit. Code Optim.*, 20(4), 2023.

[30] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, and Chita R. Das. Optimizing CPU performance for recommendation systems at-scale. In *Proc. ACM/IEEE ISCA*, 2023.

[31] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proc. MLSys*, 2019.

[32] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Proc. ACM/IEEE ISCA*, 2020.

[33] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S Lee, and Xuan Zhang. Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation. In *Proc. IEEE HPCA*, 2022.

[34] Liu Ke, Xuan Zhang, Benjamin Lee, G. Edward Suh, and Hsien-Hsin S. Lee. DisaggRec: Architecting disaggregated systems for large-scale personalized recommendation. *arXiv preprint arXiv:2212.00939*, 2022.

[35] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[36] Daniar H. Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. EVStore: Storage and caching capabilities for scaling embedding tables in deep recommendation systems. In *Proc. ACM ASPLOS*, 2023.

[37] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, Zeliang Chen, Ellie Wen, Paul Rivera, Jie You, Chun-cheng Jason Chen, and Mosharaf Chowdhury. AdaEmbed: Adaptive embedding for large-scale recommendation models. In *Proc. USENIX OSDI*, 2023.

[38] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proc. ACM ASPLOS*, 2023.

[39] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proc. ACM EuroSys*, 2023.

[40] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. In *Proc. ACM SIGCOMM*, 2019.

[41] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In *Proc. USENIX NSDI*, 2023.

[42] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. SmartIO: Zero-overhead device sharing through PCIe networking. *ACM Trans. Comput. Syst.*, 38(1–2), 2021.

[43] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: The evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proc. ACM SIGCOMM*, 2022.

[44] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *Proc. ICLR*, 2018.

[45] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proc. ICML*, 2017.

[46] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proc. ACM/IEEE ISCA*, 2022.

[47] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[48] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined GPU scheduling. In *Proc. ACM SOSP*, 2023.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, 2019.

[50] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proc. ACM SIGCOMM*, 2022.

[51] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *Proc. USENIX OSDI*, 2020.

[52] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Netw. Learn. Syst.*, 20(1):61–80, 2009.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proc. USENIX OSDI*, 2018.

[54] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A large-scale deep learning recommender system with low-latency model update. In *Proc. USENIX OSDI*, 2022.

[55] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. UGache: A unified GPU cache for embedding-based deep learning. In *Proc. ACM SOSP*, 2023.

[56] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu.

Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *Proc. USENIX OSDI*, 2021.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. NIPS*, 2017.

[58] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in Alibaba. In *Proc. ACM KDD*, 2018.

[59] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. Merlin HugeCTR: GPU-accelerated recommender system training and inference. In *Proc. ACM RecSys*, 2022.

[60] Zheng Wang, Yuke Wang, Jiaqi Deng, Da Zheng, Ang Li, and Yufei Ding. RAP: Resource-aware automated GPU sharing for multi-GPU recommendation model training and input preprocessing. In *Proc. ACM ASPLOS*, 2024.

[61] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. EL-Rec: Efficient large-scale recommendation model training via tensor-train embedding table. In *Proc. ACM/IEEE SC*, 2022.

[62] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *Proc. USENIX OSDI*, 2020.

[63] Yingcan Wei, Matthias Langer, Fan Yu, Minseok Lee, Jie Liu, Ji Shi, and Zehuan Wang. A GPU-specialized inference parameter server for large-scale deep recommendation models. In *Proc. ACM RecSys*, 2022.

[64] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling GPU-sharing workloads with fragmentation gradient descent. In *Proc. USENIX ATC*, 2023.

[65] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An efficient GPU embedding cache for personalized recommendations. In *Proc. ACM EuroSys*, 2022.

[66] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex Bronstein, Ronald Dreslinski, Trevor Mudge, and Nishil Talati. GRACE: A scalable graph-based approach to accelerating recommendation model inference. In *Proc. ACM ASPLOS*, 2023.

[67] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards zero copy dataflows using RDMA. In *Proc. ACM SIGCOMM Posters and Demos*, 2017.

[68] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. Workload consolidation in Alibaba clusters: The good, the bad, and the ugly. In *Proc. ACM SoCC*, 2022.

[69] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proc. ACM KDD*, 2018.

[70] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *Proc. USENIX OSDI*, 2022.

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

[72] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a low-overhead container overlay network. In *Proc. USENIX NSDI*, 2019.

Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production

Jianbo Dong[*], *Kun Qian*[*], *Pengcheng Zhang*[*], *Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, Zhihui Ren, Zhicheng Wang, Bin Luo, Peng Zhang, Yang Liu, Yanqing Chen, Yu Guan, Weicheng Wang, Chaojie Yang, Yang Zhang, Man Yuan, Hanyu Zhao, Yong Li, Zihan Zhao, Shan Li, Xianlong Zeng, Zhiping Yao, Binzhang Fu, Ennan Zhai, Wei Lin, Chao Wang, Dennis Cai*
Alibaba Cloud

Abstract

Despite the success of diagnosis systems in traditional cloud computing, these systems are not suitable for pinpointing faults in AI model training cloud scenarios due to the differences in computing paradigms between traditional cloud computing and model training. As one of the largest cloud providers, we present Aegis, a fault diagnosis system specifically designed for AI model training service. We share our experience in the motivation, design, and evolution of Aegis. Keeping easy-to-deploy as the primary principle, Aegis Phase-1 started by enhancing existing general-purpose diagnosis systems. After several months of evolution, Aegis Phase-2 cogitatively customized the collective communication library for sophisticated failure localization in runtime without modifying customer code. Besides the failure localization, we further equipped Aegis with the capabilities on handling performance degradation and failure checking before delivery. Aegis has been deployed in our production training cloud service for one year. Aegis decreases more than 97% of the idle time wasted by diagnosis, 84% of the training task restart count, and 71% of the performance degradation.

1 Introduction

The AI model training has brought about tremendous revolutions to today's cloud services. In a typical AI model training scenario, the customer (i.e., model designer) submits the AI model as a task to the training cluster maintained by a training cloud service provider (e.g., AWS, Azure and Google). The hosts within the training cluster perform model computation, then synchronize intermediate results via collective communication, then perform model computation, and continue. Eventually, these hosts finish a training process (typically lasting weeks) based on the above synchronization way.

Reliability is important. The reliability of the training cluster is crucial to both the training service provider and customers. If a failure occurs during model training, the entire training process should be restarted, resulting in a significant loss of time and money [36, 45, 60]. As the scale of the AI model increases, maintaining the reliability of the training cluster becomes increasingly more important and challenging.

Why maintaining the reliability is challenging? The large-scale AI model training clusters are equipped with tens of thousands of high-tier GPUs (e.g., NVIDIA A100 [13] and H100 [16]) to cooperate through high-speed network connections (including rail-optimized network [1] and NVLINK [17]) to achieve high-throughput training. Nevertheless, the failure rates of these high-performance components (both hardware and network) are much higher than regular ones such as CPU and Clos network used in general cloud computing scenario (see §2.1.1 for more details). Even worse, due to the synchronization nature of model training, the single-point failures resulting from the high-performance GPUs within one host can lead to a cascading failure across all hosts in this training task. The diagnosis systems used in traditional cloud computing [23–26, 28, 29, 31, 32, 34, 35, 37–43, 46–48, 50–55, 57–59, 61, 63, 64], in principle, localize the root causes by tracing back the source-destination path through the sequence of system component calls such as the 5-tuples or devices of given faults; however, in the model training, task failures spread from a single point to the entire cluster, causing the culprit to be hidden among error reports from all hosts. As a result, the diagnosis systems used for traditional cloud computing are not suitable for localizing the root causes in model training cluster.

The state-of-the-art efforts. To fill this gap, some diagnosis systems for the model training scenarios were proposed, such as SuperBench [60] deployed in Microsoft and MegaScale [36] deployed in ByteDance. SuperBench provides a comprehensive benchmark suite for gray failure checking before the cluster deployment. While useful, it is hard to localize the root causes of faults occurred in the model training runtime. On the other hand, MegaScale identifies failures by monitoring CUDA events from "critical code segments" in customer model. This is only suitable for scenarios where the model designer and model training service providers belong to the same party; otherwise, no model designer (i.e., customer)

* Equal Contribution.

is willing to allow the training service provider to monitor or modify their code. Such an assumption is impractical to public model training service providers.

Our approach: Aegis. As one of the largest model training service providers in this world, we decided to build a practical diagnosis system that achieve our key goal: diagnosing root causes of failures in service runtime without modifying customer code. This goal is important to public model training service providers like us, because (1) we need to cover failures during the entire lifecycle, not only the cluster deployment phase and (2) our diagnosis system should be general and transparent for various customers. To this end, we build Aegis. This paper shares our experience in the motivation (§2), design (§3), and evolution (§4) of Aegis.

Driven by our in-production investigation and motivation (detailed in §2), we started the design of Aegis. In the Phase-1 of Aegis (§4.1), we enhanced our existing diagnosis system with the training-output log and a diagnosis procedure fitting in the large-scale model training scenario. This enhancement enabled us to narrow the entire task-level failure down to specific error devices in major failure cases. For more sophisticated cases, we employ offline diagnosis as the backstop.

While Aegis Phase-1 constructs a comprehensive diagnosis system handling various failures in our model training scenario, once the offline diagnosis is involved, all used hosts in this large-scale AI model training task need to be isolated, greatly complicating the scheduling of the training cluster and harming the overall cluster utilization. Therefore, Aegis evolved to Phase-2 (§4.2). Phase-2 focused on diagnosing most failure cases directly at runtime. We conducted a comprehensive investigation of the possible runtime information sources. Adhering to the principle of minimizing customers' perceptions, we chose to customize the Collective Communication Library (or CCL) to acquire runtime status because of the following two reasons. (1) in mainstream training frameworks (e.g., Megatron [49] and DeepSpeed [8]), CCL is integrated as an independent plugin. Customizing CCL would not introduce any code modification in customers' models or training frameworks. (2) CCL "sits at the boundary" of computation and communication, making it a perfect place for generating ideal failure diagnosis information. Through defining appropriate CCL metrics and constructing the corresponding runtime diagnosis system, we improve the runtime diagnosis ratio from 77% to close 100%.

Besides failure diagnosis, significant performance degradation is another crucial problem. Based on our experience, we further develop the basic-metric correlating diagnosis and enhanced procedure-aware diagnosis to discover the root cause of performance degradation (§5).

Other than solving issues during the training procedure, we note that more than 73% of tasks failed at the initialization phase. This phenomenon indicates errors already existed in the cluster before the model training tasks start. To minimize unnecessary retries from customers, we deploy Check Before Delivery (CBD) to conduct essential checks before delivering hosts to customers (§6).

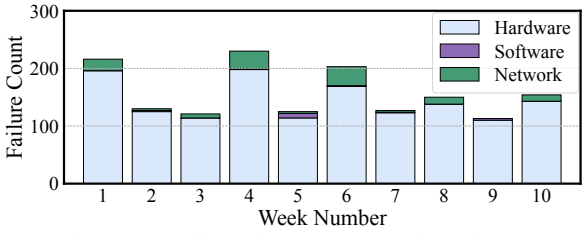


Figure 1: Failures in a representative cluster.

We present the diagnosis efficiency on our training clusters during the evolution of Aegis (§7). For confidential reasons, we use the statistics from our in-house LLM training project as a representation. During the last 16 months, the training scale of our project increased by more than 40×. Impressively, with the deployment of Aegis Phase-1, the GPU idle time caused by failure diagnosis is decreased by 71%. With the deployment of Aegis Phase-2, almost all model training task failures can be diagnosed in runtime. The GPU idle time is further reduced by 91%. For handling performance degradation, with the gradual deployment of the correlation diagnosis and enhanced procedure-aware diagnosis, the degree of performance degradation is decreased by 71%. With the deployment of CBD, the training task restart count is decreased by 84%.

2 Motivation

2.1 Challenges Introduced by Model Training

Large-scale model training is fundamentally different from traditional cloud computing. As shown in Figure 1, we studied the repairing tickets from one of our largest training clusters (containing $O(1K)$ hosts with $O(10K)$ GPUs) in production during the last ten weeks. Each week, 100-230 critical failures occurred in the training cluster, which is orderly higher than that in general cloud computing data centers. If not pinpointed and fixed on time, these failures may contribute to model training task crashes. This subsection presents our findings on why more failures occur and why failure diagnosis is harder in large-scale model training.

2.1.1 Higher hardware failure ratio

Large-scale model training introduces unique challenges, particularly in terms of hardware reliability.

High failure rate of high-tier GPUs. High-tier GPUs (e.g., NVIDIA A100 [13] and H100 [16]), the most fundamental component of the large-scale model training cluster, exhibit a significantly higher failure rate than traditional computing hardware. For instance, our statistical data show that an A100 GPU, on average, fails after approximately 400 days of operation, whereas an H100 GPU has an even shorter mean time to failure of around 200 days. Given a large-scale training task involving thousands of GPUs, this failure rate is orders higher than that of general cloud computing scenarios. As shown in Figure 2, 45.6% of failures are caused by GPU-related reasons (i.e., GPU execution error, GPU driver error, GPU memory error, CUDA error, NVLINK error and GPU ECC error).

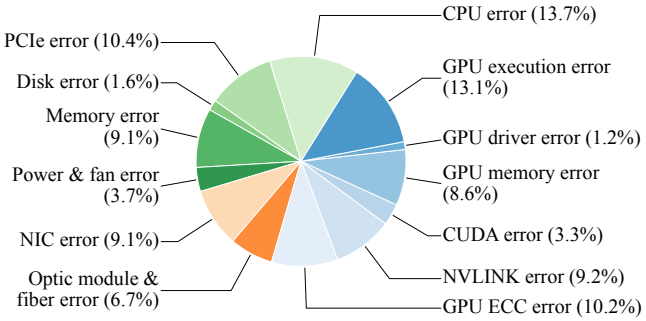


Figure 2: Types of failures encountered in production.

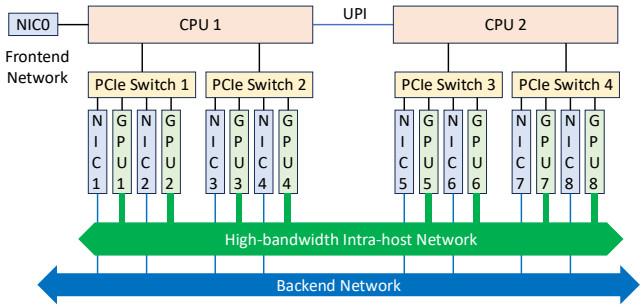


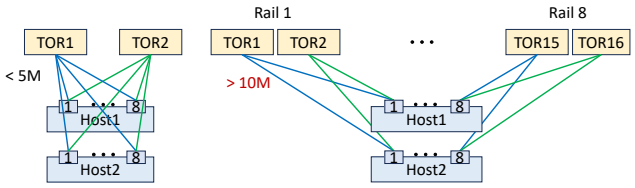
Figure 3: Intra-host network topology.

Complex intra-host network topology. As shown in Figure 3, each host in the our training cluster is equipped with eight GPUs and eight NICs, interconnected via PCIe. GPUs in the same host also utilize NVLINK for inter-GPU communication, which, while enhancing performance, introduces more complex forwarding paths and higher failure rates. Our operational experience has revealed 9.2% issues related to NVLINK failures, as well as 10.4% performance anomalies due to PCIe error. These intra-host network problems are very rare in general cloud computing scenarios.

More link failures in large-scale model training cluster. To maximally utilize the high bandwidth provided by the intra-host network, rail-optimized networking [45] is widely deployed in construction training clusters. While this approach improves performance, as shown in Figure 4, compared with the traditional single-ToR topology, the rail-optimized connections require long-distance links. It necessitates using the optic modules and optic fiber to overcome the distance limitations of copper cables. However, optic modules and fiber have a higher failure ratio [18]. Statistics from our production reveal that optic modules and fiber can introduce a $1.2 \sim 10\times$ higher failure ratio compared with DAC, which varies according to different vendors and link speeds. Although we have implemented the dual-ToR design proposed in HPN [45] to mitigate these issues, the inherently higher failure rates still introduce significant operational challenges. While one of the two dual-ToR links fails, the entire training task may not crash, but may encounter significant performance degradation.

2.1.2 Lacking direct root cause indicators

Besides the high failure ratio, the specific workload of large-scale model training differs from general cloud computing



(a) Single-ToR access (b) Rail-optimized access

Figure 4: Different types of host accessing topology.

tasks, further introducing difficulties in diagnosing the root cause of failures. The limited diagnostic information provided by mainstream training frameworks adds additional difficulty to the failure diagnosis. In traditional general cloud computing scenarios, a failure only affects several specific hosts, with clear fault localization indicators (e.g., the specific 5-tuple connection encounters high RPC completion time). In contrast, in the model training, task failures spread from a single point (e.g., a host or a link) to the entire cluster, causing the culprit to be hidden among all the faulty tasks.

In addition, the entire large-scale model training procedure heavily relies on collective communication to cooperate with all GPUs. Usually, the failure is observed throughout the entire training task crash (e.g., most hosts concurrently encounter the same CCL timeout error). While there are many potential root causes, the CCL timeout is often the first signal to identify the occurrence of a failure. As a result, diagnosing the problem typically begins with the network, but the root cause may not be in the network.

2.2 Why Our Existing Systems Do Not Help?

We have three diagnosis tools used in our data center networks for general cloud computing.

Tool 1: Network monitoring and analysis. During the running of training tasks, each NIC and switch continuously generates logs recording their runtime status, especially warning and error messages. Furthermore, a series of statistic-based monitors are deployed in hosts and switches (e.g., RX/TX throughput, Out-of-order number, ECN mark number on each port). This diagnosis system is primarily designed to collect and analyze logs and statistical data from NICs and switches. By matching specific network failure patterns (e.g., critical errors shown in the log and abnormal statistics), it can automatically identify the failure node and isolate it.

Tool 2: RDMA Pingmesh. In the general cloud computing scenario, we have deployed TCP Pingmesh service on a large scale, and we have transformed it into RDMA Pingmesh during the support of high-performance block storage service, which is similar to R-Pingmesh [41]. RDMA Pingmesh can actively probe the network to diagnose connectivity and high latency issues.

Tool 3: Inband network diagnosis. The above two systems are responsible for end-to-end information to diagnose the faulty device in the training cluster. However, in some complex failure scenarios, we also need hop-by-hop statistics,

rather than end-to-end information, to discover the root causes of failures. We build such a system, which can conduct coloring on specific packets and trace their forwarding status among each single switch.

The procedure: Tool1+Tool2+Tool3. The entire diagnosing procedure (combining the above three tools) works as follows. When application monitors report an abnormal event, this abnormal event points to a clear source-destination pair. Our network monitoring and analysis system is triggered to analyze whether critical errors occur among the source-destination pair. If so, devices with critical errors would be isolated. If there is no such clear conclusion, Pingmesh results are used to verify whether abnormal packet loss and high latency occur in the network. If so, the in-band diagnosis system is triggered to color specific traffic that passes through the source-destination path hop-by-hop to determine the location of the failure.

Limitations. This diagnosis system works well in general cloud computing scenarios; however, it is not suitable in the model training scenario due to the following two reasons.

Reason 1: Focusing on the network itself. In general cloud computing scenarios, a failure notification points to a clear source-destination path. Therefore, existing diagnosis systems only need to focus on identifying faults inside the network. However, serving model training in the cloud is quite different. It heavily relies on collective communication to cooperate with all GPU hosts. Any single failure would cause a cascading task crash across all participating hosts. Therefore, a large number of secondary issue errors may overshadow the root cause error. We encounter numerous false positives in the diagnostic results by directly employing existing diagnosis systems in model training production.

Reason 2: Focusing on single request/response. Our traditional diagnosing systems diagnose faults in the request-response way, focusing on the issues of individual connections. In contrast, large-scale model training requires correlating information from multiple devices to accurately diagnose faults, necessitating an automated localization system that is vacant in the previous procedure.

2.3 Limitations of State-of-the-Art Efforts

State-of-the-art general-purpose diagnosis systems [23–26, 28, 29, 31, 32, 34, 35, 37–43, 46–48, 50–55, 57–59, 61–64] move a step towards handling failures beyond the network. Some of them further consider the correlation between different devices to improve the analysis accuracy [29, 30, 33, 35, 44]. However, due to the lack of taking into account model training-specific features such as collective communication, they are unable to cover many failures in the model training scenario. For example, in LLM training, each single collective communication forces dozens or even hundreds of GPUs in synchrony. Simply extending these correlation-analyzing methods to such a large scale forces the comparison among many nodes. According to our experience, it leads to an increase in unacceptable diagnosis latency and accuracy loss. We, therefore, turn to investigate whether there is any diagnosis system specifically designed for the model training scenario. There are two representative diagnosis solutions built for model training clusters and large-scale deployed in production (i.e., SuperBench [60] deployed in Microsoft and MegaScale [36] deployed in ByteDance). Unfortunately, as shown in Figure 5, both cannot perfectly fit the demands in the model training cloud scenario due to the following reasons.

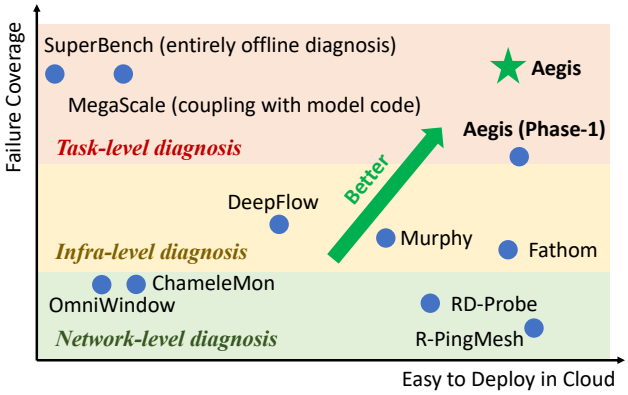


Figure 5: Limitations of state-of-the-art solutions.

Time-consuming offline diagnosis is insufficient. SuperBench constructs a comprehensive benchmark suite, including computation/communication microbenchmarks and typical end-to-end model training benchmarks. By executing this benchmark suite before delivering the training cluster to production, SuperBench can proactively figure out possible issues. We have deployed a similar procedure during each host's online procedure. However, failures may happen during the entire life cycle of the cluster, not only before the online procedure.

It is also impractical to execute SuperBench after any failures. Initializing a single model training benchmark would take tens of minutes, making the entire SuperBench execution lasting hours. Employing this system as the only diagnosis method would introduce a great waste of computation resources and a bad user experience (i.e., prolong waiting time after each failed training).

Deep coupling with customers' code is inappropriate. MegaScale, on the contrary, proposes a fully runtime method to execute failure diagnosis. It monitors the executions of CUDA events in "critical code segments" to diagnose training issues. It is a proper solution for companies that train models for their own use (e.g., ByteDance and Meta). However, it is not suitable for us who offer public AI training services for customers. The reasons are twofold.

First, MegaScale needs a clear definition of "critical code segments". However, as a cloud service provider, we offer training services for various customized models from different customers. Each model may have different architectures and implementations, leading to varying definitions of "critical code segments". Requiring clients to explicitly define

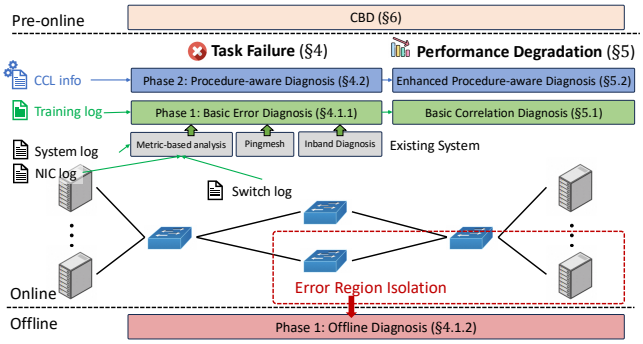


Figure 6: Aegis overview.

the "critical code segments" they use is also impractical, as many model details are highly confidential. Therefore, we need a generic and efficient solution that introduces minimal intrusion on customers.

Second, monitoring CUDA events requires initializing a customized monitor module exactly in the same thread these CUDA events are called. It means this initialization can only be explicitly called in customers' model code. Deploying a new tool that forces modifying customers' model code is hard. The negotiation between our salesman and customers mainly bottlenecks the entire deploying procedure.

3 Aegis Overview

We decided to build a diagnosis system (named Aegis) for the AI model training scenario. Aegis achieves our key goal: diagnosing the culprit of failures in service runtime without modifying customer code. Notice that, in production, when a failure happens, the most urgent issue is timely locating which device is introducing this failure/degradation. Then, we can isolate this device and go on the training. The root cause analysis is conducted offline (i.e., after the isolation), which is another big topic and not covered in Aegis. Figure 6 shows the overview of Aegis. Aegis focuses on two types of abnormalities: training failure and performance degradation. Training failure is the first critical disease we need to cure. Aegis has experienced a two-phase evolution for handling training failure. Considering the limitations of our existing diagnosis system as discussed in §2.2, in Aegis Phase-1, we further involve the error information from the training log and construct a training-specific runtime diagnosis procedure (§4.1.1). We also design a comprehensive offline diagnosis backstop for hardcore cases that cannot be handled in runtime diagnosis (§4.1.2). During the operation, we find that a non-negligible number of failure cases need an offline diagnosis, which leads to unsatisfied GPU utilization. Therefore, we evolve Aegis Phase-2 to get more training-specific information through customizing CCL to conduct the training procedure-aware diagnosis (§4.2). For handling performance degradation, based on previous experience, we design both metric correlation diagnosis (§5.1) and enhanced procedure-aware degradation diagnosis (§5.2). To further improve user experience, we add a new "pre-online" process conducting efficient cluster checking before delivering it to the customer (§6).

4 Task Failure Diagnosis

This section illustrates the evolution of Aegis. We start by enhancing existing diagnosis systems and then move to constructing a better runtime diagnosis system.

4.1 Phase-1: Enhancing Existing Systems

4.1.1 Basic error diagnosis

In most failure cases, there are no clear indicators even to determine the scope of the failure. Therefore, diagnosis begins with identifying whether the current issue originates from the endpoint or the network. At the very beginning of our diagnosis practice, most failures are located through manual diagnosis. Our engineers filter error and warning logs on each host and switch through different sources (e.g., OS dmesg [9], training log, CCL log, NIC driver, switch syslog [5] and customized counters) and analyze the time-sequence relationship between these abnormal reporters with the actual failure. After dealing with hundreds of online failure cases, a series of critical errors are summarized and transformed into our initial automated task failure diagnosing system. Two main challenges are faced and conquered in this procedure.

Not all reported errors are critical. Error logs are widespread during the entire training process. Not all errors deterministically lead to task failures or performance degradation. Considering ECC errors as an example, there are multiple types of ECC errors. Only double-bit ECC errors (e.g., XID 48 Error [21]) and uncorrectable ECC errors (e.g., XID 94/95 Error) are root causes of failures. Other single-bit ECC errors (e.g., XID 92 Error) and correctable ECC errors (e.g., XID 63/64 Error) indicate HBM memory errors, but do not trigger failures. However, widespread errors are reported from different hosts near the task's crash time. If all these hosts were directly isolated, the entire utilization of the training cluster would be greatly damaged.

Our solution first involves identifying critical errors that intrinsically lead to task anomalies, such as GPU missing, PCIe lane drops, or NVLink issues. When such issues are detected, the corresponding host is directly isolated. This method covers only a portion of potential anomalies.

In the early stages of operating LLM training clusters, our engineers devoted much effort to handling online training failures. They summarized other error patterns that certainly lead to task failure. This experience formed the foundation for our critical error diagnosis: $CriticalError()$. Critical errors contain several categories in production: (1) hardware failure (e.g., double-bit ECC error, link down, GPU/NVLink/NIC missing, fan error, and power error) (2) unrecoverable software failure (e.g., GPU/NIC driver error) and (3) unbearable performance degradation (e.g., GPU/host overheat).

Not all critical errors point to a clear location. Besides the clear critical errors, which can directly indicate failure location, many other errors do not point to specific nodes. A

widely encountered example is the crash of network connections (e.g., "connection reset by peer"). These failures can trigger NCCL error handlers, causing the corresponding training thread to exit and leading to cascading thread crashes in other hosts. We also construct a distributed error list to record these errors: *DistError()*.

We enhance our fault diagnosis process with the above experience, as shown in Appendix A (Algorithm 1).

• If a machine encounters critical errors, it is isolated, and the task is restarted.

• If distributed errors appear on only two hosts, it means errors occur on these hosts. They are isolated, and the task is restarted. There is a trade-off behind this process logic. When the potential culprit host set is small enough, it is more efficient to directly isolate all nodes in this list to accelerate the diagnosis procedure. However, the side-effect is that some normal hosts in this set are also isolated, introducing resource wastage. In production, we take the size of this potential culprit host set to 2 to achieve the sweet point.

• For distributed errors across multiple machines, *RootDiag()* analyzes the reported errors to identify if they can be clustered by the source or destination. If GPU G_j is the root cause of the failure, then connections from G_j and connections to G_j crash the first time. Therefore, *RootDiag()* can precisely determine the faulty G_j. If not, it means that the root cause is not on the host-side. Then, network components need to be further checked.

• If errors are distributed without a clear pattern, systemic issues are most likely, such as network or configuration problems. We develop *ConfigCheck()* and *NetDiag()* to conduct further diagnosis. *ConfigCheck()* maintains a checklist and corresponding scripts to identify various causes of configuration errors. *NetDiag()* is constructed by the existing DCN diagnosing system illustrated in §2.2.

• If all the above procedures cannot pinpoint the root cause of the failure, then all hosts used in this training task need to be isolated and conduct offline diagnosis (details in §4.1.2).

Lesson: Exhausting host-side critical failures first is the most efficient way to diagnose. In large-scale model training, host-side issues may be misinterpreted as network issues. In practice, 71% distributed failures turn out to be irrelevant to the network. Therefore, in environments with mixed network-side and host-side faults, solving host-side issues first is important and efficient.

4.1.2 Offline failure diagnosis

After the entire diagnosis procedure, some issues still cannot be directly diagnosed with the available runtime information. To locate the root cause of these failures, we have designed an offline failure localization mechanism. It isolates and diagnoses all suspicious hosts used in the current training task. Unlike SuperBench [60] monopolizes the entire cluster for multiple hours of testing, our offline system diagnoses failures in parallel for targeting hosts. After ensuring each host is problem-free, this host is returned to online service.

Parallelized offline failure localization. To expedite the offline localization process, we design a parallelized approach to maximize efficiency. In the offline localization procedure, all hosts undergo a series of self-checks, including stress testing for CPU, GPU, PCIe, and NVLink. This part is fully parallelized (i.e., each host runs self-checks independently). If an issue is identified during the single-host self-checks, that host is marked as faulty. If no issues are detected in the single-node tests, further multi-host failure diagnosis is required.

The next step is a multi-host failure diagnosis. We select typical models similar in SuperBench [60], and we include more emerging typical models (e.g., MoE models [12] and Multimodal models [4, 20]). The basic idea is that the end-to-end training failure is triggered by a specific combination of computing and communication in the customer's model. Traversing our selected typical models, we can determine which model covers this combination of computing and communication and use it as the reference model in the following multi-host diagnosis. The cluster is then divided into smaller segments, and the reference model is trained on different segments independently, ultimately pinpointing the problematic host. Once a subset of the cluster is confirmed normal, these hosts are promptly returned to the resource pool, minimizing resource wastage.

Topology-aware parallel localization. During parallel localization, the partitioning of different subsets is crucial. Since the network is shared among different hosts, splitting hosts indiscriminately may lead to parallel training tasks competing with the same network links. This can result in two side effects. (1) If the failure occurs within the network (e.g., silent packet loss) and the parallel diagnosing tasks use the same problematic link, both tasks will be affected, making the parallel diagnosing procedure fail to locate the root cause. (2) If the fault is not in the network but on the host, the current running of training tasks may share the same network link and cause congestion, leading to inaccurate diagnosis results.

To handle this issue, hosts are not evenly split into different subsets. We split hosts into two subsets according to their location in the physical network topology. We count the number of hosts in different Pods and ToR groups (in rail-optimized topology, multiple ToRs are used to serve the same host, called a ToR group), respectively. Hosts are split according to whether they belong to the same Pods and ToR groups. As a result, during the parallel diagnosis, traffic from different diagnosing tasks would not interfere with each other in the network.

After training the reference model on two subsets, the subset encountering failure is chosen for further diagnosis. Hosts in the other subset can be returned to online service. With the proceeding of the parallel diagnosis, left hosts would gradually converge to be in the same ToR groups. In this case, the splitting could be arbitrary since any splitting would not lead to network congestion. If both subsets encounter failures, the

root cause exists in both subsets, and then all hosts in these two subsets need to be isolated.

Transforming the missing piece into a new clue. The above procedure can cover the diagnosis of most failure cases. However, there is a missing piece: if the root cause is the misbehavior of Core (Tier-3) switches or Aggregation (Tier-2) switches, this independent parallel diagnosis would miss this root cause of failure (since we delicately minimize traffic passing Core and Aggregation switches).

We indeed encounter this issue in production. There was a failure in a training task occupying 1.5K GPUs. In the offline failure diagnosis, we successfully find a reference model to reproduce the failure. However, when we employ the parallel diagnosis, the failure cannot be reproduced in any subsets of hosts. We are surprised at first since there is a missing piece that the default diagnosis procedure cannot handle. After several times of reproduction and further analysis, we derived insight from this missing piece and concluded that there must be some misbehavior in the Aggregation switches. Through succeeding switch-specific diagnosis, the root cause is clear: silent packet loss happens on one Aggregation switch. But why is this switch error not detected by the $NetDiag()$ in the online diagnosis? We encounter an exceptional kind of silent packet loss, which only drops packets larger than 1KB. As a result, the RDMA Pingmesh system does not throw any error since the size of all probing packets is 64B.

After comprehensive reviews of this failure case, we make several enhancements to our system. (1) We supplement the offline diagnosis to handle this case automatically. (2) We enhance RDMA Pingmesh to cover varied lengths of probes.

4.2 Phase-2: Procedure-aware Diagnosis

Despite significant improvements in failure localization through enhanced legacy systems, we still encounter many cases that need to trigger the offline diagnosis to finally locate the root cause, inevitably leading to considerable computation resource waste. Furthermore, even though most cases can be handled with Aegis Phase-1, we encountered several rare cases where typical models could not reproduce the failures encountered in production. In these cases, to finally solve customers' problems, we had to cooperate deeply with them to reproduce failure cases online and collect more diagnosis information. These failures are caused by systematic reasons, requiring not just basic error information but also training-procedure-specific information to find out the culprit. Therefore, we upgrade our system to diagnose runtime failures using a procedure-aware approach.

4.2.1 What is the ideal solution?

The main difficulty of fully online diagnosis is lacking precise information. Therefore, we enhance our online task monitoring capabilities to provide more valuable runtime information. However, several practical constraints must be considered before implementing such improvements:

High confidentiality. LLM training is a synchronous process, which is also the main reason for the difficulties in failure diagnosis. Accurate localization of such issues requires detailed outputs, and since the root causes vary across different cases, the necessary data for diagnosis also differs. Choosing the right metrics for high-confidence diagnosis is crucial.

Minimal customer modifications. Comprehensive fault localization typically requires extensive metrics collection, which demands tight fusion with customers' code or training frameworks. However, as a model training cloud service provider, extensive modifications to customer code or training framework are impossible in production. The ideal solution is fully transparent to customers.

Low overhead. Adding new information collection and processing should introduce minimal overhead to avoid impacting the main training tasks.

4.2.2 Customizing CCL is the bridge

After comprehensive consideration, we employ a customized CCL as the bridge for enhanced runtime diagnosis. This decision is based on several valuable characteristics of CCL:

First, in mainstream training frameworks (e.g., Megatron and DeepSpeed), collective communication is a modularized component that can be replaced independently. By replacing the CCL, we can collect customized diagnostic information without any change in customers' model codes. Unlike higher layers, which may involve extensive modifications, customizing the CCL offers a more practical solution for diagnostics.

Furthermore, collective communication sits at the boundary of computation and communication. Precise runtime information from this layer can provide clear information about the host-side processing time (computation) and network-side processing time (communication). This information is vital for the localization of faulty devices.

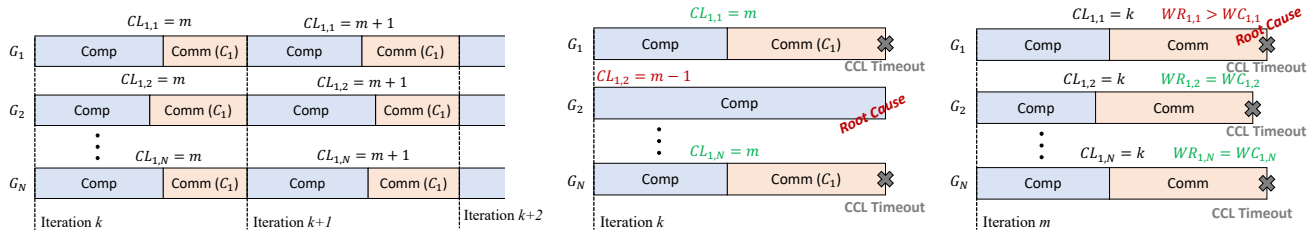
Information collection. During the training, our customized CCL records several statistics of each communication operator (C_i) in each GPU (G_j).

- Collective launch count($CL_{i,j}$) records how many times C_i is launched by G_j.
- Work request count ($WR_{i,j}$) records how many work requests in C_i are launched by G_j.
- Work completion count ($WC_{i,j}$) records how many work requests in C_i are finished by G_j.

We have tried other different metrics in our testbed and found that the information mentioned in our paper is both sufficient and necessary. Keeping collected metrics lightweight and easy to deploy is important.

As shown in Figure 7a, all GPUs alternately execute computation and collective communication in a synchronized way. In normal cases, $CL_{i,j}$ from different G_j keep the synchronized increasing in each iteration. We further present how to localize failures with the above statistics.

Scenario-1: failure in computation. If a failure occurs in the computation phase, as shown in Figure 7b, G_n ($n = 2$ in this case) fail in launching the succeeding C_i ($i = 1$ in this



(a) Normal training iterations. (b) Failure in computation. (c) Failure in communication.

Figure 7: Customizing CCL for failure diagnosis.

case). As a result, all other workers in the same communicator would stall at the C_i and crash owing to CCL timeout. In this case, $CL_{i,n} < CL_{i,j\neq n}$ in the same group. As a result, G_n is pinpointed as the root cause of the failure.

Scenario-2: failure in communication. If the failure is in communication, then the transmission of a specific work request in C_i would fail, causing all GPUs in this group to endure CCL timeout as shown in Figure 7c. More detailed statistics $WR_{i,j}$ and $WC_{i,j}$ are used for further diagnosis. In normal GPUs, $WR_{i,j} = WC_{i,j}$. If $WR_{i,n} < WC_{i,n}$ ($n = 1$ in this case), it means G_n is related to the root cause. We further conduct *NetDiag()* on all sources and destinations related to these abnormal work requests.

After deploying deep runtime diagnosis in production, almost all training failure cases can be pinpointed in runtime (more details in §7).

Limitations. Leveraging information from collective communication is a compromise that is easy to deploy. Actually, just using collective communication information is not enough to find the root cause of failures. However, as aforementioned, collective communication is located at the boundary between computation and communication. This intrinsic characteristic is important for our primary goal: locating the culprit.

As a cloud service provider, our customers may employ various versions of official or self-constructed images with differing CUDA, drivers, and CCL versions. To provide consistent diagnosis ability for all customers, we need to make sure (1) our diagnosis system can seamlessly deploy in different images and model training tasks and (2) solutions deployed in different environments should deliver the same diagnosis information for delivering the consistent diagnosis performance. Therefore, we need to provide our corresponding customized version based on all CCL-released versions. Although the above limitations exist in customizing CCL, it is still easy to deploy compared with other alternatives (e.g., customizing the training framework or modifying customers' model code).

5 Performance Degradation Diagnosis

Besides complete training task failure, some device abnormalities may not crash the entire training but lead to significant performance degradation. These abnormalities should also be diagnosed on time. Since the training task is still running when a performance degradation occurs, we cannot use *OfflineDiag()* as the final fallback solution. Therefore, we design a degradation diagnosis system.

5.1 Basic Correlating Diagnosis

Similar to Phase-1 in §4.1, we first leverage the existing runtime statistics to detect the potential performance degradation.

Key metric selection. The first challenge is to determine precise metrics that can help recognize and diagnose the fault. After solving various cases in production, we notice that most performance degradation is triggered by one single abnormal device, where two categories of metrics can indicate it.

(1) Abnormal operating metrics. These metrics are designed to directly indicate that some components are running at abnormal conditions. For example, the Retran metric denotes how many packets are retransmitted per second. In ordinary cases, this Retran metric should always be zero. The high Retran metric denotes misbehavior in the network.

(2) Performance metrics. These metrics are designed to reflect the execution efficiency of specific components. The abnormal evolution of one component would lead to overall performance degradation. For example, the Actual TensorFLOPS metric denotes how many tensor float-point calculations are completed each second.

We select 20+ metrics according to operating in production including host metrics (e.g., CPU utilization, GPU utilization, GPU temperature, and PCIe utilization) and network metrics (e.g., Bandwidth utilization, Retransmission count, Switch port queue length, and ECN count). Owing to the confidential policy, we cannot release a detailed list of all metrics used. Intuitively, if one metric (especially abnormal operating metrics) is running in a faulty range (e.g., continuously high Retransmission metric), then this value should lead to performance degradation. To filter these clear signals, we set corresponding thresholds for these metrics, respectively.

However, simply using static thresholds cannot fit various training scenarios. Simply setting thresholds for every single metric would lead to numerous misjudgments since resource utilization varies greatly, even in the normal training procedure. Considering that the entire training procedure is well-organized, we, therefore, leverage the synchronizing training characteristics for further correlating diagnosis.

Cross-host correlating diagnosis. The monitored results of the same metric from different hosts should follow the same

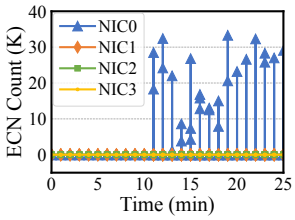
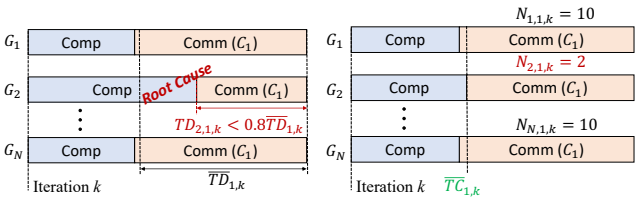


Figure 8: Abnormal ECN metric evolution.



(a) Computation degradation. (b) Communication degradation.

Figure 9: Customizing CCL for performance diagnosis.

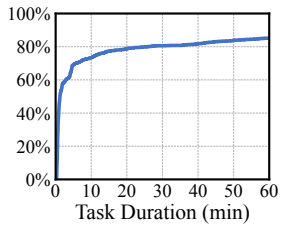


Figure 10: Durations of training tasks in production.

changing pattern among different training iterations. The basic idea is that a small part of nodes causes the entire performance degradation, and performance degradation usually occurs along with abnormal changes in some metrics, which can indicate the root cause. Therefore, we design a Z-Score [22] outlier analyzer for different metrics.

For each selected metric, the outlier analyzer calculates the average value λ and standard deviation δ in the period T. If the metric value from a single host is higher than $\lambda+2\delta$ for the lasting T, then this host is defined as an outlier. T is set to be ten minutes in production. In practice, the abnormal node usually generates significantly different values compared with others, which makes $\lambda+2\delta$ a simple and good enough threshold for diagnosis in production.

Actually, we have tried a series of other outlier analysis mechanisms (e.g., LOF [27], Isolation Forest [3] and DBSCAN [7]). They lead to similar precision and recall ratios. Considering that this entire process needs to run in a stream processing way, the calculation cannot be too complex. We finally choose this simple but efficient outlier analysis mechanism. This correlating diagnosis helps us troubleshoot various malfunctioning devices, including decelerating GPU/CPU/PCIe/NIC/link/switch, which covers a large proportion of failures in production.

Case study. We use an actual case met in production to further illustrate how the cross-host correlation diagnosis runs. During our in-house LLM model's training, as shown in Figure 8, the ECN statistic from one NIC increases from zero to 10-30K per second. At the same time, our model training team reports a 26% training iteration time increase. The correlating diagnosis mechanism immediately identifies this abnormal situation, since this abnormal NIC exceeds $\lambda+2\delta$ of the ECN metric. The root cause is that one of the links connecting to this NIC drops packets silently. It triggers all traffic forward to this NIC through another link, introducing network congestion at the last hop and finally delaying the entire training iteration. Correlating diagnosis immediately pinpoints the root cause of this performance degradation. After isolating the host containing this abnormal NIC and restarting the training task, the training performance returns to normal.

Limitation. This correlating diagnosis can work out the root cause for performance degradation cases in which one or several hosts generate significantly different metrics compared with others. However, this is not always true. It is also common that when performance degradation happens, several metrics change on all hosts, and we are unable to pinpoint the root cause. Therefore, we need to acquire more information and a new mechanism to solve these hard cases.

5.2 Enhancing Procedure-aware Diagnosis

Inspired by the design choice in §4.2, we choose to further customize CCL for more information helping degradation diagnosis. We further record the following statistics of each collective operator (C_i) for each GPU (G_j) in iteration I_k.

- In I_k, the duration of C_i in G_j is $TD_{i,j,k}$.
- In I_k, average duration of C_i is $\overline{TD}_{i,k}$.
- In I_k, the network throughput for the last L ($L=5$ in practice) work requests of C_i in G_j is $N_{i,j,k}$.
- In I_k, average network throughput of C_i is $\overline{N}_{i,k}$.

Figure 9a represents the computation degradation case, where the unexpectedly long duration of the computation is the culprit of performance degradation. Since the end of each collective operation is synchronized, we can utilize the duration of communication time to deduce the computation time. If the $TD_{i,j,k} < \alpha\overline{TC}_{i,k}$ ($\alpha=0.8$ in practice), then G_j is the root cause of the computation degradation. Figure 9b represents the communication degradation case, where the long duration of the communication is the culprit of performance degradation. If $N_{i,j,k} > \beta\overline{N}_{i,k}$ ($\beta=1.5$ in practice), there is a communication degradation. We use a slack threshold here to resist noise caused by possible temporary network congestion. Based on this information, we filter out GPUs group $\mathbb{G}$ directly suffering degradation. $\mathbb{G}$ is further used to determine which source or destination is the root cause of this communication degradation. The principle of this procedure is similar to *RootDiag()* in Algorithm 1 in Appendix A. We omit the details due to limited space.

6 Solving Problems Before Delivery

We analyze the runtime duration of all failed training tasks. As shown in Figure 10, 73% of tasks failed within the first 10 minutes, which is unexpected since the initialization phase of a training task usually takes between 5 to 20 minutes. It indicates that many tasks are failing during the initialization process. It is also in alignment with our experience: many components (both software and hardware) may already endure errors before engaging in the new training task. We have reviewed all failures that occurred during the initialization phase, and there are two main reasons for these failures.

Table 1: CBD task list

Phase	Tasks	Time
Configuration check in parallel	Host configuration check	<1min
	GPU configuration check	
	NIC configuration check	
Single-host test in parallel	GPU kernels test	3min
	NVLink test	
	HBM test	
	PCIe test	
	CPU execution test	
	Dataset/Model/Checkpoint load test	
Multi-hosts test in parallel	Collective communication test	6min
	Comput./Comm. overlap test	

Frequent component updates. Components such as training frameworks, CCL, container networks, NIC drivers, and switches are updated frequently. For example, as released by Meta [19], during the training of LLaMa3, 47 planned updates are executed during a 54-day training snapshot (i.e., 26 times per month). In our model training cloud service, we have a bunch of update requirements for many purposes (including bugfix, safety reinforcements, version unifying and performance optimizations) from many components. To keep the entire service steady, we do not execute updates at such a high frequency. Critical updates, including bugfix and safety reinforcements, are merged to be released weekly. Other updates are released monthly. Even so, updates still trigger many failures in production.

Post-usage failures. If a host encounters a fault after its last use, it will trigger the failure of the new training task during the initialization phase. This issue is especially challenging in cloud environments where hosts are dynamically allocated from a shared resource pool, making failure reproducing much harder. Additionally, once a host is delivered to the customer, the cloud provider cannot run diagnostics on it arbitrarily, making diagnosis much more difficult.

To address these issues, we introduce the Check Before Delivery (CBD) procedure. This check is done right before the resources are handed over to the customers. It brings two main advantages. First, this check process is added at the final stage of resource delivery, so it does not disrupt the existing workflow. Second, by running the validation after the entire environment is set up, we can catch more issues. For example, a connectivity test executed on the physical host will miss connectivity problems caused by incorrect routing configurations in the container network. These issues can only be detected after the container is completely created.

However, CBD has a drawback. Since CBD is called after the environment is fully prepared, it must be in the last phase of delivery. This means CBD needs to be efficient to avoid impacting user experience. Taking these factors into consideration, we designed a set of CBD operations (as outlined in Table 1). We comprehensively select representative tests that cover the main components and organize them to execute in parallel. The entire execution of CBD is less than 10 minutes. If a large number of machines fail in the CBD procedure

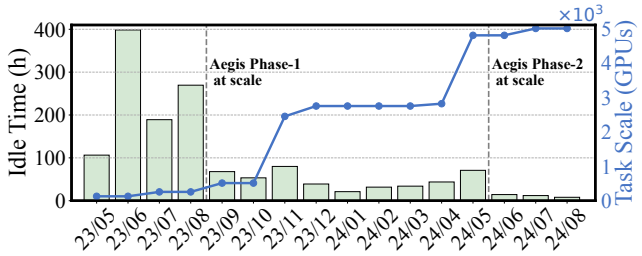


Figure 11: Evolution of idle time in production.

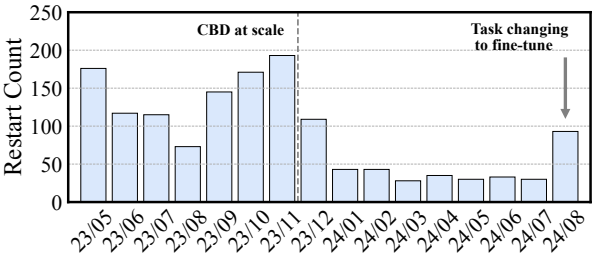


Figure 12: Evolution of restart counts in production.

(reaching a certain threshold), we roll back recent updates to prevent widespread service disruption. There are several different sale modes of the model training service. For the complete PaaS mode, introducing an extra 10 minutes before the startup of each single training task is still unbearable. We also deliver a lightweight CBD version, which only contains parallelized configuration checks and other critical quick local host tests. This lightweight CBD can be completed within 1 minute and can cover most fundamental failures.

With the deployment of CBD, we have intercepted 1-2% problematic hosts before final delivery. If not detected on time, these problematic hosts would lead to training task failures. Considering this significant benefit, we have made CBD a mandatory procedure in delivery.

7 Evaluation: Aegis in Production

The first version of Aegis was online in September 2023, and it immediately became one of the most fundamental components in all succeeding delivery of training services. Aegis has served dozens of large-scale training clusters for more than one year. Due to confidential reasons, we cannot release statistics from our external customers. Therefore, we statistic training task information from our inner model training team, which works to train one of the top-tier LLMs (i.e., large language model) in the world.

7.1 Evolution of Training Stability

As shown in Figure 11, the line records the scale of the training tasks from our inner model training team, which increases by more than 40× during the last 16 months. The bars represent the monthly accumulated idle time of the training tasks owing to the waiting for the failure diagnosis. After Aegis Phase-1 is online, the idle time (time duration where no task running in the training cluster) is decreased by 71% in the next month. This result is impressive, considering that we even doubled the training scale in September 2023. There is

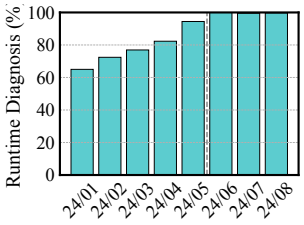


Figure 13: Runtime diagnosis percentage.

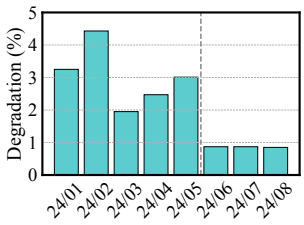


Figure 14: Performance degradation percentage.

an increase of idle time in November 2023, which is owning to a 4× boosting of training scale. The increase in the training scale introduces several unexpected corner case issues, which consume a long time for diagnosis and affect idle time. Aegis Phase-2 is deployed in June 2024, directly leading to 91% save of the training idle time. This improvement mainly comes from the fact that more failures can be solved without involving offline diagnosis.

Figure 12 shows the restart counter of the training task. The increase of the training task scale in November 2023 also triggers more training restart, and a large amount of failures during the initialization phase. We, therefore, speed up the development of CBD and put it into online service in December 2023. It contributes 44.8% decrease in the restart counter in the next month. Through continuously handling more cases and optimizing the checklist, it finally decreased 84.6% of the restart counter. With CBD fully deployed and comprehensively optimized, it discovers around 1-2% problematic hosts before final delivery. Note that the restart number increases in August 2024, the reason is that our model training team switched the task from pre-train to fine-tuning, introducing planned experiments and tests. The root causes of these errors are diverse (e.g., device aging, incorrect configurations, wiring mistakes caused by repairs, complex serving mode switching, etc.). Moreover, since we serve model training requests from multiple tenants, not all tasks are mature and steady-running. Many failures are caused by unoptimized LLM designs or incorrect use of the training infrastructure.

7.2 Runtime Failure Diagnosis

We further answer how many failure cases are diagnosed in runtime (rather than offline) in Figure 13. This metric is important since each offline diagnosis would introduce great damage to the overall GPU utilization and user experience. With the deployment of Aegis Phase-2, the runtime diagnose percentage gradually converges to near 100%. It means that a training task can automatically recover from almost all types of failures without human interference.

7.3 Handling Performance Degradation

To quantify the efficiency of performance degradation diagnosis of Aegis, we deeply cooperate with our model training team to acquire records of iteration time in all training tasks. The iteration time T_k is measured from our model training log. The standard iteration time is calculated as $T_S = 1.2 \times \overline{T_k}$. Performance degradation is calculated as $\frac{\sum_{k:T_k>T_S}(T_k - T_S)}{\sum_{all} T_k}$. The

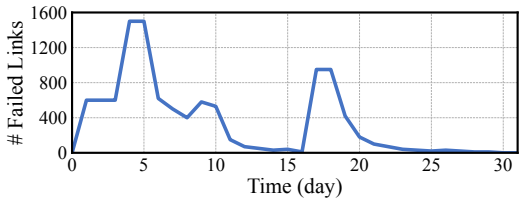


Figure 15: Number of failed links during the batch failure.

statistic results are shown in Figure 14. Performance degradation diagnosis of Aegis is deployed in June 2024. It significantly eliminates performance degradation by 71%.

8 Experience and Lessons

We share key experiences and lessons we learned during the evolution of Aegis as well as the entire reliability guarantee for model training cloud service (additional experience can be found in Appendix B).

Handling batch link failure cases. We have built and delivered dozens of large-scale training clusters. At the delivery of one cluster in them, we encounter an unbearable increase in the link failure ratio ($10-20\times$ higher than the normal link failure ratio). Our investigation revealed that the primary cause was overlapping construction timelines for the data center's building infrastructure and the installation of servers and network wiring. This overlap resulted in significant contamination of optical modules and fibers.

Figure 15 illustrates the change in failure rates over a month. With new machines delivered in batches, the number of failed links increases swiftly due to contaminated links. As cleaning efforts progressed, the failure rate decreased gradually. It takes tens of deep cleaning to completely solve this contamination problem. After identifying this problem, we implemented stricter guidelines for data center construction and delivery to prevent similar issues in the future.

Multiple sale modes and heterogeneous devices in production. Another significant factor affecting the stability of training clusters is the complexity of delivery scenarios. The constant updates to top-tier GPUs (e.g., NVIDIA A100 [13], H100 [16]), NICs (e.g., NVIDIA CX-6 [6], CX-7 [15], BF3 [14]), and switches have led to the deployment of a wide variety combinations of heterogeneous devices in our clusters. This diversity in hardware configurations adds complexity to both the testing and delivery processes, as well as to identifying the root causes of stability issues. Furthermore, to satisfy the diverse needs of our customers, we offer different sales models. For customers focused solely on model optimization, we provide the Platform-as-a-Service (PaaS) sale mode. Therefore, these customers only need to care about constructing novel models and managing their Docker images. On the other hand, for customers who seek further optimization that combines models, training frameworks, and infrastructure, we offer a basic Infrastructure-as-a-Service (IaaS) sale mode. These sale modes involve significant differences in the technology stacks, such as how devices are virtualized and

how the entire cluster is managed. Each mode has its unique set of configurations and delivering procedures, greatly impacting the overall reliability.

Furthermore, in practical operations, we often need to transfer machines originally used in one environment to another, such as relocating servers from one cluster to another with a completely different network architecture or converting a cluster from IaaS to PaaS. These transitions involve numerous configuration changes. Initially, we directly reconfigure all hosts according to the new scenario. However, we encounter several issues where the configurations from the previous setup are not fully compatible with the new one. For example, some modules lack initialization options and only have the overwrite function (e.g., NIC firmware). If the new configurations did not completely overwrite the old ones, residual configurations could remain active, leading to potential issues.

To address these problems, we devote significant effort to thoroughly reviewing and rewriting all configurations to ensure compatibility under different scenarios. By adding a corresponding configuration check in CBD, we ultimately avoid such problems from occurring again.

Traffic pattern evolution triggers the congestion control issue. During the continuous delivery of new training clusters, we encounter a significant issue, where the iteration time increased significantly during the training. Through comprehensive correlation analysis, Aegis identifies the problem but does not find out any abnormal indicators on individual hosts. Further Aegis's runtime analysis based on collective communication information reveals that the performance degradation was due to communication delays caused by many hosts. Despite isolating the abnormal hosts, the training performance can recover, but soon, a similar issue occurs again.

Initially, we suspect that the issue might be related to some faulty switches, since network problems are typically the cause of persistent communication performance issues. However, our thorough investigations show no abnormalities in switch behavior. To resolve this, we conduct extensive offline reproduction and testing. We eventually pinpointed the cause: a bug in the congestion control implementation of NICs. The bug causes an issue where a small number of continuous ECN signals could cause the NIC to enter a preset maximum rate limit. This rate limit is set very low, resulting in significant communication slowdowns.

We report this issue to the NIC vendor, who also confirms this bug. Given that a firmware update to fix the issue would take a considerably long time, we implement two immediate patches in production. (1) We raise the maximum speed limit to alleviate the performance impact. (2) We employ periodic resets of the congestion control state to prevent prolonged performance degradation. We later fix this issue permanently by updating the official bugfix firmware.

Protecting customers' privacy. It is always an important topic for cloud service providers to keep customers' privacy (e.g., training dataset, customized models, training strategies and model checkpoints) well protected. Aegis Phase-1 builds the fault diagnosis ability fully based on existing statistics, introducing no privacy risk. During the design of Aegis Phase-2, to achieve further runtime information for better diagnosis, we investigate and attempt a series of different methods. We have had multiple times of conversations with our product development, solution architect and solution sales teams to figure out the best solution path for Aegis Phase-2. Although for those heavy solutions (e.g., encoding specific statistic functions in customers' models), we can force the deployment of these solutions by re-signing authorization agreements with customers, these solutions would be rejected by most of our customers. After extensive design deliberation, we ultimately opt to enhance CCL.

9 Related Work

Large-scale AI model training diagnosis systems. SuperBench [60] provides a comprehensive benchmark suite for diagnosing issues before deployment, which cannot cover runtime failures. MegaScale [36] monitors CUDA events of "critical code segments" in the model code, which is impractical from the perspective of the cloud service provider. There are some other diagnosis solutions based on infrastructure logs and statistics like monitor_train_log [2] and SageMaker [11], which can only cover a limited range of failures. Dynolog [10] integrates PyTorch profiler to further get code-block level tracing, which however cannot locating root causes in the GPU kernel executions. We have tried this technology roadmap before, but owing to the limited failure coverage, we eventually chose to not deploy them in production.

General abnormality diagnosis systems. In the general cloud computing scenario, various efforts have been devoted to anomaly diagnosis in the host [23–25, 28, 31, 34, 35, 39, 42, 43, 46–48, 50, 52, 55, 57] or in the network [26, 29, 32, 35, 37, 38, 40, 41, 51, 53, 54, 56, 58, 59, 61, 63, 64]. However, as illustrated in §2, without specific optimization focusing model training scenarios, these solutions cannot provide satisfied diagnosing precision for model training specific failure cases.

10 Conclusion

We present the evolution of Aegis, a large-scale model training diagnosis system, in our production. Keeping easy-to-deploy in the cloud as the first principle, Aegis solves training failures and performance degradation cases. Aegis decreases 97% of the idle time wasted by failure diagnosis, 84% of the training task restarts, and 71% of the performance degradation.

Acknowledgements

We acknowledge all teams within Alibaba Cloud that contributed to the success of Aegis, including the High-Performance Network, PAI, Lingjun, Network Automation, Network Operation, Network Systems and Optical Network teams, to name a few. We also thank our shepherd Peng Zhang, and the NSDI reviewers for their insightful comments. Ennan Zhai is the corresponding author.

References

[1] NVIDIA DGX SuperPOD: Next Generation Scalable Infrastructure for AI Leadership. https://docs.nvidia.com/https:/docs.nvidia.com/dgx-superpod-reference-architecture-dgx-h100.pdf, 2023.

[2] OPT-175 Logbook. https://github.com/facebookresearch/metaseq/blob/main/projects/OPT/chronicles/OPT175B_Logbook.pdf, 2023.

[3] Anomaly detection using Isolation Forest – A Complete Guide. https://www.analyticsvidhya.com/blog/2021/07/anomaly-detection-using-isolation-forest-a-complete-guide/, 2024.

[4] CLIP. https://github.com/openai/CLIP, 2024.

[5] Configuring Syslog. https://www.cisco.com/c/en/us/td/docs/switches/metro/me1200/controller/guide/b_nid_controller_book/b_nid_controller_book_chapter_010101.pdf, 2024.

[6] ConnectX-6. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/, 2024.

[7] DBSCAN vs. K-Means: A Guide in Python. https://www.newhorizons.com/resources/blog/dbscan-vs-kmeans-a-guide-in-python, 2024.

[8] DeepSpeed. https://www.microsoft.com/en-us/research/project/deepspeed/, 2024.

[9] dmesg(1) — Linux manual page. https://man7.org/linux/man-pages/man1/dmesg.1.html, 2024.

[10] Dynolog: a performance monitoring daemon for heterogeneous CPU-GPU systems. https://github.com/facebookincubator/dynolog, 2024.

[11] Machine Learning Service - Amazon SageMaker. https://aws.amazon.com/pm/sagemaker/?nc1=h_ls, 2024.

[12] Mixtral-8x7B. https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1, 2024.

[13] NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/, 2024.

[14] NVIDIA BLUEFIELD-3 DPU PROGRAMMABLE DATA CENTER INFRASTRUCTURE ON-A-CHIP. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2024.

[15] NVIDIA CONNECTX-7 400G ETHERNET SMART ACCELERATION FOR CLOUD, DATA-CENTER AND EDGE. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf, 2024.

[16] NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/, 2024.

[17] NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/, 2024.

[18] Ten Reasons to Use Passive DAC Cables in Your Data Center. https://vitextech.com/ten-reasons-to-use-passive-dac-cables-in-your-data-center/, 2024.

[19] The Llama 3 Herd of Models. https://ai.meta.com/research/publications/the-llama-3-herd-of-models/, 2024.

[20] Vision Transformer (ViT). https://huggingface.co/docs/transformers/model_doc/vit, 2024.

[21] Xid Errors. https://docs.nvidia.com/deploy/xid-errors/index.html, 2024.

[22] Z-Score: Meaning and Formula. https://www.investopedia.com/terms/z/zscore.asp#:~:text=Z%2Dscore%20is%20a%20statistical,traders%20to%20help%20determine%20volatility, 2024.

[23] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Geoff Outhred, and Boon Thau Loo. Closing the network diagnostics gap with vigil. In Proceedings of the SIGCOMM Posters and Demos, SIGCOMM Posters and Demos '17, page 40–42, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 419–435, Renton, WA, April 2018. USENIX Association.

[25] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 440–453, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and

Protocols for Computer Communication, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: identifying density-based local outliers. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[28] Rui Ding, Xunpeng Liu, Shibo Yang, Qun Huang, Baoshu Xie, Ronghua Sun, Zhi Zhang, and Bolong Cui. Rd-probe: Scalable monitoring with sufficient coverage in complex datacenter networks. In Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24, page 258–273, New York, NY, USA, 2024. Association for Computing Machinery.

[29] Chongrong Fang, Haoyu Liu, Mao Miao, Jie Ye, Lei Wang, Wansheng Zhang, Daxiang Kang, Biao Lyv, Peng Cheng, and Jiming Chen. Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 31–43, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. SIGCOMM '20, page 390–403, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 139–152, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 139–152, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G2: A graph processing system for diagnosing distributed systems. In 2011 USENIX Annual Technical Conference (USENIX ATC 11), Portland, OR, June 2011. USENIX Association.

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

[35] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. Murphy: Performance diagnosis of distributed cloud applications. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 438–451, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[37] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pages 253–268. USENIX Association, April 2021.

[38] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 421–436, Boston, MA, February 2019. USENIX Association.

[39] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 287–305, Renton, WA, April 2022. USENIX Association.

[40] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies, CoNEXT '16, page 481–495, New York, NY, USA, 2016. Association for Computing Machinery.

[41] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, Xiang Shi, Haohan Xu, Yang Bai, Dongyang Song, Haoran Wei, Bo Li, Yongchen Pan, Tian Pan, and Tao Huang. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24, page 554–567, New York, NY, USA, 2024. Association for Computing Machinery.

[42] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 15–29, Boston, MA, April 2023. USENIX Association.

[43] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 129–143, New York, NY, USA, 2016. Association for Computing Machinery.

[44] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pages 353–366, San Jose, CA, April 2012. USENIX Association.

[45] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[46] Mubashir Adnan Qureshi, Junhua Yan, Yuchung Cheng, Soheil Hassas Yeganeh, Yousuk Seung, Neal Cardwell, Willem De Bruijn, Van Jacobson, Jasleen Kaur, David Wetherall, and Amin Vahdat. Fathom: Understanding datacenter application network performance. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 394–405, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 595–612, Boston, MA, March 2017. USENIX Association.

[48] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 420–437, New York, NY, USA, 2023. Association for Computing Machinery.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. CoRR, abs/1909.08053, 2019.

[50] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '19, page 2828–2837, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Haifeng Sun, Jiaheng Li, Jintao He, Jie Gui, and Qun Huang. Omniwindow: A general and efficient window mechanism framework for network telemetry. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 867–880, New York, NY, USA, 2023. Association for Computing Machinery.

[52] Ming Sun, Ya Su, Shenglin Zhang, Yuanpu Cao, Yuqing Liu, Dan Pei, Wenfei Wu, Yongsu Zhang, Xiaozhou Liu, and Junliang Tang. Ctf: Anomaly detection in high-dimensional time series with coarse-to-fine model transfer. In IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, pages 1–10, 2021.

[53] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with PathDump. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 233–248, Savannah, GA, November 2016. USENIX Association.

[54] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging

with SwitchPointer. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 453–456, Renton, WA, April 2018. USENIX Association.

[55] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 599–614, Boston, MA, February 2019. USENIX Association.

[56] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 599–614, Boston, MA, February 2019. USENIX Association.

[57] Junxiao Wang, Heng Qi, Yang He, Wenxin Li, Keqiu Li, and Xiaobo Zhou. Flowtracer: An effective flow trajectory detection solution based on probabilistic packet tagging in sdn-enabled networks. IEEE Transactions on Network and Service Management, 16(4):1884–1898, 2019.

[58] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and T. S. Eugene Ng. Closed-loop network performance monitoring and diagnosis with SpiderMon. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 267–285, Renton, WA, April 2022. USENIX Association.

[59] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 395–420, Boston, MA, February 2019. USENIX Association.

[60] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, Jithin Jose, Hossein Pourreza, Jeff Baxter, Kushal Datta, Prabhat Ram, Luke Melton, Joe Chau, Peng Cheng, Yongqiang Xiong, and Lidong Zhou. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 835–850, Santa Clara, CA, July 2024. USENIX Association.

[61] Kaicheng Yang, Yuhan Wu, Ruijie Miao, Tong Yang, Zirui Liu, Zicang Xu, Rui Qiu, Yikai Zhao, Hanglong Lv, Zhigang Ji, and Gaogang Xie. Chamelemon: Shifting measurement attention as network state changes. In Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, page 881–903, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pages 293–306, Hollywood, CA, October 2012. USENIX Association.

[63] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 76–89, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 479–491, New York, NY, USA, 2015. Association for Computing Machinery.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A Failure Diagnosis Procedure

Algorithm 1 shows the pseudo-code of the entire Failure Diagnosis Procedure.

B Additional Experience

Dual-ToR really helps, but it migrates failure cases to degradation cases. As aforementioned, we deploy a dual-ToR design in production to conquer the high link failure ratio. This design is very effective. We handle dozens of link-repairing tickets every week, but most of them do not lead to the crash of training tasks. Furthermore, thanks to dual-ToR, we can conduct a hotfix for these faulty links without isolating the host (i.e., the training task can train continuously during the hotfix). We deliver a complete Standard Operating Procedure (SOP) for resident staff in the cluster to conduct the hotfix. They need to check (1) the correctness of both ports' links, (2) the status of both links, and (3) the switch ports'

Algorithm 1 Failure Diagnosis Procedure

```
T //The set containing all hosts used in this training
L_CE = CriticalError(T)//get critical error locations
L_DE = DistError(T)//get distributed error locations
if L_CE.size() > 0 then
    Isolate(L_CE) & Restart the task
else if L_DE.size() <= 2 then
    Isolate(L_DE) & Restart the task
else
    N = RootDiag(L_DE)
    if N! = NULL then
        CheckError(N)
        Isolate(N) & Restart the task
    else//distributed problem
        R_C = ConfigCheck(L_DE)
        R_N = NetDiag(L_DE)
        if R_C||R_N! = NULL then
            Repair(R_C, R_N)
        else
            Isolate L_DE & OfflineDiag(T)
        end if
    end if
end if
```

status before conducting the real link replacement. More than $O(10K)$ link hotfixs are successfully executed.

Although dual-ToR can significantly decrease the possibility of training crashes caused by link failure, it transfers these failure cases into performance degradation cases (since the available network bandwidth is halved). As a result, we devote more effort to handling performance-related issues. As the training does not crash, actually customers can independently determine whether to proactively terminate the current training right after a new checkpoint and restart it, greatly improving the overall user experience.

Reboot or repair? As illustrated in Figure 2, most stability issues ultimately trace back to problems with the fault in hosts. Once a faulty host is isolated, a critical next step is determining whether it can be fixed by reboot or repair. Some issues can be fixed with a reboot, but others necessitate sending the host for repair, which can take several weeks or even months. The primary challenge is that error reports alone cannot definitively indicate whether a problem can be resolved by reboot. For example, an ECC Error could be caused by an unexpected bit flip, which might be fixed by a reboot. On the other hand, an ECC Error could indicate a hardware fault in the HBM, which must be repaired by sending the host back to the vendor.

Given the lack of an efficient method to determine whether the host needs reboot or repair, we have developed a SOP for it. After isolating the faulty host, we first perform a reboot and then conduct a series of hardware stress tests. Unlike CBD, since the host is already isolated, we need a comprehensive (rather than fast) test to make sure the host is back to normal. We first check the current installation settings. If any irregularities are found, the machine is sent to execute the complete re-installation. If all settings are correct, the host further undergoes a thorough computing power, CPU, GPU, Memory, PCIe, NVLink and network check. If all checks pass, the host is returned to the online cluster pool. Otherwise, the host is sent for repair.

Despite the above rigorous tests, we still occasionally encounter corner cases where a host that passes all tests still exhibits faulty issues. To address this, we have implemented an additional backstop: if a host passes the reboot tests three times, but continues to cause failures online, it is mandatory to send for repair. We also engage with our vendors to understand the root cause and update our test cases accordingly. Accurately and efficiently determining whether a host requires repair remains an open issue. Our current approach is practical for production but still leaves room for further refinement and improvement.

PAPAYA Federated Analytics Stack: Engineering Privacy, Scalability and Practicality

Harish Srinivas Graham Cormode Mehrdad Honarkhah Samuel Lurye
Jonathan Hehir Lunwen He George Hong Ahmed Magdy Dzmitry Huba
Kaikai Wang Shen Guo Shoubhik Bhattacharya
Meta

Abstract

Cross-device *Federated Analytics* (FA) is a distributed computation paradigm designed to answer analytics queries about and derive insights from data held locally on users' devices. On-device computations combined with other privacy and security measures ensure that only minimal data is transmitted off-device, achieving a high standard of data protection. Despite FA's broad relevance, the applicability of existing FA systems is limited by compromised accuracy; lack of flexibility for data analytics; and an inability to scale effectively. In this paper, we describe our approach to combine privacy, scalability, and practicality to build and deploy a system that overcomes these limitations. The PAPAYA FA system leverages trusted execution environments (TEEs) and optimizes the use of on-device computing resources to facilitate federated data processing across large fleets of devices, while ensuring robust, defensible, and verifiable privacy safeguards. We focus on federated analytics (statistics and monitoring), in contrast to systems for Federated Learning (ML workloads), and we flag the key differences.

1 Introduction

The scale of distributed systems that allow users to transact and interact have grown massively over the past two decades, prompted by the widespread availability of powerful mobile devices: primarily, smartphones, but also tablets, smart watches, and other wearables. These systems are typically mediated by a centralized organization, such as Google, Apple, Meta, Bytedance, or other players broadly in the "high tech" arena. Providing a positive user experience in such systems relies on their operators monitoring a variety of metrics, and gathering data to inform ML workflows such as recommender systems. Consequently, it is necessary to instrument the system for data collection and analysis of user activities and preferences. We refer to this process as *analytics*.

Increasingly, there are strong motivations to ensure high levels of privacy for analytics. Internally, companies recognize the high expectations of users that their data will be used responsibly, and will be protected from threats. Externally, a range of laws and regulations govern how organizations must manage data safely, including GDPR[1], ePD[2], and CCPA[3]. These mean that analytics systems must have strong protections on how user data is stored and processed. Meanwhile, we design the system to operate effectively, so that monitoring queries can be evaluated accurately and unobtrusively.

In response to these needs, the model of federated computation, and specifically federated analytics (FA), has recently risen to prominence [5]. This model embodies the notion that data gathered by client devices should remain on those devices, and only minimal summary information should be sent from devices to a server, which can compute and release aggregate results. Beyond this data minimization paradigm, additional protections can be instantiated, to ensure: deidentification (the identity of the client is dissociated from the messages they send); security (data is encrypted to prevent it being observed by unauthorised parties while in transit or at rest); and privacy (the results of a query do not disclose information about any individual participant).

Putting this philosophy into practice requires significant levels of systems and algorithm engineering. Careful design choices are needed to ensure that components can cope with the scale of millions to billions of active participants. Components must interact reliably, so that the security and privacy guarantees compose constructively, and do not allow leakage or expose vulnerabilities. The system must also be flexible enough to allow complex queries to be evaluated over subsets of users, whose ongoing active participation is not guaranteed.

1.1 Our Contributions

Variations of federated analytics technology are utilized across the industry, such as by Google for identifying popular items and producing geographic heatmaps [2, 19, 45] and by Apple for autocorrect and user experience purposes [1, 38].

[1] https://gdpr-info.eu/
[2] https://www.edps.europa.eu/data-protection/our-work/
[3] https://oag.ca.gov/privacy/ccpa

However, the first generation of FA systems faced challenges that tend to limit their wider applicability. Some systems suffer from low accuracy due to overly restrictive models of privacy, making them less reliable for deriving valuable insights. Others lack the flexibility that analysts need to obtain the information necessary for making data-driven decisions. Moreover, several early FA solutions do not scale effectively, preventing them from reaching many millions of devices.

In this paper, we present the Federated Analytics stack at Meta that integrates privacy, scale, and practicality in a cohesive architecture. The FA stack enables efficient computation across three zones: (1) Device, (2) Trusted Environment, and (3) Untrusted Orchestrator. We first provide an overview of the challenges and then our approach to solve them.

Practicality and Accessibility to Analysts. Previous FA systems have struggled with limited expressivity, slow query deployment (time from query creation to result), and requiring deep expertise in federated paradigms—barriers to wider adoption. Our approach addresses these challenges as follows:

- **Expressive and Flexible Computational Model.** We provide a powerful yet intuitive model combining local (device-level) data transformations with a minimal set of private cross-device aggregation primitives. Analysts can configure federated computations using familiar SQL queries to define local operations, select cross-device aggregation methods, and specify privacy parameters. This balance of expressivity and simplicity meets most analytical needs without requiring deep technical knowledge. See section 3.2.
- **Ad-Hoc Query Support and Fast Iteration.** A rapid iteration cycle is crucial for an effective analytics platform. With active query management, periodic device synchronization with the server, and batched computations, our system allows analysts to go from query creation to results within a day. This quick turnaround facilitates agile data exploration and decision-making.

Scalability and Resource Consumption. Our system must tackle two aspects: (1) device population—the total number of devices answering a query, which can vary from thousands to billions, and (2) query volume—the number of active queries within the FA system. Additionally, the processing must be stateful, fault tolerant, and mindful of resource usage on constrained devices. See Sections 3.6 and 3.7.

- **One-shot Algorithms.** Unlike Federated Learning, the system is optimized for only one or a very few (constant) rounds of data collection. We express these algorithms with a flexible primitive, namely Secure Sum and thresholding. Furthermore the algorithms are robust to devices dropping out or being unavailable. They also avoid any inter-device dependence, and require that client information can be encapsulated in a single message. We give a discussion of FA algorithms for quantiles in Appendix A.
- **Batch Processing.** To reduce resource use, PAPAYA batches computations on each device. Each device retrieves all relevant queries, executes them and makes one report back, effectively amortising the cost across metrics. See Section 3.6
- **Predictable Query Load.** We randomize the sync and reporting schedules of individual devices to distribute data submission over a defined period, controlled by a system parameter, ensuring a manageable and predictable Query Per Second (QPS) to the TEEs. This is in Section 5.1

It's important to highlight that *one-shot algorithms* with batched processing were critical to minimise the computational and communication cost on resource constrained devices and maximise inclusion of all device types.

Privacy and Security (Section 4). Federated systems inherently offer a baseline of privacy through data minimization, i.e., keeping raw user data on the device, but our stack aims to provide a higher privacy standard through well-defined mathematical guarantees. Given the diversity of differential privacy models, varying trust levels among actors, and evolving data analysis needs, privacy management becomes complex. We address this by centering privacy guarantees around the user's device. Each user's device takes responsibility to ensure that their data is handled properly during its processing and the outputs meet the expected privacy standard.

- **Simple Data Handling Off-device.** TEEs provide a secure environment where the device can verify the behaviour; i.e., how data is handled. We have explicitly designed the TEE code to be simple and use-case agnostic, to ensure a reliable, verifiable, and auditable implementation. Concretely, the only role of this environment is to perform Secure Sum across devices, threshold and apply differentially private noise. With SQL data transformation on device and post processing of the anonymized aggregate on untrusted servers we show that this Secure Sum and threshold functionality is sufficient to support a broadly applicable analytics platform.
- **Device Autonomy and Control.** Each device independently decides which queries to execute and when. It ensures, via remote attestation, that data processing is conducted off-device on a trusted TEE, running authenticated code. This mechanism guarantees that user data is only processed under secure and verified conditions
- **Support for Different Privacy Modes.** Different privacy models, such as local and distributed differential privacy, offer varying trade-offs between privacy and utility. Recognizing there is no one-size-fits-all solution, our stack supports both. For each query, the analyst can choose between on-device or TEE-based noise addition

and configure privacy parameters accordingly. Devices validate these parameters before accepting a query, ensuring that only those queries meeting the user-defined privacy standards are processed.

Production-Grade System and Evaluation (Section 5). Building on standard concepts in Federated Analytics, we developed a cohesive architecture that addresses the limitations discussed above, with a focus on resource efficiency and fault tolerance (Section 3.7). We successfully implemented this architecture at production scale. Beyond system design, we present empirical results from real-world experiments involving a population of nearly 100 million Android devices. These experiments carefully ensure system and data heterogeneity, measuring key performance metrics such as the time required to iterate over devices and data, and the effects of variables like time of day and device usage patterns. The large-scale experiments demonstrate the system's ability to handle challenges such as unpredictable device availability, diverse network conditions, and maintaining a predictable Query Per Second (QPS) rate to the TEEs. To the best of our knowledge, this is the first large-scale evaluation of a production FA system of this magnitude.

Similarities and Differences with Federated Learning. Both FA and FL instantiate the federated model of computation to protect client data. However, they differ significantly in their workloads, each posing distinct challenges to the system. Thus our FA system has fundamental differences in protocol, execution, scalability and hence is implemented separately. Specifically, FL systems are optimized for ML model training, which entails a fairly homogeneous set of tasks around sending a (large) model to clients, and obtaining (large) updates back from clients in the form of gradient updates. This process is repeated many times across small batches of clients (hundreds to thousands). By contrast, the FA system is meant for general purpose analytics on all device types. It optimizes for computations with fewer rounds of interaction, smaller messages from client to server, and participation from more clients (as many as billions). While we can use the FA system to run FL workloads, it would not be a good fit, and vice-versa.

Example use-cases. The system is used for a range of applications in production. Some indicative scenarios include: counting daily and monthly active users of different products, while ensuring that duplicates are not counted repeatedly [24]; identifying popular content (heavy hitters) within different geographic regions; producing heatmaps of density of activity at differing levels of granularity [2]; tracking the tail of response time distributions to ensure that SLAs are met and to raise warnings (see Appendix A); efficient collection of quality statistics (means, counts and variances) on system health metrics that impact user experience [11]; gathering accuracy and calibration metrics on the performance of deployed federated learning systems; populating dashboards for adoption of new features in applications; and reporting results of federated experiments (A/B testing) on different user interface designs.

2 Background: Secure Computation in TEEs

No matter the approach, federated analytics requires client devices to report some information about their data. Therefore, in order to provide robust security and privacy guarantees, it is necessary to rely on *secure computation*. The goal of secure computation is to evaluate functions on distributed inputs while disclosing only the result of the computation and revealing as little information about the inputs as possible. There are two feasible approaches for secure computation that are resilient against a curious or adversarial server and which are suited for a federated workload: secure multiparty computation (MPC) [13] and trusted execution environments (TEEs) [36]. While fully homomorphic encryption (FHE) is often suggested as a third alternative, existing FHE techniques are not flexible or performant enough to operate at scale.

MPC involves the use of techniques like secret sharing to instantiate partially homomorphic encryption across multiple co-operating servers [7]. Meanwhile, TEEs rely on hardware security to ensure that the TEE operates as described and does not leak any information to an external observer. TEEs provide several features for establishing trust that a unit of code has been executed faithfully and privately:

- **Confidentiality**: The state of the code's execution remains secret, unless it explicitly publishes a message.
- **Integrity**: The code's execution cannot be affected by any party, except by the code explicitly receiving a legal input.
- **Measurement/Attestation**: The TEE can prove to a remote party what code (binary) is executing and what its starting state was, ensuring confidentiality and integrity.

Our focus in this paper is on the trust that clients place on the FA system, and in ensuring that raw information does not leak to the analyst or other clients or even the orchestrator. We therefore focus on the security and privacy attached to client messages, and providing verification back to clients that their data is handled correctly. Handling clients who try to subvert the protocol by manipulating their responses to "poison" the results is not in scope for this work—separate mitigations are in place to ensure that clients are only running authorized binaries. TEEs enable us to derive anonymized, aggregated insights from distributed datasets while protecting the data of individual clients. Via attestation, clients obtain proof of confidentiality and integrity before data ever leaves their devices.

Enforcing Security through Remote Attestation. In our deployment, we use Intel SGX hardware to provide TEE functionality in the form of *secure enclaves*, similar to the Papaya FL system and others [18, 27, 38]. One needs to be mindful

about the possible attacks on SGX and apply appropriate mitigations [20]. With client attestation and hosting on public cloud one can limit the attack vectors. While TEEs provide confidentiality and integrity guarantees, these guarantees are valuable only if a client device can obtain proof that it is sending its data to a valid TEE running a known, trusted binary before the data leaves the device. It is possible to do this by using *remote attestation* as follows:

1. Before protocol execution, the TEE code is made available for audit along with the hash of the trusted binary. Ideally, the code is fully open-sourced to allow public scrutiny.

2. The TEE generates an *attestation quote* (AQ). This quote is used to cryptographically verify the initial state of the TEE, the hash of the binary running inside the TEE, the hash of public parameters used to initialize the TEE at runtime, and a Diffie-Hellman (DH) key exchange context used to establish a shared secret with the TEE. Under standard operating assumptions, it is not feasible to forge an AQ.

3. Upon receiving an AQ the client checks (a) the hash of the running trusted binary is the same as the one published with source code, (b) the public parameters used to initialize the TEE at runtime are valid, and (c) the DH key exchange context was actually generated by the TEE. The client aborts if any of these conditions can't be verified.

4. The client uses the DH key exchange context to establish a shared secret with the TEE, and sends its data encrypted.

Via this protocol, each client ensures it is talking to a legitimate TEE running trusted code with acceptable parameters.

3 System Design

Within a global-scale distributed system, we identify multiple types of participating entities. The system is operated for the benefit of many users (ranging in scale from millions to billions). Each user makes use of *client devices* (*clients* for short) to interact with the system. Each user can operate multiple devices, and one device can be shared among multiple users (switching between accounts). For simplicity, we focus on the case where each user uses a primary client device to access the system. The client devices communicate with a number of system servers which deliver the service on behalf of the system operator. Again, we simplify this to focus on client device interactions with a single device-facing server, which in turn will interact with various *backend servers* on the operator side. In order to offer additional privacy guarantees, the system engages additional *trusted aggregators* outside the direct control of the system operator to perform aggregation of client messages. These may be sourced from, e.g., a cloud service provider. Within this setting, trusted *analysts* seek to find the answer to various queries about client performance, such as measuring client latency and user engagement with new features. We seek a solution where client devices retain control over data generated by user actions, sharing only the minimum information necessary to support monitoring queries. This paradigm is *federated analytics*, a.k.a. *FA* [5].

The PAPAYA FA system (Figure 1) comprises three applications: an end-user application that runs on client devices, a TEE-based trusted secure aggregator that can run on a system server or a cloud server, and a server application that runs on a data center server and orchestrates the complete protocol.

3.1 System Overview

We introduce the major components of the federated system. We describe how an analyst can derive aggregated insights from a distributed dataset in a secure, privacy-preserving way:

1. The analyst authors a *federated query*. This has two parts: a SQL-like query that runs on user devices and specifies what data should be uploaded to the server for aggregation; and a server specification that describes how it should aggregate client data and which privacy techniques to use.

2. The analyst publishes the federated query to the *untrusted orchestrating server* (UO), which allocates resources for aggregation and makes the query visible to user devices.

3. The *client runtime* on each user device downloads the federated query spec and retrieves/transforms the relevant data from the on-device data store (see Section 3.4).

4. The client runtime uses a secure channel to upload data to the *trusted secure aggregator* (TSA) running in a TEE.

5. The TSA uses the federated query spec to aggregate the data across user devices before releasing an anonymized, aggregated result to the orchestrating server.

6. The UO uploads the anonymized, aggregated result to a database for consumption by the analyst.

We now present each of the above components in more detail.

3.2 Federated Query

First, we introduce some terminology: *Dimensions* are discrete data attributes representing "group by" columns, in the sense that a federated query will group by the specified dimensions before aggregating within each unique set of dimension values. For example, if a query includes `city` as a dimension, then data points with `city="Paris"` will be aggregated separately from data points with `city="New York"`. *Metrics*

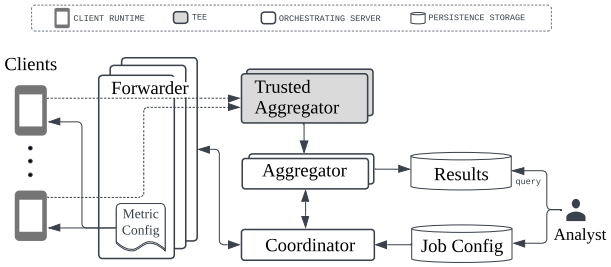


Figure 1: Overall system architecture

```
query:
  onDeviceQuery: "SELECT ...", // SQL to run on device
  dimensionCols: ["city", "day"] // grouping columns
  metricCols: // aggregations (e.g., count, mean, ...)
    mean: ["timeSpent"]
privacy:
  centralDP:
    epsilon: ...
  kAnonThreshold: ...
output: ... // where to persist the anonymized result
```

Figure 2: Example configuration for federated query

are quantitative measurements aggregated across clients after grouping by the dimensions of interest, e.g., total time spent on an app across all users.

For example, suppose an analyst wants to compute average time spent by city and by day of the week. The query is split into two parts:

1. *Data transformation on device*. This is typically defined by a SQL query which runs on the client device. In our example, this local query would retrieve all relevant data on device, group by the dimensions `city` and `day`, and then sum the total time spent for each city-day pair.

2. *Private aggregation across devices*. This defines how to aggregate the reports from different devices under the chosen privacy-preserving regime. In our example, this will tell the trusted secure aggregator to take the average `timeSpent` across devices for each city-day pair, and to apply privacy perturbations before publishing the result.

The schemata of on-device tables are present on server and accessible to analysts. The analyst can configure their query such as in Figure 2. The query result is a table in the data center with one column for each dimension and one column for the metric. In our example, there would be columns for `city`, `day`, and mean `timeSpent`. Each row would contain a unique city-day pair and the average time spent associated with that pair.

With the above framing, the most common analytical queries can be realized with only a handful of secure aggregation protocols—such as `COUNT`, `SUM`, `MEAN`, and `QUANTILE`—in combination with on-device local transformation and downstream post-processing to capture use case-specific needs.

3.3 Untrusted Orchestrating Server

Once the analyst has defined their federated query, it is handed off to the untrusted orchestrating server (UO) to begin execution. The UO has the following responsibilities:

- Provide centralized coordination to ensure queries progress with enough clients active in the face of hardware failures;
- Send active federated queries to clients;
- Facilitate communication between clients and TSAs; and
- Publish query results to persistent storage.

The UO has several sub-components:

- A fleet of *aggregators*. Each federated query is assigned to a single aggregator at a time. The assigned aggregator is responsible for allocating a TSA for the query, requesting periodic results from the TSA, publishing query results to persistent storage and reporting query progress. Each aggregator may be responsible for multiple queries.
- A central *coordinator*, which monitors the state of each federated query, assigns each query to an aggregator and builds the list of active queries to broadcast to clients.
- A *forwarder* layer, which handles incoming client requests and forwards them to the relevant backend components.

After a federated query is registered with the UO, it becomes visible to clients, which retrieve query instructions from the forwarder layer using the client runtime.

3.4 Client Runtime

The client application runs on end-user devices and so needs to be lean, performant and also mindful of the resources consumed by the process. The client runtime (Figure 3) provides:

- A *local store* that securely persists data on the device. It manages data lifetime and scope, and provides the ability to run simple analytic functions over the data.
- An *engine* to execute the client protocol, described below.
- A *scheduler* to monitor the resources consumed and invoke the engine if the device is idle and cumulative resources consumed by the runtime are below a set threshold.

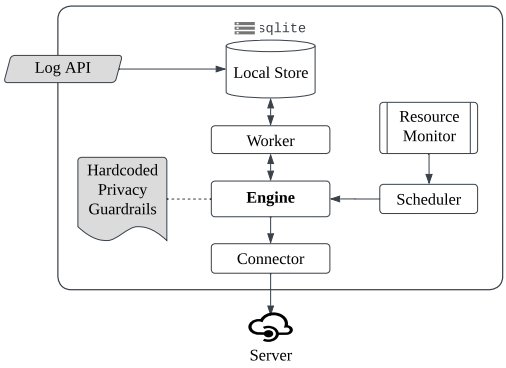


Figure 3: Core components of the Client runtime

The client protocol is split into selection and execution phases:

Selection Phase. Each client periodically fetches the list of active federated queries by polling the UO (subject to a self-enforced daily limit on total resources consumed). Each query configuration in the list has the instructions the client needs to execute locally for each query. Each query configuration may also contain extra information the client uses to decide whether or not it can execute the query. For example, it may include privacy-related parameters to be used for aggregation on the server side, and the client can reject a query if these parameters do not meet locally enforced guardrails (e.g., by capping the number of queries per day, or barring access to certain features). The query configuration could also include a client subsampling rate, where the client uses its own randomness to reject the query with some probability; this can be leveraged to establish stronger privacy guarantees [38]. Last, the client inspects its local state to see if it has any new data to report for each query. After these steps, the client has a list of queries to execute.

Execution Phase. The goal of the execution phase is for the client to calculate the required outputs for each query from the selection phase, then share the outputs with the TSA assigned to each query. The client splits the set of queries into batches. For each task in a batch, the client retrieves and transforms the relevant data; validates the TSA and establishes an encrypted channel via remote attestation; then encrypts its data and sends the encrypted reports to the server, which forwards them to the target TSAs.

3.5 Trusted Secure Aggregator

The TSA is the component of our system that handles the concrete aggregation logic to combine data from different clients. One TSA corresponds to one federated query, though there can be multiple independent TSA instances on the same physical host. The TSA runs inside a TEE and uses remote attestation to establish trust and shared secrets with clients. Clients encrypt their data before sending it to the TSA, so it is the *only* component of our backend that sees plaintext data from individual client reports. Once enough time has passed and enough client devices have reported, the TSA releases an aggregated anonymized result to the UO.

Secure Sum and Thresholding (SST). Many analytical queries can be realized with only a handful of private aggregation protocols such as `COUNT`, `SUM`, `MEAN` and `QUANTILE` (Section 3.2). To further streamline this, our system implements each aggregation primitive via private sparse histograms. Here, *histogram* refers to taking a set of key-value pairs from distributed client devices and outputting a map from keys (or "buckets") to two quantities: the sum of values for the key across all clients with that key, and the count of clients that reported a value for the key. Histograms are one of the most used queries within FA, and many FA queries rely on histograms as a building block, including prefix queries, range queries, heavy hitters, and quantiles. Specifically, these queries use histograms over data with different bucket granularities to build a picture of the data distribution. Histograms allow us to address privacy and scalability with a single aggregation primitive. The TSA running in a TEE allows us to build histograms while maintaining strong privacy and security guarantees via a *Secure Sum and thresholding (SST)* process (Figure 4). This process, which can run from minutes to days, goes as follows:

1. At the start of federated query execution, a TSA is initialized with an empty histogram.
2. Each client validates the TSA and shares encrypted key-value pairs (essentially a "mini" histogram) with the TSA if validation succeeds, using remote attestation (Section 2).
3. The TSA decrypts each client's key-value pairs and immediately aggregates them into the histogram before discarding the individual client data.
4. Once enough clients have reported and enough time has passed (see Section 5.1), the TSA anonymizes the histogram by applying privacy noise to both the sum value and client count value for each bucket. It filters out any buckets with a noisy client count below a threshold specified by the analyst—i.e., if too few clients report a value for some bucket, that bucket will not be released from the TSA (see Section 4.2).
5. The anonymized histogram is released to the UO.

3.6 Handling Scale

For scalability, we employ strategies that address both the volume of devices and the number of queries, while enabling random and unbiased device sampling.

Single-Query Scalability. Employing one-shot algorithms built on Secure Sum and thresholding, each device submits data only once per query, reducing traffic and processing load. To mitigate the "thundering herd" problem, where multiple

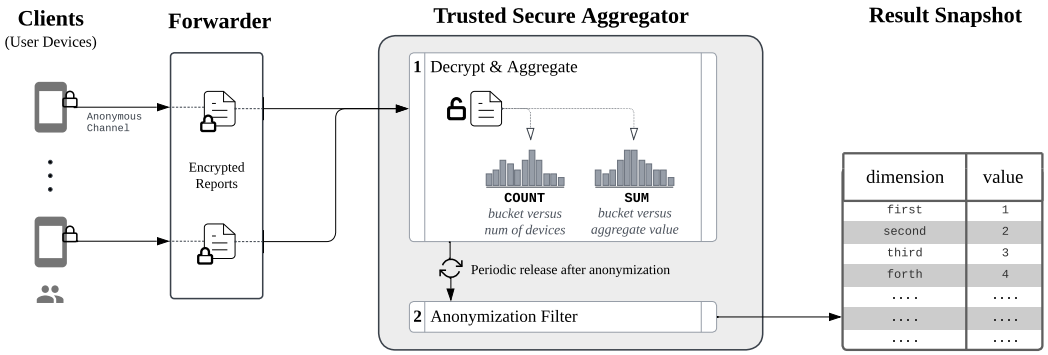


Figure 4: Secure Sum and Thresholding (SST)

devices attempt to connect simultaneously, we randomize the reporting schedules of individual clients. This distributes the data submission over a defined period, controlled by a system parameter, ensuring a manageable and predictable QPS to the TEEs (see Section 5 and Figure 6). The server immediately aggregates incoming reports, using memory proportional to the histogram's dimensions and simplifying the processing involved. Our experiments show a single server is sufficient for one query, but this can be expanded to a tree-level aggregation scheme to distribute the workload.

Multi-Query Scalability. To handle multiple queries, our system batches computations on each device and shards by query on the server. Devices receive all relevant metrics at once, execute their computations, and report back collectively. This setup amortizes process initiation costs and server communication, scaling with query volume.

3.7 Handling Failures

There are multiple potential failure modes, which we classify into two clusters: faulty clients, and faulty servers. For simplicity, we consider faulty connections and clients together, since clients often have unreliable connections that are subject to interruptions. To circumvent this, we break the device work into batches of size ~ 10. This size was empirically determined and can be fine tuned on the respective platform to make sure interruptions are infrequent. If interruptions do happen, we retry during the next period. The client computation is idempotent and will occur again until a successful acknowledgement (`ACK`) for that metric has been received from the server.

This leaves failures that occur on the server side, especially the aggregator and TSA, as they are stateful. Each aggregator-TSA pair stores periodic snapshots of query progress (every few minutes) that would allow a different aggregator-TSA pair to recover intermediate results and resume query execution in case of failure. The coordinator component of the UO can detect fatal query execution errors and will reassign and restart a query on a new aggregator when this occurs. If the coordinator itself fails, a new coordinator instance is started, recovering the previous state from persistent storage.

An important issue here is the privacy of the intermediate aggregation state with partial query results. If the partial results meet the privacy requirements as defined in the query, they can be released and stored in plaintext. Alternatively, the intermediate results can be stored in an encrypted form that is only accessible by another TEE running the same binary as the TEE that generated the results. This is achieved by maintaining a separate group of TEEs responsible for generating, storing and replicating encryption keys. Maintaining robust privacy guarantees is possible with careful bookkeeping and replication logic [28]. Encrypted aggregation state becomes unrecoverable when the associated encryption key is lost, which occurs if and only if a majority of the TEEs with that key fail. This is not a significant obstacle in practice. Because we replicate encryption keys rather than actual data, it is relatively inexpensive to increase the number of replication nodes. As intermediate aggregation state is cumulative, we only need the latest encrypted data. Increasing the frequency of encrypted snapshot generation further reduces the impact of data loss.

A final consideration for the robustness comes from the federated algorithms themselves. In Appendix A, we require that the algorithms do not need participation from all clients, expecting that drop outs will occur as a matter of course. Additionally, if a client is malicious and tries to 'poison' its output the effects are negligible due to (1) its contribution is bounded per report on the TEE prior to aggregation, (2) the typical large scale of operation.

4 Privacy and Security

Secure Aggregation provides a baseline of data protection by insisting that clients do not share raw information, and only aggregated statistics are revealed. We now discuss how data is further protected as it is handled by the FA stack via authentication, access control, and explicit privacy noise addition.

4.1 Secure Data Handling

Storage. Our FA system protects data at rest by design by storing the data on the user's device. The device application manages data scope and lifetime, with encryption and access controls applied to protect data from potential adversaries. Data retention time is configurable with max lifetime (typically 30 days) hard-coded in the application as a guardrail.

Communication. Protecting data during transit is a well-studied problem, and we rely on the standard Transport Layer Security (TLS) protocol to secure network communications. To further minimize leakage of data, communications happen via anonymous authenticated channels, making use of the Anonymous Credentials Service (ACS) library [26, 44]. Thus, the platform is unaware of the identity of the client.

Processing. Secure processing is nuanced due to intricate interactions. Each user's device takes responsibility to ensure that their data is handled properly during its processing. All data from one user is either handled on their device or in an environment where their device can verify the data handling.

Device control over computation. A first component of privacy is admission control. Each device determines which computations to run and when, based on eligibility criteria like previous FA participation, geographic region, hardware type, software version, user features, available data, privacy guardrails, and local randomness. In this way, client devices have total control over their participation in the FA process.

Validation before sharing. Because additional data processing occurs on the TSA server, it is not sufficient that the device controls its own computation. The device must also ensure that the server will handle its data properly and privately. In order for the device to make a fully informed decision about whether to provide data without trusting the UO, the device can use remote attestation as described in Section 2. With this protocol, before sharing any data, the client validates that its data will be processed by a legitimate TEE running a known binary. We have explicitly designed the TEE code to be simple and use case-agnostic, ensuring a reliable, verifiable, and auditable implementation. Thus, prior to any communication, clients obtain proof that their data will be handled as expected.

4.2 Private Handling of Outputs

The previous section discussed how the system provides protections for the data while it is processed in order to compute aggregate statistics. However, aggregating statistics alone is not necessarily sufficient to preserve a strong notion of privacy, as some statistics can still inform on the state of the client. We enforce additional measures in order to meet the higher bar of differential privacy.

Definition 1 ([17]) *Approximate differential privacy (DP) is parameterized by* (ε, δ) *and requires that we define a randomized mechanism* M *such that, for pairs of inputs* X*,* X' *that are neighboring (differing in the presence of a single piece of information) and for any possible output set* O*,* $\Pr[M(X) \in O] \leq \exp(\varepsilon)\Pr[M(X') \in O] + \delta$.

Many randomized DP mechanisms have been defined for different applications; here, we focus on obtaining privacy for a histogram of values, where adding zero-mean Gaussian noise to each count gives (ε, δ)-DP [17].

After adding noise, we apply k-anonymity, where any counts below k are removed from reports. While k-anonymity on its own has been deprecated as a formal privacy tool, this additional step provides an intuitive notion of privacy to our histograms for both users and decision makers—in contrast with the formal DP guarantee, which requires some statistical sophistication to appreciate. Moreover, when histogram dimensions are not known *a priori*, this thresholding step is critical to the DP guarantee [43].

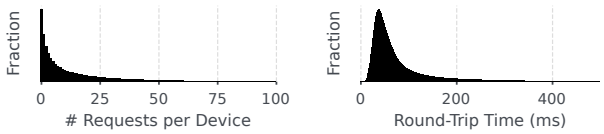
We briefly show DP models which vary in how the notion of neighboring inputs is defined and at what granularity the definition is enforced: centrally or by each client individually.

Central DP at the Enclave. Applying DP via the central aggregator (TEE) is the most straightforward way to obtain DP. The TEE can compute the exact histogram, in the form of a `SUM` or `COUNT` aggregation, then add noise to each value in the bucket of the histogram to achieve differential privacy.

Local DP. Local differential privacy takes the standard DP definition, but applies it to every message from every device. For `COUNT`-queries we can represent the user's input as a 1-hot vector and randomly flip the bits, or pick a value to report from an appropriate probability distribution. The enclave or server aggregates the reports from all devices, and performs a statistical de-biasing step to obtain the estimated histogram.

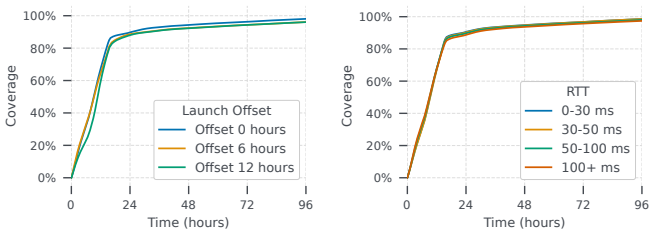
Distributed Privacy Noise. In the distributed DP model, each client adds a small amount of noise, so that after aggregation the noise is sufficient to give the desired DP guarantee. Each client builds a "mini histogram" with the full set of buckets, records its values, and draws random noise to add. The noisy mini histograms are sent to the TSA, which sums them and checks that enough have been received to achieve the central noise requirements. It then performs k-anonymity enforcement and releases the final result. We use the "sample-and-threshold" approach to distributed noise addition, where the uncertainty is introduced due to client randomly deciding whether or not to participate in the data collection [5].

Periodic Data Release. An important consideration in anonymizing the histogram is when results should be released. Client devices have intermittent connectivity and availability; there can be a delay on the order of days before enough participate in FA query to report results with confidence. This poses a challenge to our system, as analysts want results as soon as possible. We balance the desire to give up-to-date results against the privacy cost of making multiple disclosures. The TSA sends partial results to the orchestrating server every few hours. To uphold the privacy guarantee, we limit the num-



(a) Daily values stored per device (b) Round Trip Times (RTT)

Figure 5: Heterogeneity of data: *(a)* number of sampled requests per device, *(b)* round-trip times



(a) Three query runs (b) Coverage by RTT

Figure 6: Coverage of the device population over time: *(a)* for different query execution times, *(b)* by device latency

ber of times the TEE releases partial results. The overall DP privacy parameters (ε, δ) set by the query configuration are budgeted across all releases, using "composition" results [17].

5 Empirical Study

We perform our experimental study using queries in our production deployment of the PAPAYA FA stack. Queries are issued on various system health metrics to ensure smooth running of different systems. Here, we focus on queries of network activity: round trip times (RTT) per network request and request volume. These queries give insights into system and device heterogeneity. They are used to track system performance and are an important canary for regression issues.

Formally, each device i records a set of n_i round trip times $x_{i1}, \ldots, x_{in_i}$, which are aggregated on-device into a local RTT histogram $u_i = (u_{i1}, \ldots, u_{iB})$, where u_{ik} denotes the number of RTT values from device i assigned to bucket k. We compute a federated histogram $v = \sum_i u_i$, making use of the trusted secure aggregator (TSA, Section 3.5). To objectively measure performance, the data points x_{ij} are also stored in a central database (for evaluation purposes only), from which we compute a ground-truth histogram $w = (w_1, \ldots, w_B)$, where w_k denotes the total number of data points in bucket k across all devices. For the device activity histogram, each device has a single data point of interest, n_i (as defined above). So the local histogram produced is a one-hot vector $u_i \in \{0,1\}^B$, i.e., $u_i[j] = 1$ encodes n_i is in bucket j out of B. The rest of the federated histogram collection proceeds as in the RTT case.

Each device has a periodic job to poll the server, compute the local histogram u_i, and report to the TSA. This job has a 10-second timeout, runs in the background, and is run at most twice per day to minimize the burden placed on client devices. Each device also adds individual randomness on when to initiate reporting, to smooth out traffic load.

Queries run on a population of nearly 100 million Android phones that report the relevant data to Intel SGX devices operated by a secure cloud service provider, under the guidance of the UO. Data are gathered at both daily and hourly granularities. The number of data points in the hourly reports is proportionately lower than in daily reports.

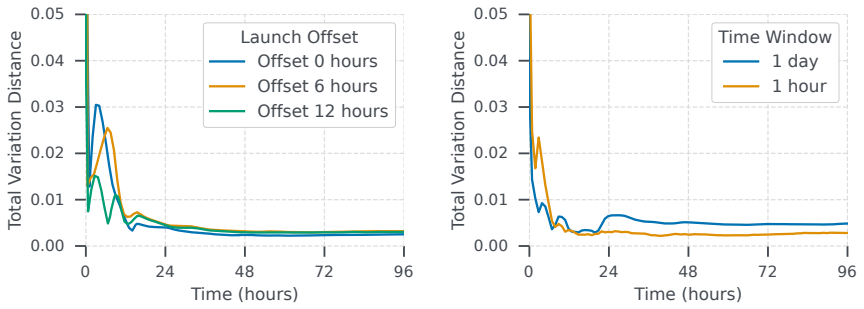
Figure 5 shows the distribution of device data, highlighting two different aspects of heterogeneity. Figure 5a shows that usage patterns differ widely across devices, leading to varying dataset sizes on each device. While the most common case is for clients to have just a single sampled value to report, it is not unusual for them to have tens, with a few having in excess of 100 values to report. Figure 5b shows the differences in network RTT. The mode is around 50 ms RTT, but the distribution stretches out to half a second or more. This reflects variation in device resources and capabilities.

5.1 Collection Speed and Scalability

To begin, we evaluate how long it takes for a task to reach all devices and iterate over the data in its entirety. It is important to note that in practice a random sample of a fraction of the devices or data is considered sufficient to answer most metrics. However, in order to build a thorough understanding of our system, we evaluate the time needed to access all data and devices. We measure wall-clock time from the launch of the query and compute *coverage*, the number of data points processed by the FA task divided by the number of ground-truth points in the database. There are many variables that can impact the collection speed, including time of day (device usage and connectivity patterns vary throughout the day), job restrictions (configuration indicating when a device should not report, based on conditions like network metering, battery status, cumulative quotas, etc.), and other device characteristics (e.g., capabilities, frequency of use, connection quality).

Figure 6 shows how quickly our system accesses the population over different query executions. These queries request the round trip time (RTT) values from devices. Figure 6a shows three executions of the same query launched at six-hour intervals. Encouragingly, results are not much influenced by time of day, and so are similar across all runs.

In fact, the behavior we see in Figure 6 is most strongly influenced by our choice of system parameters: We observe a linear growth in coverage up to around 85% over the initial 16 hours after issuing the query. This reflects that clients check into the server at random, with a uniform delay of 14-16 hours. Coupled with random offsets, this means that active



(a) TVD: different offsets

(b) TVD: Different queries

Figure 7: Accuracy over time: *(a)* for different query execution times, *(b)* for two different queries

client reports are spread uniformly through the initial 16 hour period after the query is issued. This accounts for a majority of clients. The remaining 15% of less active clients check in more gradually, hitting 90% in 24 hours and reaching over 96% coverage after 96 hours, i.e., 4 days from when the query is issued. We should not expect to attain 100% coverage, as a small minority of devices may go fully offline, have local storage reset, or otherwise be unable to report in. For faster collection, we could narrow the check-in window from 16 hours to a lower value. This would speed up the time to reach 85% coverage, but we expect that the "long tail" of less active clients would still require up to a week to check in, due to their sporadic availability.

Figure 6b tests whether the speed of coverage is correlated with network connection quality, by measuring coverage from a single query based on subsets of the histogram corresponding to different ranges of RTT values. The correlations observed are very small, with the coverage rate for low and high latencies following remarkably similar curves. On close inspection, there is a small effect: low latencies have higher coverage than high latencies. This is easiest to see at 16 hours—as time passes, this gap shrinks further.

Through our experiments, we observed that the majority of resource consumption on devices is driven by process initiation and communication with the server, while the actual computation of metrics is comparatively insignificant. This aligns with our hypothesis and design, as unlike Federated Learning, we only execute lightweight SQL queries on the device. Our batched processing setup effectively amortizes these initiation and communication costs, enabling the system to handle many concurrent queries (around 100) efficiently.

5.2 Analysis of Accuracy

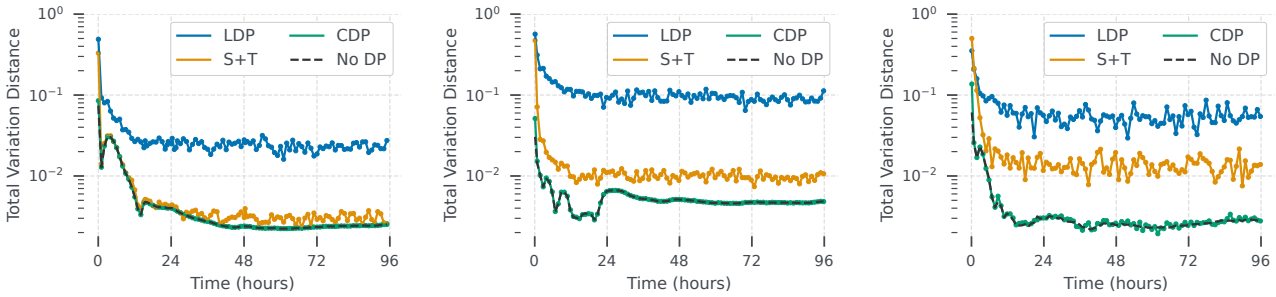
Next we evaluate the accuracy of a metric computed in the federated setting against the same metric computed from the ground-truth dataset. As explained in Section 3.2, all metrics can be computed through data manipulation on the device followed by a handful of aggregation primitives. Since each aggregation primitive is built via the histogram operator using the Secure Sum and Threshold primitive (Section 3.5), measuring the accuracy of a histogram can be used as a proxy for measuring accuracy of any standard metric.

Letting $n_v = \sum_k v_k$ denote the number of data points in the histogram v, the normalized histogram $\bar{v} = \frac{1}{n_v} v$ reflects the relative frequencies per bucket. To plot error between federated histogram v and the ground-truth w, we use total variation distance (TVD) between their normalized versions, $d_{\mathrm{TV}}(\bar{v},\bar{w}) = \frac{1}{2}\|\bar{v}-\bar{w}\|_1 = \max_{S\subseteq\{1,\ldots,B\}} \left|\sum_{k\in S}(\bar{v}_k - \bar{w}_k)\right|$. This straightforward measurement of error reflects the bucket-wise absolute differences between the frequencies observed in $\bar{v}$ vs. $\bar{w}$ for any subset of histogram buckets.

Figure 7 shows the total variation distance across several queries as a function of the time since query launch. In Figure 7a, histograms of round trip time (RTT) are created with $B = 51$ buckets: 0-10 ms, 10-20 ms, ..., 490-500 ms, 500+ ms. These are created with the same time offsets as in Figure 6. In Figure 7b, histograms of device request counts are created at the daily and hourly grain, using $B = 50$ and $B = 15$ buckets (respectively), corresponding to sampled counts of 1, 2, ..., $B-1$, B+. In both cases, we see that it is possible to achieve a very accurate representation: the final TVD is negligible, meaning that the distribution obtained via federated reporting is nearly identical to the ground truth. This is obtained quickly: within 12 hours, an accurate result is found, corresponding to when half of clients have checked in (per Figure 6). Even within a few hours, the result is pretty accurate.

5.3 Analysis of Privacy Mechanisms

To better guide how different privacy guarantees may affect query accuracy, we study how adding differentially private noise impacts the accuracy of histograms. In Section 4.2, we described ways to provide a differential privacy guarantee, based on how and where noise is added in the system: centrally at the secure enclave (central DP, *CDP*), by each client locally (local DP, *LDP*), or in a distributed fashion (sample-and-threshold, *S+T*). We compare generating his-



(a) RTT histogram accuracy (b) Daily event count histogram accuracy (c) Hourly event count histogram accuracy

Figure 8: Experiments on histogram generation with different models of privacy noise addition

tograms in our system under these three different models of privacy via TVD to the ground truth. Each data release from the CDP and S+T privacy mechanisms satisfies (ε, δ)-DP (Definition 1), and each LDP release satisfies $(\varepsilon, 0)$-DP, all with $\varepsilon = 1, \delta = 10^{-8}$.

Recall that against the ground truth, the steady-state total variation distance (Section 5.2) is considerably below $10^{-2} = 0.01$ (Figure 7). Figure 8 shows that there is a clear difference in behavior based on the scale of the input population and the privacy model. When building the histogram of round trip times (Figure 8a), we see that local differential privacy (LDP) achieves an order of magnitude more noise than other methods, and this gap does not decay over time. Nevertheless, this error is in the region of 0.02, which is tolerable for most of our applications. The gap between privacy mechanisms is replicated across other tests (Figures 8b, 8c), where the error for CDP and S+T is much lower than LDP—for CDP, the result is essentially indistinguishable from the un-noised data collected. Results in Figures 8b and 8c consider the accuracy of reporting histograms on the amount of activity recorded on a daily and hourly basis (respectively). Note that the hourly activity counts are proportionately lower than the daily ones. Here, the hourly activity was 34 times lower than the daily activity, as the measurements correspond to a quieter than average period. Hence, the impact of privacy noise introduces more variation for the hourly event counts. This is particularly notable for S+T, where thresholding the sampled data leads to more signal loss. Meanwhile, error introduced by CDP noise addition remains almost imperceptible on these plots.

Thus, for histogram generation, privacy can be achieved with minimal impact on accuracy in FA. The three privacy techniques shown are arguably incomparable, since they exercise different trust models. But even our "weakest" model, CDP, is stronger than simply gathering the data on a regular server, since we use secure TEEs to perform the aggregation and noise addition. Furthermore, the local model of privacy, as previously deployed at scale [1, 19], has the biggest impact on accuracy. So, where permissible, we advocate for central or distributed privacy noise within FA systems.

Additional experiments on the impact of privacy noise of the accuracy for the quantiles (CDF) query are presented and discussed in Appendix A.

6 Related Work

Instrumenting large scale deployments of distributed clients has been studied widely in the distributed systems and networking communities. However, building systems with data privacy as a main goal has enjoyed less focus until recently.

The notion of federated computation has emerged as a distributed paradigm that emphasizes the privacy of the participants [5]. Systems implementing federated *learning* (FL) have been described by Google [8], Apple [31], LinkedIn [41], Snap [32], Meta [27] and others in the big tech arena. FL is primarily concerned with the training of machine learning models, with examples held by the clients used as the training set. Typically, FL workflows have access to a very large number of clients (millions or billions), but engage only a few thousand in each round. In each round, a batch of clients suggest improvements to the current model. This process is repeated over a large number of rounds, often several thousands in total [23]. The duration can thus be days to weeks, allowing for clients that may be slow to respond or only intermittently available. Here, we focus on analytics tasks that are qualitatively different in nature. Rather than concentrate on a single computation task (model training), we seek a system with greater flexibility for processing a range of possible queries. These queries are typically processed in few rounds: ideally, a single round or two. However, we seek to engage many more clients in each round (at least tens of thousands, often millions), and so the latency to complete a round is higher, due to greater variability in client availability and dropouts.

This approach to FA was promoted by Google [33], with an emphasis on gathering information on frequency distributions. The antecedents of this model can be seen in Google's prior RAPPOR system [19] and Apple's differential privacy deployment [1]. Both these systems use local differential privacy to mask client inputs, building on the classical technique

of randomized response [42]. Clients are prompted to provide (perturbed) information on browsing and typing behavior (e.g., frequently visited webpages, new words), in order to improve autocomplete suggestions and detect malware. This information is gathered regularly and perturbed based on a sparse vector encoding with random noise. These first systems demonstrated the utility of private information gathering. However, the level of noise due to local DP is high, requiring a very large population of clients and a loose setting of privacy parameters to get actionable information [39].

Subsequent developments have sought to broaden the range of statistics that can be collected, and to provide stronger privacy and security protections. Work on secure aggregation [4, 7] highlights the power of a primitive that computes the sum of inputs from a large number of clients without revealing any intermediate values. Google's described solution uses interactions between clients to recover from other clients dropping out during the protocol, via cryptographic techniques (threshold secret sharing) and a single coordinating server. Other approaches to implement secure aggregation achieve a similar functionality but remove client-client communications by instead having two or more independent aggregating servers. PRIO, implemented by Mozilla, performs aggregation of inputs from clients based on secret sharing, along with simple zero-knowledge proofs to check that client inputs are well-formed [9, 12, 25]. Apple's sampling-based system adopts this model, where two servers (*leader* and *helper*, as in [21]) interact to perform the aggregation [38]. Systems such as Honeycrisp and Orchard use additively homomorphic cryptography to aggregate client reports across multiple servers [34, 35]. For our implementation, we adopt TEEs for aggregation, to simplify key management and scaling issues, and to be able to handle the thresholding primitive.

Most effort in FA has been directed towards learning frequency distributions of (discrete) values held by clients, such as commonly visited web sites. Since rare values are likely to be privacy-revealing (e.g., URLs that encode identifying information), we seek popular values, or "heavy hitters." Google researchers show that sampling a random (secret) subset of clients can achieve differential privacy [45]. PRIO also targets towards secure computation of heavy hitters, where DP noise can be introduced by the aggregating servers [12, 29]. Other approaches are via distributed noise addition, where each client adds small random noise [2], or via shuffling client responses to disassociate the message from the sender, which has a similar statistical effect [6, 38]. Explicitly suppressing responses with low frequencies captures the intuitive privacy definition of k-anonymity [14].

7 Discussion and Concluding Remarks

We have laid out our experience and insights in building a state-of-the-art large scale FA system. The chief takeaways that we draw from our experience are: (1) It is indeed possible to achieve large scale, secure federated analytics. At this scale, failures are inevitable, and so it was important for us to design with these in mind (e.g, incorporating snapshotting and recovery). (2) Security is increasingly available as a commodity, thanks to availability of secure hardware. Designing for simplicity ensured that security was auditable and efficient for us. (3) A wide range of tasks are served by the Secure Sum and threshold operation, including finding popular values, capturing distributions, and gathering feature usage statistics. (4) We obtained privacy in addition to security by designing in noise addition (differential privacy), and at scale this made minimal impact on accuracy.

Key design choices that could be revisited in future are:

Privacy Models and TEEs. The FA system accommodates various privacy models (central, local and distributed DP). The use of TEEs is of particular importance in allowing complex central DP algorithms, and the secure aggregation for local and distributed DP. Current TEE hardware comes with some tradeoffs, which will alter as new generations are released. Our threat model treats data poisoning efforts as negligible, but protections against malicious clients could be hardened.

Privacy Budgeting. DP parameter ε captures the degree of privacy, which is weakened as more queries are answered. Our pragmatic approach is to focus on per-query privacy impact, and seek to ensure that analysts do not posed repeated queries to the same data. In future we may use stronger constraints, and more sophisticated privacy accounting tools [15, 28].

Longitudinal Issues. Currently, we allow clients to respond to one batch of queries per day, pertaining to data collected over the previous 24 hours. We will loosen these constraints in future versions as we allow more fine-grain responses.

Other considerations. When we try to batch queries together for efficiency, we can combine responses while ensuring that the results do not expose too much private information. For analytics, the main demand is for aggregation queries, future plans are to expand to more types of queries and data. As we extend the deployment of our FA stack to more use cases, we will face new types of queries, and have to decide whether to answer these via custom code, or extend the capability of our system to provide more "built-in" support for common query patterns. A leading concern is how to simplify the deployment of queries by non-expert analysts, and in particular how to provide useful debugging and cross-checking information, while maintaining the high levels of privacy guarantees.

Acknowledgements

Contributions to this project extend beyond the authors. We are grateful for the following people who have engaged in system design/implementation and provided valuable feedback: Ilya Mironov, Jon Millican, Eric Northup, Ravi Ranjan, Anthony Shoumikhin, Pavel Ustinov, Renchang Miao, Vlad Grytsun, Igor L. Markov and Akash Bharadwaj.

A Federated Algorithms: Quantiles

The federated model opens itself to a number of design techniques to answer a wide range of queries. Simple queries, such as sums and counts, are straightforward to implement within FA, while more complex queries require more nuanced solutions. Examples include finding the "heavy hitters" (frequent items) from a large domain [45], estimating U-statistics across pairs of observations [3], finding the mean of scalar and vector values [40], measuring classifier accuracy and AUC [10], and making heat maps of data distributions [2].

For FA algorithms, there is a strong preference for techniques that require only one or a very few (constant) rounds of data collection. This is distinct from Federated Learning, where it is common to iterate towards a solution over hundreds or thousands of short-lived rounds. It is also highly desirable to express the algorithms using well-supported tooling, specifically the SST primitive. We favor algorithms that are robust to clients drop outs and unavailability, via sampling and ensuring that client information is encapsulated in a single message.

To illustrate the design space and considerations of building support for a non-trivial query into a federated analytics system, we study an example query in depth: to find the quantiles of a (distributed) data distribution. Finding quantiles (or percentiles) is one of the most requested queries for the FA stack. We have made several implementations to handle different trust and privacy models, and thus refined our understanding and optimized the approach that we advocate. Our first efforts used multiple rounds of interaction, which slowed down the process, and led to synchronization issues. Subsequently, we focused on approaches using few rounds.

Given a (multi)set of readings, the quantile query seeks the point p such that a q fraction of readings are below it. For instance, the median asks for the 0.5-quantile, where half the readings are below and half are above. We also often seek the 90%-ile, 95%-ile and the 99%-ile corresponding to the 0.9-, 0.95- and 0.99-quantiles. There is noise in the reported answer as we use approximate algorithms and introduce noise for (differential) privacy reasons. Outside of FA, there are algorithms to build compact summaries for the quantiles problem, particularly when processing the inputs as a single stream of data. These include the q-digest [37], t-digest [16], dd-sketch [30] and Greenwald-Khanna (GK) summary [22]. However, these do not all immediately map to the federated setting, nor do they provide a privacy guarantee.

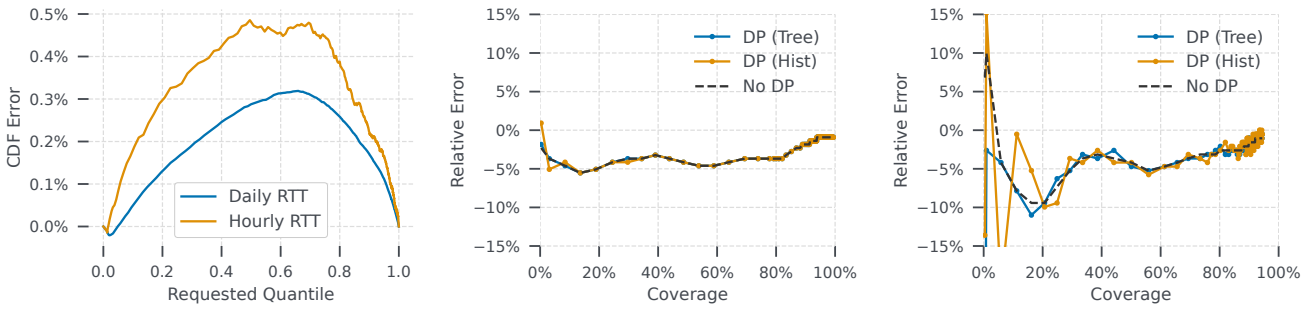
The simplest approach to answering a fixed quantile query in the federated setting is to perform a binary search over multiple rounds. We start with a range $[\texttt{low},\texttt{high}]$ that all the data falls in, and issue a federated counting query to find what fraction of examples fall in this range. This process is repeated for ranges $[\texttt{low}, p]$, adjusting p higher and lower based on the result of the previous rounds, until we find an answer for which the count is close enough to the target q (as a fraction of the number of participating clients). Typically, 8-12 rounds suffice, provided the initial range is fairly tight around the true data. However, this can be slow to complete.

The key to reducing the number of rounds is to think of the multi-round algorithm as looking up values in a collection of histograms of progressively finer granularity. The first round looks up the two values in a two bucket histogram, where the buckets correspond to the left and right halves of the value domain, respectively. Depending on what answer is obtained, the second round looks up two values in a four bucket histogram (corresponding to dividing the domain into four equal pieces), and so on. Importantly, although the set of buckets to inspect depends on what we find in the previous round, the choice of bucket boundaries is not data dependent. Hence, we can build out the complete set of histograms in a single round of FA, and use the output of this query to answer all-quantiles queries. We refer to this as the hierarchical, or *tree*, approach to quantile estimation.

To put this into practice we need some prior knowledge of a range that all values fall into (or a first round of FA to gather this knowledge), in order to specify the histograms. We must determine the number of histograms to build, based on the granularity of the finest histogram. Building histograms out to a depth of 12 (giving 2^{12} buckets at the finest level of detail) gives a good level of accuracy in practice, while remaining relatively quite compact. In some cases, rather than the full hierarchy, it suffices to collect data at the finest level and treat the resulting (noisy) histogram as if it gave the exact distribution. We refer to this as the flat, or *hist*, approach.

A.1 Experiments on Quantile Data Collection

We implement and apply these quantile algorithms in order to evaluate the impact of our design choices on query accuracy and scalability. Figure 9a shows the error in estimating the cumulative distribution function (CDF) of the number of data points per device, based on histograms of $B = 2048$ buckets representing sampled counts of $1, 2, \ldots, 2047, 2048+$. That is, for each potential quantile query (e.g., the median, being the 0.5-quantile), we identify which true quantile the reported value corresponds to, using knowledge of the ground truth distribution. We plot this for both daily and hourly round-trip times, using data collected after 48 hours from initiating the query. The pattern for both is similar: the error is zero at either extreme, since the measure can be satisfied by reporting an arbitrarily small value for the 0.0-quantile, and an arbitrarily large value for the 1.0-quantile. We gain more information when studying quantiles that are closer to the middle of the distribution. After 48 hours of data collection, the maximum error is 0.32% for the daily measurements, and 0.49% for the hourly data—this is the Kolmogorov-Smirnov test statistic for measuring similarity of distributions. In both cases, it is much less than 1%, indicating close agreement. The error is higher in the hourly case, due to fewer observations.



(a) CDF approximation error for RTT

(b) DP error for daily 90%-ile daily RTT

(c) DP error for 90%-ile hourly RTT

Figure 9: CDF error after 48 hours *(left)* and relative error in DP estimation of 90th percentile RTT *(center, right)*

Figures 9b, 9c show our results for estimating the 90th percentile (i.e., 0.9-quantile), under different privacy approaches. Per Appendix A, we add noise either to a flat histogram, *DP (hist)*, or to a hierarchy of histograms, *DP (tree)*. In both cases, we follow a central DP model, adding Gaussian noise to the histograms to achieve an (ε, δ)-DP guarantee for $\varepsilon = 1, \delta = 10^{-8}$. We compare the accuracy of these quantile estimates against a federated quantile estimate without any privacy noise addition (*No DP*). Here, we measure the relative error: we compute the ratio of reported value to the ground truth value of the 90th percentile.

Unsurprisingly, when only a few clients have reported, there is a lot of uncertainty in the value of the target quantile, particularly for the hourly data (Figure 9c). However, once more than about a quarter of clients have submitted their input, we can find a reliable estimate of the quantile, up to a few percentage points in variation. For these experiments, where we have many clients each reporting a single contribution to the histogram, the impact of DP noise is marginal: the greater uncertainty comes from the sampling effect of partial client participation. The tree histogram method adheres closer to the no-DP case than the simple flat histogram. Our practical experience is that the tree approach is preferred when larger histograms are needed to represent the input. We conclude that we can obtain very accurate answers to quantile queries while enjoying FA's strong security and privacy properties.

References

[1] Apple. Apple differential privacy technical overview, 2017. https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf, last accessed 19/07/21.

[2] E. Bagdasaryan, P. Kairouz, S. Mellem, A. Gascón, K. A. Bonawitz, D. Estrin, and M. Gruteser. Towards sparse federated analytics: Location heatmaps under distributed differential privacy with secure aggregation. Proc. Priv. Enhancing Technol., 2022(4):162–182, 2022.

[3] J. Bell, A. Bellet, A. Gascón, and T. Kulkarni. Private protocols for u-statistics in the local model and beyond. In Artificial Intelligence and Statistics, AISTATS, volume 108 of Proceedings of Machine Learning Research, pages 1573–1583. PMLR, 2020.

[4] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In ACM SIGSAC Conference on Computer and Communications Security, pages 1253–1269. ACM, 2020.

[5] A. Bharadwaj and G. Cormode. An introduction to federated computation. In SIGMOD '22: International Conference on Management of Data, pages 2448–2451. ACM, 2022.

[6] A. Bittau, Ú. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnés, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In Symposium on Operating Systems Principles, pages 441–459. ACM, 2017.

[7] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In ACM SIGSAC Conference on Computer and Communications Security, page

1175–1191, New York, NY, USA, 2017. Association for Computing Machinery.

[8] K. A. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards federated learning at scale: System design. In Proceedings of the Second Conference on Machine Learning and Systems. mlsys.org, 2019.

[9] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Lightweight techniques for private heavy hitters. In IEEE Symposium on Security and Privacy, pages 762–776. IEEE, 2021.

[10] G. Cormode and I. L. Markov. Federated calibration and evaluation of binary classifiers. Proc. VLDB Endow., 16(11):3253–3265, 2023.

[11] G. Cormode, I. L. Markov, and H. Srinivas. Private and efficient federated numerical aggregation. In Proceedings 27th International Conference on Extending Database Technology, pages 734–742. OpenProceedings.org, 2024.

[12] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In USENIX Symposium on Networked Systems Design and Implementation, NSDI, pages 259–282. USENIX Association, 2017.

[13] R. Cramer, I. Damgård, and J. B. Nielsen. Secure Multiparty Computation and Secret Sharing. Cambridge University Press, 2015.

[14] A. Davidson, P. Snyder, E. B. Quirk, J. Genereux, B. Livshits, and H. Haddadi. STAR: secret sharing for private threshold aggregation reporting. In ACM SIGSAC Conference on Computer and Communications Security, pages 697–710. ACM, 2022.

[15] V. Doroshenko, B. Ghazi, P. Kamath, R. Kumar, and P. Manurangsi. Connect the dots: Tighter discrete approximations of privacy loss distributions. Proc. Priv. Enhancing Technol., 2022(4):552–570, 2022.

[16] T. Dunning and O. Ertl. Computing extremely accurate quantiles using t-digests. CoRR, abs/1902.04023, 2019.

[17] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. Foundations and Trends in Theoretical Computer Science, 9(3-4):211–407, 2014.

[18] H. Eichner, D. Ramage, K. A. Bonawitz, D. Huba, T. Santoro, B. McLarnon, T. V. Overveldt, N. Fallen, P. Kairouz, A. Cheu, K. Daly, A. Gascón, M. Gruteser, and B. McMahan. Confidential federated computations. CoRR, abs/2404.10764, 2024.

[19] Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In ACM SIGSAC Conference on Computer and Communications Security, pages 1054–1067. ACM, 2014.

[20] S. Fei, Z. Yan, W. Ding, and H. Xie. Security vulnerabilities of sgx and countermeasures: A survey. ACM Comput. Surv., 54(6), July 2021.

[21] T. Geoghegan, C. Patton, E. Rescorla, and C. A. Wood. Distributed Aggregation Protocol for Privacy Preserving Measurement. Internet-Draft draft-ietf-ppm-dap-08, Internet Engineering Task Force, Oct. 2023. Work in Progress.

[22] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In ACM SIGMOD international conference on Management of data, pages 58–66. ACM, 2001.

[23] A. Hard, K. Rao, R. Mathews, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage. Federated learning for mobile keyboard prediction. CoRR, abs/1811.03604, 2018.

[24] J. Hehir, D. Ting, and G. Cormode. Sketch-flip-merge: Mergeable sketches for private distinct counting. In International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA, volume 202 of Proceedings of Machine Learning Research, pages 12846–12865. PMLR, 2023.

[25] B. Holley. Built for privacy: Partnering to deploy oblivious http and prio in firefox. https://blog.mozilla.org/en/products/firefox/partnership-ohttp-prio/, Oct. 2023.

[26] S. Huang, S. Iyengar, S. Jeyaraman, S. Kushwah, C.-K. Lee, Z. Luo, P. Mohassel, A. Raghunathan, S. Shaikh, Y.-C. Sung, and A. Zhang. DIT: De-identified authenticated telemetry at scale, 2021.

[27] D. Huba, J. Nguyen, K. Malik, R. Zhu, M. Rabbat, A. Yousefpour, C. Wu, H. Zhan, P. Ustinov, H. Srinivas, K. Wang, A. Shoumikhin, J. Min, and M. Malek. PAPAYA: practical, private, and scalable federated learning. In Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022. mlsys.org, 2022.

[28] J. Jin, C. Chuengsatiansup, T. Murray, B. I. P. Rubinstein, Y. Yarom, and O. Ohrimenko. Elephants do not forget: Differential privacy with state continuity for privacy budget. In Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, page 1909–1923, New York, NY, USA, 2024. Association for Computing Machinery.

[29] D. Keeler, C. Komlo, E. Lepert, S. Veitch, and X. He. Dprio: Efficient differential privacy with high utility for prio. Proc. Priv. Enhancing Technol., 2023(3):375–390, 2023.

[30] C. Masson, J. E. Rim, and H. K. Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. Proc. VLDB Endow., 12(12):2195–2205, 2019.

[31] M. Paulik, M. Seigel, H. Mason, D. Telaar, J. Kluivers, R. van Dalen, C. W. Lau, L. Carlson, F. Granqvist, C. Vandevelde, S. Agarwal, J. Freudiger, A. Byde, A. Bhowmick, G. Kapoor, S. Beaumont, Áine Cahill, D. Hughes, O. Javidbakht, F. Dong, R. Rishi, and S. Hung. Federated evaluation and tuning for on-device personalization: System design & applications, 2022.

[32] V. Pihur, A. Korolova, F. Liu, S. Sankuratripati, M. Yung, D. Huang, and R. Zeng. Differentially-private "draw and discard" machine learning: Training distributed model from enormous crowds. In Cyber Security, Cryptology, and Machine Learning - 6th International Symposium, CSCML 2022, volume 13301 of Lecture Notes in Computer Science, pages 468–486. Springer, 2022.

[33] D. Ramage and S. Mazzocchi. Federated analytics: Collaborative data science without data collection. https://blog.research.google/2020/05/federated-analytics-collaborative-data.html, 2020.

[34] E. Roth, D. Noble, B. H. Falk, and A. Haeberlen. Honeycrisp: large-scale differentially private aggregation without a trusted core. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 196–210. ACM, 2019.

[35] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. In 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020, pages 1065–1081. USENIX Association, 2020.

[36] C. Shepherd and K. Markantonakis. Trusted Execution Environments. Springer, 2024.

[37] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In International Conference on Embedded Networked Sensor Systems, pages 239–249. ACM, 2004.

[38] K. Talwar, S. Wang, A. McMillan, V. Jina, V. Feldman, B. Basile, Á. Cahill, Y. S. Chan, M. Chatzidakis, J. Chen, O. Chick, M. Chitnis, S. Ganta, Y. Goren, F. Granqvist, K. Guo, F. Jacobs, O. Javidbakht, A. Liu, R. Low, D. Mascenik, S. Myers, D. Park, W. Park, G. Parsa, T. Pauly, C. Priebe, R. Rishi, G. Rothblum, M. Scaria, L. Song, C. Song, K. Tarbe, S. Vogt, L. Winstrom, and S. Zhou. Samplable anonymous aggregation for private federated data analysis. CoRR, abs/2307.15017, 2023.

[39] J. Tang, A. Korolova, X. Bai, X. Wang, and X. Wang. Privacy loss in apple's implementation of differential privacy on macos 10.12. CoRR, abs/1709.02753, 2017.

[40] S. Vargaftik, R. Ben-Basat, A. Portnoy, G. Mendelson, Y. Ben-Itzhak, and M. Mitzenmacher. DRIVE: one-bit distributed mean estimation. In Advances in Neural Information Processing Systems, pages 362–377, 2021.

[41] E. Wang, A. Kannan, Y. Liang, B. Chen, and M. Chowdhury. FLINT: A platform for federated learning integration. CoRR, abs/2302.12862, 2023.

[42] S. L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. Journal of the American Statistical Association, 60(309):63–69, 1965.

[43] A. Wilkins, D. Kifer, D. Zhang, and B. Karrer. Exact privacy analysis of the gaussian sparse histogram mechanism. Journal of Privacy and Confidentiality, 14(1), 2024.

[44] R. Zhang and H. Xiong. Open-sourcing Anonymous Credential Service. https://engineering.fb.com/2022/12/12/security/anonymous-credential-service-acs-open-source/. Accessed: 2024-04-24.

[45] W. Zhu, P. Kairouz, B. McMahan, H. Sun, and W. Li. Federated heavy hitters discovery with differential privacy. In International Conference on Artificial Intelligence and Statistics, volume 108 of Proceedings of Machine Learning Research, pages 3837–3847. PMLR, 2020.

HA/TCP: A Reliable and Scalable Framework for TCP Network Functions

Haoyu Gu
University of Waterloo

Ali José Mashtizadeh
University of Waterloo

Bernard Wong
University of Waterloo

Abstract

Layer 7 network functions (NFs) are a critical piece of modern network infrastructure. As a result, the scalability and reliability of these NFs are important but challenging because of the complexity of layer 7 NFs. This paper presents HA/TCP, a framework that enables migration and failover of layer 7 NFs. HA/TCP uses a novel replication mechanism to synchronize the state between replicas with low overhead, enabling seamless migration and failover of TCP connections. HA/TCP encapsulates the implementation details into our replicated socket interface to allow developers to easily add high availability to their layer 7 NFs such as WAN accelerators, load balancers, and proxies. Our benchmarks show that HA/TCP provides reliability for a 100 Gbps NF with as little as 0.2% decrease in client throughput. HA/TCP transparently migrates a connection between replicas in 38 µs, including the network latency. We provide reliability to a SOCKS proxy and a WAN accelerator with less than 2% decrease in throughput and a modest increase in CPU usage.

1 Introduction

Middleboxes are ubiquitous in modern networks. Providers often deploy middleboxes or network functions through network function virtualization (NFV) to simplify their management and deployment by freeing them from physical appliances. Layer 7 NFs are a growing area in industry and academia that includes caching proxies [3, 30], TLS termination [40], TCP splicers [14, 36, 37], and WAN accelerators/optimizers [44, 48]. These NFs provide complex network services that process and transform traffic in non-trivial ways at critical points in the network. For example, Cloudflare uses layer 7 proxies to improve performance and security for web services. As a central point for incoming traffic, every outage translates to a financial loss and a reputational loss for both customers and service providers [19, 52].

Layer 7 NFs compound the well known reliability and scalability concerns of users of layer 2–3 NFs [45]. Many layer 7 NFs perform transformations, e.g., traffic encapsulation and TLS termination, that prevent using fail-to-wire to temporarily bypass the NF to avoid service outages. Disruptions of these NFs are problematic as connections are terminated inside the NF [36], making failures visible to end nodes.

To support migration and failover, these NFs must migrate all states including, the TCP stack and any data currently being processed by the NF. Layer 7 NFs are unique because they read-modify-write many state variables throughout their software stack (e.g., TCP, TLS, and NF state) multiple times during traffic processing. State changes must be replicated immediately to allow a replica to take over connections seamlessly when failures occur.

Prior approaches to NF reliability, including those of S6 [47], Stateless-NF [31], and FTC [21], cannot be applied to layer 7 NFs. These prior systems commonly assume that layer 2–3 NFs modify at most a few state variables. Most of these systems store the state remotely for migration and failover, and it is practical because most NFs only access or modify 1–2 state variables per flow or per packet. Layer 7 NFs have complex state that is accessed multiple times throughout the packet processing pipeline, necessitating multiple remote accesses to the state that is more costly than existing systems are designed to handle.

This paper introduces HA/TCP, the first framework to support the migration and failover of TCP-based layer 7 NFs for reliability and multi-node scalability. HA/TCP actively replicates traffic from primary to replica to support seamless failover and migration. We build a high performance IP-based protocol for low overhead traffic replication between primary and replica. HA/TCP introduces *replicated sockets*, a compatible socket API that hides the complexity of replication and eliminates the non-determinism between primary and replica. Replicated sockets simplify reliable NF development because we observed that most NF state derives from the TCP flow.

HA/TCP is highly optimized for modern high speed networks. Our benchmarks show that HA/TCP can saturate a 100 Gbps link while providing reliability. We made many critical optimizations, including eliminating all deep and shallow

copies of packets inside the critical path, relying on TCP itself to correct network errors between the primary and replica, supporting receive side scaling (RSS) and process concurrency, and supporting TCP options important to NF performance. Together these optimizations improve the performance of HA/TCP by more than 10×.

HA/TCP is built on a production grade TCP stack based on FreeBSD 13.1 and runs under DPDK using F-Stack. Our system consists of a TCP/IP stack extension to replicate the connection state, the replicated sockets API that eliminates non-determinism in sockets, and the IP Clustering system negotiates with network switches for load balancing. HA/TCP uses the common address redundancy protocol (CARP [9]) for failure detection and leader election. We implement many optimizations for high throughput and low latency. To evaluate HA/TCP, we built three layer 7 NFs including a WAN accelerator, a SOCKS proxy and a distributed load balancer. Each NF supports HA/TCP in less than 100 lines of code.

Our system makes the following contributions:

- HA/TCP is the first system to provide reliability and scalability for TCP-based layer 7 NFs. We introduce the *replicated sockets* interface that simplifies the development of reliable NFs.
- HA/TCP migrates TCP connections 2.4× faster than Prism [24] and 1.7× faster than Capybara [12] when including network latency.
- The implementation of several systems in addition to HA/TCP including: the first open-source WAN accelerator, our distributed load balancer, and our novel IP clustering system that works with commodity switches.
- Our experience building and optimizing HA/TCP for 100 Gbps networks.

Our evaluation results show that HA/TCP saturates 100 Gbps with just four connections. HA/TCP adds on average 11 µs to the round-trip time and reduces client throughput by as little as 0.2%. Our NF benchmark shows HA/TCP migrates transparently in 38 µs, which is faster than the state-of-the-art TCP migration systems. HA/TCP scales linearly across multiple nodes. The load-balancing benchmark shows HA/TCP evenly distributes load among nodes.

2 Background

The reliability and scalability challenges of layer 7 NFs are unique because their complexity far exceeds other types of NFs. Layer 7 NFs contain a complete TCP stack that terminates connections. A few recent works, Microboxes [36] and mOS [29], introduce frameworks that provide a custom TCP stack with better APIs for developing layer 7 NFs. Neither system provides reliability or multi-node scalability. Our work is complementary and can be used with either system.

Types of Layer 7 NFs. For our purposes, we classify layer 7 NFs into two categories: *pass-through* and *endpointing* NFs. Pass-through NFs, such as an Intrusion Detection System (IDS), are transparent to network connections as they only monitor traffic. They only read the packets that flow through connections and reconstruct the TCP state of each connection locally, but do not alter the packet contents.

Endpointing NFs, such as a TCP splicer, SOCKS proxy and WAN accelerator, terminate the TCP connection. The NF splices the traffic from one connection to another while optionally transforming the payload.

Endpointing NFs contain a complete TCP stack and the NF application. The TCP stack provides full TCP packet processing for the NF application. The TCP connections on either end of the NF terminate at the NF. The NF application works directly on the data that it reads from the TCP stack and sends over another connection.

The failure of pass-through NFs does not affect the connections that they are monitoring. This is because they tap each connection and reconstruct a copy of the minimal TCP state for monitoring purposes. Pass-through NFs use fail-to-wire to allow traffic to continue to flow after a failure.

However, if an endpointing NF fails, all connections through the NF break, because endpointing NFs terminate all TCP connections. A TCP connection is stateful, and the NF relies on having the correct state of both the TCP stack and the NF application. The connections to nodes on either end of the NF cannot be re-established without the NF state, which is lost during a failure.

The Need of Seamless Failover. Prior works have cited that vendors are reluctant to reset network connections because of user visible disruptions [45]. FTMB [45] shows that a number of popular applications come with defaults that cause user visible disruptions and suffer poor recovery times in the range of minutes. A recent Microsoft study stated that NF failures accounted for 42% of severe incidents, and that some failures lasted over a day [41].

NFs such as WAN accelerators are often deployed to efficiently make use of a company's network infrastructure to connect all of their remote sites together. Often they are deployed using a hub and spoke model [13, 43] which means a few NFs in the primary data centers can disrupt traffic nationally or globally for these companies.

There is no equivalent to fail-to-wire for endpointing NFs, because these NFs transform traffic. For example, a WAN accelerator may compress, encrypt and encapsulate a TCP flow over UDP to traverse the internet. Sending the traffic without traversing the NF may leave the destination unable to process the incoming connection that is incompatible, or the intermediate network is unable to forward the traffic.

Existing Replication Frameworks. Existing NF state management systems [20, 21, 31, 32, 42, 47] address the major problems of scalability and reliability for layer 2–3 NFs. These systems either replicate the state among NF instances or store the state on another node. These NFs can scale elastically by migrating flows and seamlessly handling failures.

Existing layer 2–3 NF frameworks show that their state is relatively simple, consisting of a few state variables [21, 31, 47]. These frameworks use this fact to efficiently replicate state updates by coalescing or piggybacking state updates into a smaller set of packets.

The state in layer 2–3 NFs does not update frequently. StatelessNF [31] summarized the states and the access patterns for common layer 2–3 NFs. For most NFs, such as load balancers, firewalls and NATs, the states are updated at the beginning or the ending of the connection, and the state is only read during steady state processing. Other NFs may read and write the state simultaneously, but at most once per packet.

The Problem with State Replication. Existing state replication systems are ill suited for layer 7 NFs for two reasons. First, the TCP NFs have a large complex state. TCP NFs terminate the TCP connection which necessitates replicating a non-trivial state. These NFs terminate both endpoints of the TCP connection to rewrite the TCP traffic. Both the NF application state and TCP stack state are part of the NF state that must be made reliable. Furthermore, all packets inside the NF application are thus part of the state. The resulting state update of a single packet is larger than the packet itself.

Second, the states are updated multiple times throughout the NF pipeline, and attempting to batch state updates leads to correctness or performance issues. All packet data, including the payloads, is part of the NF state. When sending the data in TCP, the TCP stack holds the data in the socket buffer until it receives the acknowledgment from the peer. However, the acknowledgment packet only indicates that the peer TCP stack has processed the packet, but the application may not have started processing. If the peer, say an endpoint NF, fails before the NF application finishes the processing, even if the NF gets failed over, the sender will never retransmit the previously sent packet.

Delaying the TCP acknowledgment until the NF application completes processing and updates its state is problematic. While simplifying replication, it introduces a performance problem for TCP, because the acknowledgment will be delayed. The delay increases the perceived round-trip time (RTT), which increases the TCP window size and the memory requirements of all nodes. More importantly, any variability in the RTT reduces the TCP window depending on the congestion control algorithm. The result is that the end nodes perceive worse network conditions, which hurts performance.

Prior TCP Failover Systems. Unfortunately, prior approaches to TCP failover [2, 49, 50] are not transparent, do not support common TCP extensions, and suffer from high overheads. These systems change the TCP protocol and require nodes to participate in migration or failover events. Several systems use another machine (i.e., a NF) to store traffic to ensure the replica node can resume communications after a failure [4, 16]. Their NF is a single point of failure that must be made reliable, yet they still incur overheads of 30% or more. These systems are unmaintained for over 20 years.

Our Approach. Our approach is to use active replication inside the network stack. We intercept incoming packets just before the TCP stack, transmit them to the replica, and resume TCP processing once the replica has acknowledged reception. Naïvely implementing this approach results in low performance. One of the main challenges of HA/TCP is how to achieve good performance.

One benefit of our approach is that we simplify the development requirement for providing high-availability for NFs by introducing *replicated sockets*. Rather than replicate the NF application state, we maintain output determinism through our replicated socket interface, which means the output from the network stack is identical on all nodes. NF applications need to ensure that they do not generate any externally visible non-determinism. This is similar to the programming discipline that game engines [11] use.

Providing output determinism does not preclude applications internally diverging, for example, the WAN accelerator may take a cache hit on the primary and a miss on the replica because of timing differences. NFs with unavoidable internal non-determinism must ensure that the output remains identical. In this example, the replica that misses, would fetch the value from its peer, resulting in identical output. Alternatively, one could use record/replay [38, 45] to synchronize state.

3 Design

TCP NFs using HA/TCP are deployed similarly to other NFs. In an NFV environment, we use an SDN controller (e.g., ONOS [7]) to manage the deployment of NF instances and coordinate traffic steering. The SDN controller coordinates with the network function's controller, which uses our novel IP Clustering system to manage the network function and traffic routing. Our main contribution is a TCP stack extension that supports the migration and failover of individual flows between NFs.

HA/TCP has three main goals:

1. **Transparent to Remote Nodes:** We cannot make protocol changes that require modifying existing TCP stacks in client nodes.

2. **Low Overhead:** While network reliability is important, users are not willing to sacrifice performance. The end-to-end impact on throughput and latency must be low to make HA/TCP practical.

3. **Fast Failover and Migration:** Failover and migrations must complete quickly to avoid user visible disruptions.

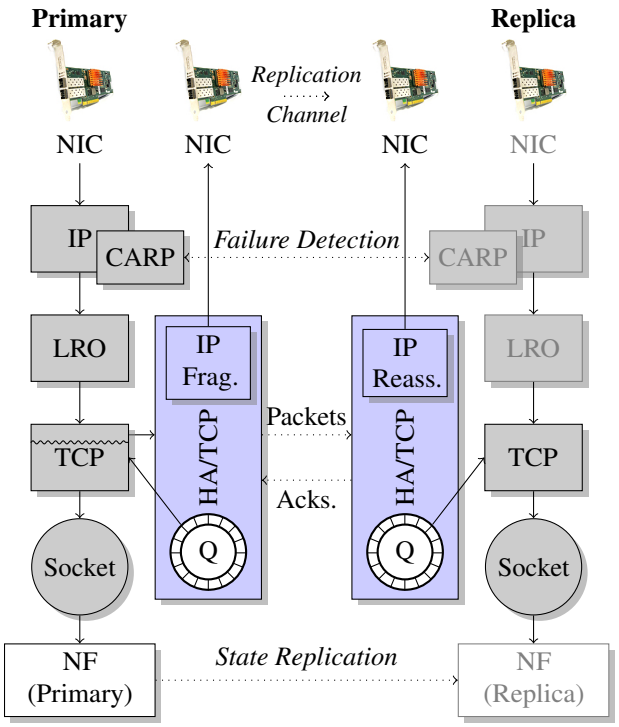


Figure 1: HA/TCP system architecture.

NFs using the HA/TCP framework, support migration and failover using our *replicated socket* interface. Our replicated socket provides a socket object that behaves identically on both primary and replica. In the rest of the design, we explain the operation of our replicated sockets, which we use to create reliable NFs. The most difficult challenge is to replicate the TCP stack to ensure that it behaves identically on both the primary and replica. In § 3.8 we discuss our extensions to CARP and our IP clustering protocol that provides additional functionality to the NFs.

3.1 Overview

Figure 1 shows the HA/TCP system architecture. HA/TCP intercepts and queues incoming packets from the TCP stack after *large receive offload* (LRO) and checksum validation. HA/TCP sends the TCP traffic and non-deterministic state (e.g., congestion control variables) over the replication channel. Both primary and replica have copies of the same incoming packet in their respective HA/TCP queue. After the replica acknowledges receiving the packet, both sides dequeue the packet and begin TCP processing. If the primary fails, CARP notifies HA/TCP to fail over the connection.

To achieve the full duplex throughput of our primary network interface, we dedicate a network interface for the replication channel between nodes. We use dual ported network interface cards (NICs), which are marginally more expensive than a single ported NICs. Our design uses active replication to a single replica. When a node fails, a new replica is deployed through NFV. Alternatively, HA/TCP supports multiple active replicas, see § 3.8.

Our performance goal is to support 100 Gbps network cards which requires examining many performance issues throughout the network stack. We can classify our optimizations into four categories: First, reducing overhead by processing larger packets. Second, eliminating memory allocations and copies throughout our code. Third, hiding replication latency. Fourth, supporting RSS for multithreaded application scalability.

HA/TCP creates listening and connected sockets that behave identically on the primary and replica, including all TCP states. The NF only needs to ensure that it identically transforms the data on the primary and replica.

Applications use `setsockopt` and `getsockopt` to enable and configure HA/TCP. A typical workflow is to enable HA/TCP on a listening socket, and any accepted connections will inherit the configuration. The listening socket may accept spoofed connections where the NF uses `getsockname` and `getpeername` to retrieve the endpoint addresses.

When the primary fails, the connection seamlessly fails over to the replica. The replica takes over the client connection and becomes the new primary. The new primary exchanges the traffic directly over the local NIC rather than the replication channel. In addition, HA/TCP provides the ability to migrate sockets to other nodes as needed.

3.2 Replicated Sockets

HA/TCP's replicated socket interface extends network sockets to provide a socket object that behaves identically on the primary and replica. Both listening and connected sockets are replicated between nodes. A common workflow for our NFs is to enable HA/TCP on a listening socket, which each accepted socket inherits.

Connected Sockets: After enabling HA/TCP, the primary continues to process traffic until a replica joins. Once a replica joins, the primary temporarily pauses the traffic and synchronizes all TCP state with the replica. Once complete, the primary resumes the client traffic.

HA/TCP Replication Channel: HA/TCP exchanges state and packet data over the replication channel, which is an IP protocol that encapsulates these packets with a small header. Our system implements and registers an IP protocol with the network stack that allows us to process packets in-line. For example, on the replica we decapsulate packets and forward them to the usual TCP input processing path as if they were received from the client directly.

The IP protocol is simpler and more efficient than using layer 4 protocols such as UDP or TCP for replication. The IP layer does not look up control blocks, update the internal state, acquire any locks nor apply congestion control, which lowers overhead and simplifies our design. Furthermore, modern NICs support IP checksum offloading.

The IP layer offers several other benefits over layer 2 protocols such as Ethernet. The IP layer provides IP fragmentation and reassembly to exchange large packets in MTU sized chunks. IP protocols are routable and can traverse many network devices including over metropolitan area networks.

As a performance optimization (see § 3.5), we do not handle packet loss in our replication channel. Instead, we rely on the clients to time out and retransmit to the primary and replica. Without an acknowledgment from the replica, the primary will not acknowledge the TCP packet and thus the client will retransmit the packets.

In an earlier design, we used a TCP connection for the replication channel, which led to performance issues from high CPU usage and the TCP meltdown problem [27]. This doubles most of the network stack overhead as every incoming packet traverses the primary's NIC driver through the TCP stack and out the replication interface's NIC driver to the replica. Similarly, the replica processes TCP twice before forwarding data to the application.

TCP meltdown is a problem that occurs when running TCP over TCP. It arises because HA/TCP holds packets on the primary until the replica acknowledges reception. If a packet is held longer than the client's retransmission timeout, the client treats this as packet loss and retransmits the packet. For example, the replication channel may delay delivery because of congestion control, which triggers a cascade of retransmissions by the client that exacerbates the issue.

Avoiding IP Reassembly Collisions: The challenge of using IP for the replication channel is that IP fragmentation suffers from reassembly collisions [26]. The IP layer breaks up larger packets into MTU sized chunks. At high throughput, packets may be incorrectly reassembled because of collisions with the fragment identifier. HA/TCP discards incorrectly assembled packets, which results in the client observing packet loss. The packet loss causes client retransmissions, which reduces application performance.

IP reassembly uses the 16-bit *IP identification* field and the 13-bit *IP offset* field to identify fragments belonging to the same packet and reassemble them into the original packet. Packets that are retransmitted because of loss or timeouts pollute the IP reassembly queue with old fragments. Usually, this is handled by clearing the IP reassembly buffer every 30 seconds. At high throughput, this is insufficient and we see periodic collisions.

To solve this we added an IP option containing a 32-bit unique ID and 32-bit timestamp to eliminate collisions. Each replicated TCP connection is assigned its own unique ID to avoid collisions between different connections.

Listening Sockets: HA/TCP replicates listening sockets to the replica. Upon establishing the replication channel on the replica, all states in the socket object and TCP stack are synchronized. The replica then copies and applies all states to the corresponding object.

HA/TCP ensures that new connections are created identically on both primary and replica. When a new connection arrives, the primary forwards the 3-way handshake packets to the replica.

First, the SYN packet, the primary's initial sequence number, and a time offset for TCP timestamps are forwarded to the replica. This packet is processed normally by the TCP input path except that it uses the initial sequence number from the primary.

Second, after receiving an acknowledgment from the replica, the primary generates and sends a SYN-ACK packet back to the client.

Third, the primary replicates the ACK message from the client and the initial window size. We replicate the initial congestion window size in case TCP fast open or the host cache provides initial values. The replica processes the TCP ACK normally.

HA/TCP inherits the replication configuration for newly accepted sockets. Before returning from `accept`, HA/TCP initializes and activates replication on the newly created socket, before the application uses the socket.

3.3 Steady State Processing on Primary

In normal operation, packet processing on the primary undergoes five steps.

First, HA/TCP intercepts incoming packets inside of the TCP input path. After the TCP input performs the checksum validation and looks up the TCP control block, which filters out any corrupt packets or packets not belonging to established connections, HA/TCP intercepts the packet.

Second, HA/TCP places the original packets in a queue. The packet mbufs and the TCP control block are stored to allow the resumption of the TCP processing without repeating the work completed in the first step.

Some time-based state is modified by buffering. The round trip time increases by the additional latency of waiting for the replica's acknowledgment. For some congestion control algorithms, we need the replica to respond with near constant latency otherwise the algorithm may believe there is congestion in the network. To avoid accidental duplicate packet detections, we deliver packets in order so the packet timestamps [8] are monotonic (see § 3.7).

Third, HA/TCP duplicates and transmits the packet to each replica. The duplication is necessary because when the packet is sent through the replication channel, the network stack frees the packet mbufs. As an optimization, we merge packet duplication with IP fragmentation to reduce overhead (see § 3.5). HA/TCP prepends a small header containing the packet size and, critically, any congestion control updates including window size changes (see § 3.4). The packets are then transmitted over the replication channel to the replica nodes.

Fourth, HA/TCP waits for an acknowledgment from the replica. While waiting for the replica to reply, HA/TCP continues to process incoming packets from the client.

Fifth, HA/TCP releases the queued packet to the TCP stack once the replica acknowledges the packet. The packet resumes processing in the TCP stack, where it left off in the first step.

3.4 Steady State Processing on Replica

The replica socket replicates the state and traffic from the primary. HA/TCP simulates the interface and TCP behaviors until a failover or migration event occurs. Critically, for good performance, the replica acknowledges packets back to the primary in constant time to avoid triggering client retransmissions and congestion control. In normal operation, packet processing on the replica also undergoes five steps.

First, HA/TCP receives a packet from the replication channel and parses the HA/TCP header. If the packet contains a replicated packet, HA/TCP removes the header and acknowledges back to the primary. All processing in this step occurs on the NIC thread associated with the replication channel. By acknowledging immediately, we eliminate any timing jitter from queuing or TCP processing. Otherwise, TCP congestion control may reduce the window because of timing variance.

After the replica acknowledges the packet, the primary may send a TCP acknowledgment to the client. The replica is required to keep all packets it has acknowledged regardless of whether the TCP stack is ready to process them. HA/TCP uses a queue to store any packets that are not ready to be processed by the TCP stack. For example, the client may acknowledge a packet from the primary that the replica has yet to transmit. In this case, the acknowledgment number is in the future and the TCP stack would discard it.

Second, HA/TCP enqueues the packet until it is ready to be processed by the replica TCP stack. The queue masks performance differences between primary and replica. When a replicated packet arrives, HA/TCP checks if the acknowledgment number is within the maximum sequence number sent, and only delivers the packet when this condition is met.

The queue length trades off tolerance for the performance differences between nodes and the failover time. Large queues better mask performance variability and tail latency in the replica application. However, the large queue increases the switchover time during the failover process because the queue must be drained before the replica assumes the role of primary. The queue size can be tuned for a specific workload and we limit the queue size to the receive socket buffer size.

HA/TCP monitors the queuing level on the replica and throttles the incoming traffic if the replica queue is saturated. The replica acknowledgment contains the queue size that the primary uses to check whether the replica is keeping up. When the replica falls behind and the queue fills up, which should not happen in a properly configured environment, the primary throttles to avoid overflowing the replica queue.

Third, HA/TCP checks the packet header to determine if any time-based state must be synchronized. The RTT estimates and window sizes are synchronized between nodes as needed to prevent inadvertently rejecting packets.

Fourth, HA/TCP iterates through all packets queued for delivery. If the condition mentioned in the second step is met, HA/TCP delivers the packet to the TCP stack; otherwise it moves to the next packet in the queue. For example, if packet loss occurs between the primary and client, the packets at the front of the queue will not match the TCP state. Without iterating through the queue, the retransmitted packets at the tail will block further processing.

Fifth, for outgoing traffic, HA/TCP inspects and discards the packets after all TCP processing is complete. Although HA/TCP frees the packet, the content of this packet resides in the socket buffer until the replica receives the TCP acknowledgment from the client. The packet can be retransmitted from the socket buffer. HA/TCP also attempts to deliver packets from the queue, after the TCP output code path transmits a packet. This delivers any packets that did not previously meet the condition (from the second step).

3.5 Performance Optimizations

HA/TCP is optimized for high performance network interfaces. In this section, we discuss each optimization and show the performance breakdowns. Table 1 shows the performance as we enable optimizations one at a time.

HA/TCP has three major data processing paths.

1. INPUT: The input path includes enqueuing the packets and transmitting them through the replication channel.

2. DELIVERY: Receives the replica acknowledgments and releases packets to the TCP stack.

3. REPLICA: The input path on the replica.

In HA/TCP, the INPUT path contains most of the high overhead operations. The DELIVERY path and REPLICA paths are similar to the baseline TCP network path. The total system overhead from HA/TCP consists of all three paths from all nodes. The overhead of each path is shown to make the effect of the optimization clear.

Recall from § 3.2, our early design uses TCP as the transport for our replication channel. Columns 2–4 in Table 1 use the TCP-based replication channel, while the remaining columns use the IP-based replication channel.

Use TCP LRO to Reduce Overhead. HA/TCP is optimized for the standard MTU of 1500 bytes to support user facing services in the cloud or data centers. In our experience, the FreeBSD TCP stack consumes CPU proportional to the packets per second (PPS). Thus, reducing the number of packets reduces the processing cost. LRO merges multiple TCP packets into a single packet earlier in the network stack to eliminate the cost of processing many TCP headers. For large flows, LRO can combine packets into 64 KiB packets.

		Unreplicated	No Opt.	w/LRO	w/LRO+NoCopy	IP+LRO	w/NoCopy+ReAsm
	Throughput	93.60 Gbps	8.60 Gbps	53.39 Gbps	73.13 Gbps	54.23 Gbps	90.50 Gbps
	Pkt/s	7.80 Mpps	0.72 Mpps	4.45 Mpps	6.09 Mpps	4.52 Mpps	7.54 Mpps
Primary	CPU INPUT	79 %	94 %	91 %	94 %	90 %	89 %
Primary	CPU DELIVERY	-	53 %	66 %	63 %	28 %	28 %
Primary	CPU APP	61 %	35 %	52 %	59 %	54 %	71 %
Repl.	CPU REPLICA	-	67 %	39 %	52 %	43 %	71 %
Repl.	CPU APP	-	47 %	42 %	56 %	38 %	60 %

Table 1: Shows HA/TCP with optimizations turned on/off. The first column shows the performance without replication. Columns two through four uses our TCP-based replication channel, and the last two columns uses our IP-based replication channel.

Without any optimization, the INPUT path is the performance bottleneck with two sources of overhead.[1] First, the INPUT path spends 9.7% of CPU time processing the incoming packets. This is roughly 0.72 M PPS with the default MTU of 1500 bytes. Without optimizations, CPU overhead scales linearly with the packets per second.

Second, the INPUT path consumes 5.3% of CPU time duplicating network packets using `m_dup`. The CPU time scales with the number of packets and the size of the payload, because `m_dup` deep copies the packet headers and data.

Enabling LRO chains consecutive packets into larger packets of up to 64 KiB in size through mbuf chaining, i.e., without deep copies. LRO trades off the processing overhead of identifying consecutive packets in a stream with the per-packet processing costs throughout the TCP stack. Furthermore, LRO piggybacks on interrupt coalescing to chain packets that are delivered in the same interrupt without increasing latency. Potentially LRO can use hardware offloads, but in the NICs we tested this performed worse.

The average packet traversing the network stack with LRO is 61400 bytes in size, resulting in a 40× reduction in effective PPS. While HA/TCP saves the per-packet CPU processing and packet duplication overheads with LRO, LRO itself adds overhead. However, the performance gain more than offsets the cost of LRO. LRO improves HA/TCP throughput by 6.2×. In our measurements, LRO slightly decreased latency because of the substantial increase in throughput.

Avoid Deep Copies of mbufs on the Primary. Our replication protocol prepends its headers to the packet for transmission. HA/TCP duplicates every incoming packet to retain an unmodified copy for the local TCP stack.

Even after enabling LRO, the INPUT path remains the performance bottleneck. The deep copying of mbufs using `m_dup` consumes 12% of CPU. To address this, we use `m_copypacket` to perform a shallow copy. The function duplicates mbuf headers and reference counts the payload. Shallow copying reduces memory copies and cuts the number of allocation/deallocations in half. Switching to `m_copypacket` improves throughput and reduces CPU usage.

Use IP-based Protocol for the Replication Channel. In § 3.2, we mentioned HA/TCP uses an IP-based protocol for the replication channel. Compared to a TCP-based transport, the IP protocol is simpler and more efficient.

With the optimizations we mentioned above, we use our original TCP-based replication channel. We found that the INPUT path consumes 8% of the CPU time on TCP encapsulation and processing. Our TCP-based replication channel reduces the client throughput by 28% to 73 Gbps as compared to an unreplicated setup. Furthermore, TCP over TCP suffers from TCP meltdown problem (see § 3.2).

Next, we implement an IP-based replication protocol. We compare the IP-based and TCP-based protocols with LRO enabled. This rules out effects from other dependent optimizations (see next section).

The IP-based protocol simplifies the locking design. Sending and receiving traffic in the TCP-based replication channel goes through the TCP stack, which locks the TCP control block. In addition, reading or writing the socket buffer for the replication channel requires an additional lock. By switching to an IP-based protocol, we eliminate these locks; moreover, this simplifies the design and avoids lock ranking problems that may lead to deadlocks.

Both versions are bottlenecked on the INPUT path because of the packet duplication function. With the IP-based protocol, the throughput only increases marginally. Turning on the shallow copying optimization significantly improves the performance but it requires the next optimization.

Create Replica Copies with IP Fragmentation. Our IP-based replication protocol depends on IP fragmentation to split large packets into smaller MTU sized packets, which replaces TSO (TCP segmentation offloading) in our TCP-based protocol. IP fragmentation is required as LRO merges small packets up to 64 KiB in size and this size far exceeds the jumbo MTU size of the replication channel.

Our next major source of overhead arises from the deep packet copies caused by IP fragmentation. When transferring

[1] Adaptive polling is disabled in our configuration. The CPU spends 5–10% on CPU scheduling and waiting for interrupts. We confirmed by monitoring the interrupt rate and using CPU profiling.

the replicated packet through the replication channel, IP fragmentation breaks the packet to fit the MTU of the replication channel. The IP fragmentation process calls the packet duplication function, which allocates and copies the mbuf. The MTU between the primary and the client is by default 1500 bytes, while the replication channel uses a 9k MTU. LRO merges small packets into a 64 KiB packet, and subsequently these large packets are broken into 9k jumbo packets.

To optimize IP fragmentation, HA/TCP unifies packet duplication and IP fragmentation. The unified IP fragmentation path only copies the packet once to simultaneously fragment the large packet into the replication channel and retain a reference to the original mbuf chain for retries and delivery to the TCP stack. Merging these two functions with shallow copying reduces the overhead. Combining these functions improves client throughput by 67% over the LRO optimization alone.

3.6 Additional Optimizations

We also added a few optimizations that we do not include in our performance breakdown. These optimizations reduce CPU usage on non-bottlenecked paths, avoid unpredictable behavior or support performance features of TCP/IP.

Allow Client to Repair Packet Loss. We expect packet loss to be more common between the client and the primary than over the replication channel. We rely on the client to retransmit any lost packets. Recall that the primary only processes the packet after receiving the acknowledgment from the replica. Any packet loss on the client-primary connection or replication channel causes the primary to not acknowledge the packet, resulting in a client retransmission. Packet loss in a data center, or even across a metro area network, is rare when compared to the wide area network, and thus having the client retransmit packets negligibly impacts performance.

Eliminate Delayed Processing on the Replica. We disable Naggle's algorithm and TCP delayed acknowledgments on the replica to reduce buffering. Both optimizations delay and batch network packets to reduce the overhead in the network stack. However, the delay results in more packet buffering and slows down the TCP state updates. The falling behind TCP state on replica leads to more buffering in the queue, and thus affects the performance of failover and migration.

Support Receive Side Scaling. RSS is a NIC feature that load balances traffic across CPU cores for performance. Connections are routed to a specific core based on a hash of the 5-tuple. Unfortunately, IP protocols are hashed based on their 3-tuple of IPs and protocol numbers resulting in a miss match between flows and application threads. We use Intel Flow Director [15] or Mellanox Flow Steering to maintain a one-to-one match between cores on the primary and replica. We use a range of IP protocol numbers so that the flow rules can forward the replication channel traffic to the correct core.

3.7 Challenges with TCP options

HA/TCP supports modern TCP options. Failing to support these features leads to performance or correctness issues.

Concurrent Listening Ports (`SO_REUSEPORT`). Multi-process or multi-threaded applications often create multiple listening sockets on the same port with `SO_REUSEPORT(_LB)`. To ensure deterministic behavior we establish a one-to-one mapping of listening sockets on the primary and replica.

SYN Cache. HA/TCP handles randomness from the TCP handshake. When the primary receives the SYN packet, the SYN cache generates random states for security purposes. These random states are sent to the replica along with the SYN packet. After the replica acknowledges the replication of this SYN packet, the primary delivers the packet again to the SYN cache for the client to respond. However, the primary needs to reapply the same random states to make sure the SYN cache generates the identical states.

Syncookie and TCP Fast Open (TFO). The syncookie and TFO use a cryptographic secret and a MAC function to allow the client to store cached values from the server. To ensure both TCP options function correctly, we synchronize these two secrets between the primary and replica.

SACK on the Replica. HA/TCP guarantees the state of SACK is identical on the primary and replica after a failover or migration. The replica delivers queued packets only when the acknowledgment number refers to packets that have already been "sent" by the replica (§ 3.4). During the migration, HA/TCP drains all remaining packets into the TCP stack, and missing packets will result in the same SACK state.

Synchronizing Clocks for PAWS. TCP PAWS [8] uses TCP timestamps to detect retransmitted packets and discards the packet. To support PAWS, HA/TCP tracks the time difference between primary and replica (`ts_offset`) and delivers the packet with an updated TCP timestamp adjusted by the time difference. The PAWS check is completed on the primary and for valid packets, the timestamp offset will ensure that the PAWS check will pass on the replica. We periodically synchronize the clocks to avoid drift.

Congestion Control Algorithms. We tested HA/TCP with both loss-based and RTT-based congestion control algorithms and found no significant performance differences. For loss-based algorithms, such as NewReno [22], DCTCP [6], and H-TCP [35], HA/TCP does not alter the retransmission mechanism in TCP, nor the congestion detection that the algorithms rely on. For RTT-based congestion control algorithms, such as HD [10] and CHD [25] that monitor network latency and latency variance to detect congestion, both of which are relatively stable. HA/TCP, and in particular the IP replication channel, is optimized to reduce the processing overhead, which results in a low and constant increase in latency (§ 5.4).

3.8 Load Balancing, Failover and Migration

HA/TCP uses the Common Address Redundancy Protocol (CARP) [9] protocol (based on VRRP [39]) to detect node failures and perform leader election, and our novel IP Clustering system to share a single IP address across nodes for scalability. We also discuss migration and having multiple replicas at the end of this section.

Failover (CARP). Each CARP node sends a periodic multicast advertisement. If three consecutive advertisements are not received from the primary, CARP declares the primary as failed. CARP then promotes a replica to be the new primary, and transmits an ARP announcement to notify the switch of the migration of the IP and MAC address to the new port.

In this paper, we set the advertisement interval to 100 ms, which results in a 300 ms detection and recovery time that is less than most application and TCP timeouts. Identifying the proper trade-off between detection time and false positives is beyond the scope of this paper.

HA/TCP fails over with minimal disruption. During steady state, HA/TCP replicates the incoming traffic to all replicas. Although replicas may lag behind the primary, our measurements show that the delay is only in the tens of milliseconds. The CARP failure detection time masks this lag and allows the replica to drain the queue before the failover.

IP Clustering. HA/TCP's IP clustering allows multiple nodes to share the same IP address for scalability. Our IP clustering system uses the link aggregation control protocol (LACP) to negotiate load balancing with the switch [1]. LACP allows one node to have multiple network interfaces to a switch, but we developed a distributed version of LACP that allows multiple nodes to share the same IP and MAC address.

Our distributed LACP protocol communicates between nodes in a cluster to load balance traffic between the switch and the nodes. The challenge is that HA/TCP's distributed LACP needs to synchronize the LACP state and ARP state between nodes continually. All the nodes must present themselves to the switch as a single node with several state fields being synchronized. ARP messages from the switch are only routed to a single node, leaving other nodes unaware of the ARP traffic. We use the replication interface to synchronize the LACP state and the ARP entries among all nodes.

Currently, IP clustering is supported without replication, because it requires additional application orchestration to manage the placement of per-connection primary-replica pairs. With complete orchestration, our IP clustering system will subsume CARP's functionality.

Migration. Migrations are driven by the primary, which stops processing the client traffic, sends a migrate command to the new primary, and demotes itself to a replica. The new primary sends the promotion message to the replica, i.e., the previous primary, to confirm the migration completion. Once complete, the replica updates its state and the new primary begins processing traffic.

Implementation	Count
HA/TCP Kernel Code	10K SLOC
HA/TCP IP Clustering	1.4K SLOC
SOCKS Proxy	3.3K SLOC
WAN Accelerator	8.7K SLOC
Distributed Load-balancer	1.2K SLOC
Latency Prober	1.9K SLOC
Multi-protocol Load Generator	1.9K SLOC

Table 2: Approximate source lines of code (SLOC) for HA/TCP and our NF applications.

Multiple Replicas. HA/TCP supports multiple replicas. Given the resource requirements of active replication between the primary and replica, it is recommended to use NFV to deploy a new replica if a node goes down. This eliminates the resource consumption of having another full replica, and the extra demands on the primary to keep them synchronized.

HA/TCP supports multiple active replicas with unicast connections for the replication channel. The primary requires additional dedicated network interfaces if it wants to serve the full available bandwidth of its primary interface. Alternatively, multicast can be used to eliminate the need for additional network interfaces with low latency and low packet loss, but it has not been implemented.

4 Implementation

HA/TCP is implemented in the FreeBSD 13.1 network stack. This is a popular operating system for network appliances and the basis of F-Stack, a TCP/IP stack for DPDK. Table 2 shows the breakdown of lines of code for HA/TCP and the NFs. The HA/TCP implementation consists of 8.8K source lines of code (SLOC) that extend the TCP stack, 100 SLOC in CARP, and 1.6K SLOC in the socket API. The IP Clustering system consists of 1.4K SLOC. The SOCKS proxy, WAN accelerator and load balancer consist of 13.2K SLOC. In addition, the SOCKS proxy and multiprotocol client conform to RFC1928 [34], RFC7230 [17] and RFC7231 [18].

5 Evaluation

We evaluate HA/TCP with micro-benchmarks and three TCP NF applications. The micro-benchmarks measure the scalability, migration/failover time, replication overhead, and latency of HA/TCP. For the three TCP NF applications we show the throughput and CPU usage.

In all benchmarks, the throughput is measured from the client application. HA/TCP replicates and forwards the traffic to the replica, and thus generates traffic in the replication channel equal to the client traffic. The replication channel uses

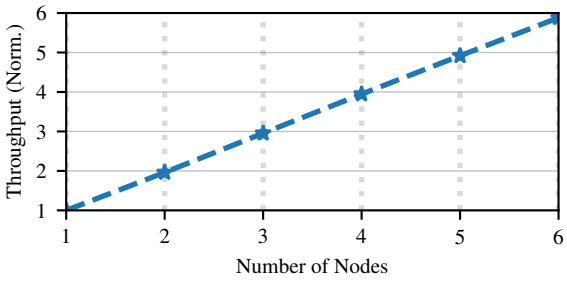


Figure 2: HA/TCP scales linearly with number of nodes while sharing a single IP and MAC address. HA/TCP load balances connections through our IP Clustering system. The throughput with 6 nodes is 2% lower than ideal.

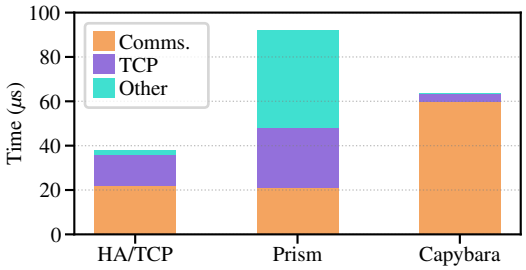


Figure 3: Migration time breakdown including network latency. HA/TCP takes 16 µs to complete the migration process, while the network latency adds the remaining 22 µs. HA/TCP is faster than the state-of-the-art systems.

a dedicated network interface to support the full bandwidth of the primary interface for client traffic.

We measure the CPU usage in the INPUT, DELIVERY and REPLICA paths (see § 3.5), along with the application CPU usage including both user and kernel time. We present a breakdown of the CPU usage for the baseline, primary and replica. The total CPU overhead is the percentage increase in CPU from both the primary and replica.

To minimize variance in our measurements, we pinned each context to a separate CPU and presented the total CPU usage. We used multiple tools to monitor system performance and hardware performance counters to breakdown the CPU usage. This is the same process we used during development.

All measurements were done on a local area network with low latency between all clients, primary and replica nodes. This represents the worst-case, as the latency between the client and primary is the same as the latency between the primary and replica. Over wide area networks, the extra latency induced by replication is negligible when compared to WAN latencies. In all benchmarks we used a single replica.

The primary and replica ran on two dual socket Intel Xeon Gold 6342 processor machines. All servers and clients use a dual-ported 100 Gbps Mellanox ConnectX-6 Ethernet NICs connected to a 100 Gbps Mellanox switch. The MTU between client and primary is set to the standard 1500 bytes, while the MTU between nodes is 9000 bytes. The TCP stack uses the New Reno congestion control algorithm. For failover tests, we set the advertisement interval (`adv_base`) to 100 ms to allow CARP to detect failures on average in 300 ms (§ 3.8).

5.1 Scalability

Figure 2 shows the scalability of HA/TCP's IP clustering. Our IP clustering allows NFs to share a single IP address across any number of nodes. The scalability is only limited by the switch's ability to load balance traffic. In this benchmark, we had the clients send traffic to six Intel Xeon Silver 4116 nodes each with a 25 Gbps interface. We use these nodes because we have a limited number of nodes with 100 Gbps NICs.

5.2 Migration and Failover

Figure 3 shows the breakdown of the migration time for HA/TCP, Prism [24] and Capybara [12]. To provide a fair comparison, we breakdown the migration time for the TCP/IP state, and omit system specific features such as application level migration that not all systems support. For each system, we breakdown the migration into three main costs: *Comms.* that includes any synchronization between nodes, *TCP* that includes the serialization and reconfiguration costs, and *Other* that contains other system specific costs.

Migrations are initiated by the source. The migrate message is sent through the replication channel to the destination in 22 µs. The communication latency dominates the total migration time. Afterwards, HA/TCP updates the TCP control block with the new IP address in 5 µs. The control block is stored in a hash table indexed by the connection's 5-tuple. HA/TCP removes and re-inserts the control block into the hash table using the new 5-tuple value. After updating the control block, HA/TCP flushes any buffered packets in 3 µs. Finally, HA/TCP re-enables the TCP processing in 2 µs.

Prism takes 92 µs if we consider only TCP operations. Prism serializes and reconfigures TCP in 27 µs. Prism takes 21 µs to communicate once to exchange the TCP state. The *Other* in Prism includes the traffic blocking and rule rewriting, which takes 44 µs to complete.

Capybara transfers the TCP state in 3.7 µs, which is much faster than HA/TCP and Prism because it uses a library OS with a custom TCP stack. HA/TCP and Prism are based on commercially used networking stacks (FreeBSD and Linux), which are substantially more complex to reconfigure. While Capybara's state transfer is fast, it communicates over the network three times. Capybara completes the migration in 64 µs including the three round trips over the network.

The failover process is identical to the migration process except that it is triggered by CARP. The failover time depends on the CARP detection time. As mentioned in § 3.8, we configured CARP to detect failures on average in 300 ms. HA/TCP finishes the migration 13 µs after failure detection.

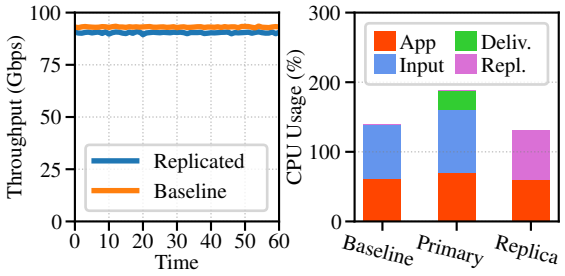


Figure 4: Replication overheads for iPerf3 with the receive-bound traffic configuration. The increase in latency results in a decrease of 3.4% in throughput.

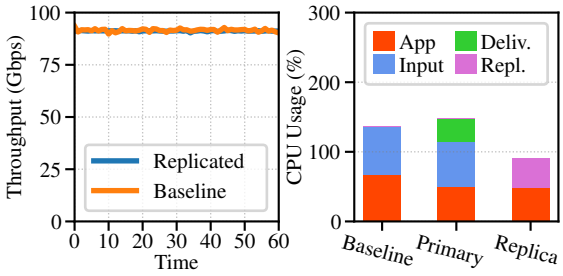


Figure 5: Replication overheads for iPerf3 with the transmit-bound traffic configuration. The throughput and CPU overheads are low because HA/TCP is mainly doing bookkeeping and replicating acknowledgments.

5.3 Replication Overhead

We ran iPerf3 in transmit-bound and receive-bound configurations to measure the overhead of the transmit and receive paths. The transmit-bound configuration has the client send data to the server, while the receive-bound configuration has the server send the data to the client.

Figures 4 and 5 show the results for the receive-bound and transmit-bound configurations. The replication channel adds 116 bytes of metadata to every packet. For iPerf3, LRO reconstructs mostly 64 KiB packets that are fragmented by our IP fragmentation code. The replication channel header should result in a 0.2% decrease in peak throughput if the workload is saturating the 100 Gbps link.

The receive-bound configuration reduces throughput by 3.4%, because of the higher input path processing costs and the additional latency of waiting for the acknowledgment from the replica, before completing TCP processing. In our measurements, we found that the input path is the bottleneck with over 90% of a single core used per connection.

The transmit-bound configuration reduces throughput by 0.3%, because the primary mostly performs bookkeeping to track which outgoing packets are ready to be sent to the client. Application CPU usage is lower than the baseline, because the longer latency taken for application acknowledgments results in more buffering in the outgoing socket buffer. The increase in latency causes the application to block more often. In response, the application increases the amount of data sent in subsequent calls, which amortizes the system call overhead.

HA/TCP uses additional memory for replication. During the steady state, both the primary and replica hold the replicated packet in the packet queue until the conditions are met. The memory overhead is small because HA/TCP reduces memory consumption through shallow copying.

In our setup, under the maximum throughput of 100 Gbps the RTT of the replication channel acknowledgments was approximately 70 µs resulting in a peak memory usage of 875 KiB. The peak memory usage on the primary is derived from the bandwidth delay product of the replication channel. We expect the replica to use a similar amount of memory.

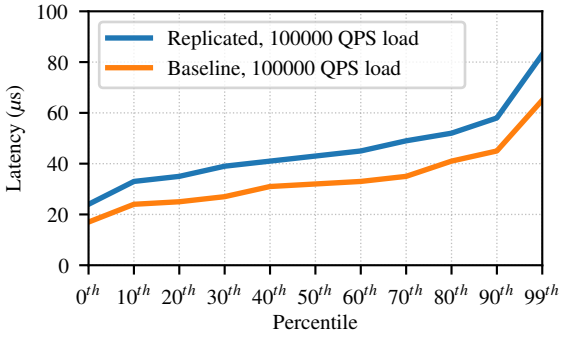


Figure 6: Latency overheads for replication. The client probes the latency by sending requests at 1000 QPS while the server is serving a 100k QPS workload. The latency increase reflects the delay in replicating packets to the replica. We achieve consistent low latency by avoiding the latency of waiting for TCP processing on the replica.

5.4 Latency Overhead

We evaluated the latency response using a distributed workload simulator for key-value stores [56]. In this benchmark, two clients connect to the server simultaneously. One client generates 100k queries per second (QPS) while another client measures the server response time by issuing 1k QPS. The results are an average of ten runs for each configuration.

Figure 6 shows the application latency with and without replication. The latency increases 11 µs on average.

The latency is lower than twice the RTT because the RTT in the replication channel is lower than the RTT between the client and the primary. We achieve this low replication channel RTT by having the replica send acknowledgments inline in the network stack, and our optimizations, including using our IP-based replication protocol, which reduce the CPU usage. Furthermore, the replica buffer hides processing delays to avoid increasing tail latency.

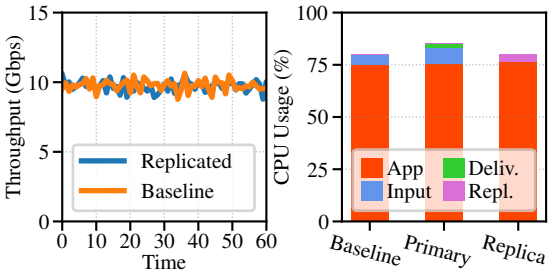


Figure 7: Replication overheads for the WAN accelerator. This workload is CPU and latency bound.

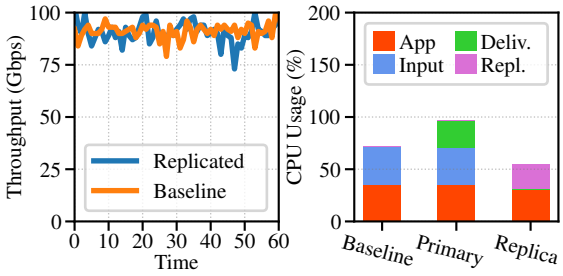


Figure 8: Replication overhead for the SOCKS proxy that shows a decrease in throughput of 2%. This workload is transmit-bound with small incoming requests and large responses. The higher variance in throughput comes from the replica's queuing mechanism delaying a substantial amount of response traffic from the server.

5.5 WAN Accelerator

The WAN accelerator is our most complex NF. WAN accelerators often serve as permanent VPNs connecting two or more data centers, but with the addition of compressing and deduplicating traffic to increase the effective bandwidth. They also aggregate traffic and send them through fewer connections between instances.

We found no open-source WAN accelerators and developed our own that will be available for others to use. WAN accelerators have very high CPU requirements because of the compression and deduplication, but commercial ones also provide protocol specific optimizations that require more CPU. Our accelerator can support 9.7 Gbps, which is faster than commercially available accelerators [48].

The WAN accelerator is configured to deduplicate and compress traffic before encapsulating and sending the traffic to the remote accelerator. This benchmark is unique because we replicate both sides of the WAN accelerator. During the test, the client creates eight long-live connections and issues requests to fetch a 10 MiB size file from the HTTP server through the accelerator. The accelerator uses eight cores for traffic deduplication and eight cores for traffic compression.

Throughput Overheads. Figure 7 shows the throughput. We apply HA/TCP to both the connection between accelerator peers and the connection between the accelerator and the server. We measured no statistically significant change in client throughput. In this benchmark, the application is not throughput bound or CPU bound, therefore we expect no change in client throughput.

CPU Overheads. The application is CPU and latency bound when passing data through the WAN accelerator's deduplication and compression pipeline on the side closest to the server. HA/TCP consumes 6.4% more CPU on the primary compared to the baseline CPU usage. Including the total replica's CPU usage, the replication consumes 106% more CPU across both the primary and the replica.

Failover. We tested failover using a multi-path setup with three appliances. There is a primary and replica on each site. This setup bridges two networks and allows for per-connection load-balancing. CARP's failure detection accounts for most of the failover time, and HA/TCP fails over all connections to the replica in 132 µs after failure detection.

5.6 SOCKS Proxy

A SOCKS proxy is an NF that forwards multiple protocols through a server to restrict Internet access from inside a secure network. It is often used to block phishing and other undesirable sites. We built an event-driven SOCKS proxy that conforms to the SOCKS5 protocol as defined in RFC1928 [34].

We used our multi-protocol load generator with 16 connections to fetch a 1 MiB file from the HTTP server through the SOCKS proxy. The long-lived connection option is enabled on the server and client. The connections between the SOCKS proxy server and clients are replicated for the benchmark.

Throughput Overheads. Figure 8 shows the client throughput with HA/TCP enabled. Replication reduces throughput by 2%. The throughput is bounded by the remote server response latency.

CPU Overheads. Figure 8 shows the CPU usage in the application and the input path does not change significantly from the baseline. HA/TCP consumes 29% CPU more on primary compared to the baseline. Including the total replica's CPU usage, the replication consumes 102% more CPU in total across both the primary and the replica.

Failover. Each buffered client request generates a 1 MiB response from the server. Even a modest amount of queuing on the replica leads to a large failover time. As part of the steady state processing on the replica, the queue is drained concurrently with failure detection. The replica is behind the primary by roughly 44 requests. The failover completes in 84 ms after failure detection.

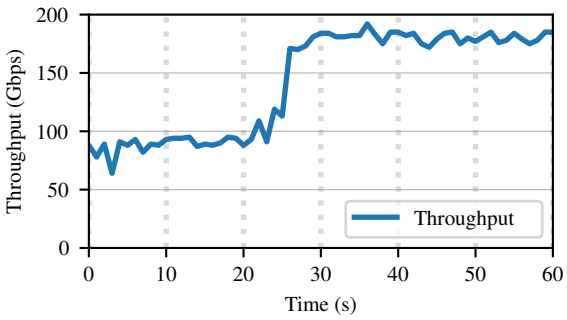


Figure 9: Connection level load-balancing. The load-balancer is deployed on two nodes. Two clients initially connect to a single server, and then load balancing is enabled to migrate half the connections to the second node.

5.7 Load Balancer

HA/TCP allows transparent connection-level load balancing among nodes. We implemented a distributed load-balancer that uses HA/TCP to balance connections for an HTTP service. The load-balancer works across multiple nodes to direct traffic to an HTTP server instance.

In this experiment, we run the load balancer on two servers and migrate half the connections while leaving the other connections alone. We used two identical servers with 100 Gbps NICs and two clients also with 100 Gbps NICs. Initially we create 64 connections to one server, and we enable the load balancing operation at around 22 s. HA/TCP then migrates 32 connections to the second server.

Figure 9 shows the aggregate throughput of all servers. One server is processing 90.6 Gbps and after the migration both are processing an aggregate of 181.2 Gbps.

6 Related Work

In addition to the works described in the background, there are other works on migration and replication of transport protocols. Most of these approaches deviate substantially from our criteria for HA/TCP.

Record-and-replay Based TCP Failover. FT-TCP [53,54] replicates TCP using record and replay. During a connection's lifetime, FT-TCP records all client network traffic and system calls on the server. When a failure occurs, the replica replays the recorded traffic and the system calls to reconstruct the primary's state. This approach suffers from high overheads and a slower failover.

Application Specific TCP Failover. Snoeren et al. [49] proposed an approach that supports TCP failover and migration. This approach uses a non-transparent state replication extension to facilitate the migration. The migration process is based on a proposed TCP migration extension [5], which requires the client to respond to a new *MigrateSYN* packet. However, this proposal has since been abandoned.

Surton et al. [51] introduced a middleware system TCPR that masks TCP failures from the border gateway protocol (BGP). TCPR implements a protocol specific *graceful recovery* technique, which avoids the cost of replaying the entire session for long lived connections, to recover the BGP session state quickly and resume communications. This proposal does not include the transparent failover and the migration features present in HA/TCP.

Other Protocols. QUIC [28, 33] is a transport protocol initially made to improve performance for HTTPS traffic. The protocol is based on UDP and allows multiple streams per connection. QUIC replaces the IP-Port tuple with a connection-ID to allow for connection migration. QUIC requires client application support. Our techniques from HA/TCP can be applied to support the failover of QUIC NFs.

Trickles [46] is a TCP-like transport protocol that is stateless on the server. The server-side protocol state and the application state are moved to the client and sent along with the user payload every time. This allows the server to migrate because all states are stored on the client. Trickles requires kernel and application changes on both servers and clients. Moreover, the state information that is included with every request, is prohibitively expensive for small packet workloads.

Zandy et al. [55] proposed a reliable network connection library. The library addresses issues for mobile clients. The library is not transparent since it requires integration on both server and client applications. In addition, the library only supports server side connection migration using an external checkpointing mechanism.

Other NF FT Techniques. FTvNF [23] combines replay of state access, similar to FTMB [45], with in-chain replication, similar to FTC [21], to provide low overhead reliability. None of these systems provide support for TCP NFs.

7 Conclusion

We present HA/TCP, a scalable framework that enables migration and failover for TCP NFs. HA/TCP provides a replicated socket interface to simplify the implementation of migration and high availability in NFs. HA/TCP fails over in milliseconds after detecting the failure. The migration is faster than the state-of-the-art systems. HA/TCP can also be used to extend these benefits to server applications.

Our code is available at https://github.com/rcslab/hatcp/

Acknowledgments

The authors would like to thank Amilios Tsalapatis, Xinan Yan, and Oscar Zhao for fruitful discussions during HA/TCP 's development. We would also like to thank the NSDI reviewers for their valuable feedback. This research is supported by the following grant: NSERC Discovery, CFI JELF, WHJIL, and NSERC CRD.

References

[1] Ieee standard for information technology - local and metropolitan area networks - part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications-aggregation of multiple link segments. *IEEE Std 802.3ad-2000*, pages 1–184, 2000.

[2] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 1, pages 329–337 vol.1, 2001.

[3] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xomb: Extensible open middleboxes with commodity servers. In *2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 49–60, 2012.

[4] Narjess Ayari, Denis Barbaron, Laurent Lefevre, and Pascale Primet. T2CP-AR: A system for Transparent TCP Active Replication. In *21st International Conference on Advanced Information Networking and Applications (AINA '07)*, pages 648–655, May 2007.

[5] Hari Balakrishnan and A Snoeren. TCP Connection Migration. Internet-Draft draft-snoeren-tcp-migrate-00, Internet Engineering Task Force, November 2000. Work in Progress.

[6] Stephen Bensley, Dave Thaler, Praveen Balasubramanian, Lars Eggert, and Glenn Judd. Data Center TCP (DCTCP): TCP Congestion Control for Data Centers. RFC 8257, October 2017.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[8] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. TCP Extensions for High Performance. RFC 7323, RFC Editor, September 2014.

[9] BSD Kernel Interfaces Manual. Common Address Redundancy Protocol, Feb 2022.

[10] Lukasz Budzisz, Rade Stanojevic, Robert Shorten, and Fred Baker. A strategy for fair coexistence of loss and delay-based congestion control algorithms. *IEEE Communications Letters*, 13(7):555–557, 2009.

[11] John Carmack. .plan File. unpublished, October 1998.

[12] Inho Choi, Nimish Wadekar, Raj Joshi, Joshua Fried, Dan R. K. Ports, Irene Zhang, and Jialin Li. Capybara: μSecond-Scale Live TCP Migration. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, page 30–36, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Cisco Systems, Inc. Best Practice Design - MX Security and SD-WAN . https://documentation.meraki.com/Architectures_and_Best_Practices/Cisco_Meraki_Best_Practice_Design/Best_Practice_Design_-_MX_Security_and_SD-WAN, 2020.

[14] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. On the performance of TCP splicing for URL-Aware redirection. In *Second USENIX Symposium on Internet Technologies & Systems (USITS 99)*, Boulder, CO, October 1999. USENIX Association.

[15] Intel Corporation. Introduction to Intel(R) Ethernet Flow Director and Memcached Performance. White paper, Intel Corporation, 2014.

[16] C. Fetzer, M. Marwah, and S. Mishra. TCP Server Fault Tolerance Using Connection Migration to a Backup Server. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, page 373, Los Alamitos, CA, USA, jun 2003. IEEE Computer Society.

[17] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, RFC Editor, June 2014.

[18] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, RFC Editor, June 2014.

[19] Edward Gately. Cloudflare outage knocks out "significant portion" of global traffic, Dec 2023.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[21] Milad Ghaznavi, Elaheh Jalalpour, Bernard Wong, Raouf Boutaba, and Ali José Mashtizadeh. Fault tolerant service function chaining. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies,*

Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 198–210, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Andrei Gurtov, Tom Henderson, and Sally Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, April 2004.

[23] Yotam Harchol, David Hay, and Tal Orenstein. Ftvnf: fault tolerant virtual network functions. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 141–147, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the Pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 535–549. USENIX Association, April 2021.

[25] David A. Hayes and Grenville Armitage. Improved coexistence and loss tolerance for delay based tcp congestion control. In *IEEE Local Computer Network Conference*, pages 24–31, 2010.

[26] J. Heffner, M. Mathis, and B. Chandler. IPv4 Reassembly Errors at High Data Rates. RFC 4963, RFC Editor, July 2007.

[27] Osamu Honda, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, and Junichi Murayama. Understanding tcp over tcp: effects of tcp tunneling on end-to-end throughput and latency. In *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, volume 6011, pages 138–146. SPIE, 2005.

[28] J. Iyengar and M. Thomson. Quic: A udp-based multiplexed and secure transport. RFC 9000, RFC Editor, May 2021.

[29] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 113–129, Boston, MA, March 2017. USENIX Association.

[30] K. Poul-Henning. Varnish HTTP Cache, Mar 2023.

[31] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association.

[32] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 501–516, Boston, MA, 2019. USENIX Association.

[33] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[34] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928, RFC Editor, March 1996.

[35] Doug Leith. H-TCP: TCP Congestion Control for High Bandwidth-Delay Product Paths. Internet-Draft draft-leith-tcp-htcp-06, Internet Engineering Task Force, April 2008. Work in Progress.

[36] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 504–517, New York, NY, USA, 2018. Association for Computing Machinery.

[37] D.A. Maltz and P. Bhagwat. Msocks: an architecture for transport layer mobility. In *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, volume 3, pages 1037–1045 vol.3, 1998.

[38] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. *SIGPLAN Not.*, 52(4):693–708, apr 2017.

[39] Stephen Nadas. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798, March 2010.

[40] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context tls (mctls): Enabling

secure in-network functionality in tls. 45(4):199–212, aug 2015.

[41] Rahul Potharaju and Navendu Jain. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, page 9–22, New York, NY, USA, 2013. Association for Computing Machinery.

[42] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: a high availability framework for middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[43] Riverbed Technology, Inc. SD-WAN Deployment Guide. https://support.riverbed.com/bin/support/static/bk3e4nsvev67aokj3qg5gtcfv4/html/ulhrppo7aoojclf7jbgjnlji07/scm_dg_html/index.html, 2022.

[44] Riverbed Technology, Inc. SteelHead CX. https://www.riverbed.com/sites/default/files/file/2021-10/steelhead-cx-data-sheet.pdf, 2022.

[45] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 227–240, New York, NY, USA, 2015. Association for Computing Machinery.

[46] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, page 175–188, USA, 2005. USENIX Association.

[47] Shinae Woo and Justine Sherry and Sangjin Han and Sue Moon and Sylvia Ratnasamy and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, 2018. USENIX Association.

[48] Silver Peak Systems, Inc. VX Virtual WAN Optimization Software. https://www.silver-peak.com/sites/default/files/infoctr/silver-peak_ds_vx-virtual-wan-optimization.pdf, 2022.

[49] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-Grained failover using connection migration. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)*, San Francisco, CA, March 2001. USENIX Association.

[50] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: connection migration for service continuity in the Internet. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 469–470, 2002.

[51] Robert Surton, Ken Birman, and Robbert van Renesse. Application-driven tcp recovery and non-stop bgp. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[52] Paul Walsh. Cloudflare outage to cause $16M of business interruption losses, Jul 2019.

[53] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 393–402, 2003.

[54] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C. Bressoud. Practical and Low-Overhead Masking of Failures of TCP-Based Servers. *ACM Trans. Comput. Syst.*, 27(2), may 2009.

[55] Victor C. Zandy and Barton P. Miller. Reliable Network Connections. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, page 95–106, New York, NY, USA, 2002. Association for Computing Machinery.

[56] Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh. SKQ: Event scheduling for optimizing tail latency in a traditional OS kernel. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 759–772. USENIX Association, July 2021.

High-level Programming for Application Networks

Xiangfeng Zhu[w] Yuyao Wang[w] Banruo Liu[w]* Yongtong Wu[w]† Nikola Bojanic[w]
Jingrong Chen[d] Gilbert Louis Bernstein[w] Arvind Krishnamurthy[w] Sam Kumar[w][l]
Ratul Mahajan[w] Danyang Zhuo[d]

[w]*University of Washington* [d]*Duke University* [l]*UCLA*

Abstract

Application networks facilitate RPC communication between the microservices of cloud applications. They are built today using service meshes which employ low-level specifications and have high overhead—it is difficult to express even simple application-specific functionality (e.g., access control based on RPC fields) and RPC latency can more than double. We develop AppNet, a framework for building expressive and high-performance application networks. Developers specify rich application network functions (ANFs) in a high-level language with generalized match-action rules. We compile the specifications to high-performance code after optimizing *where* (e.g., client, server) and *how* (e.g., RPC library, proxy) each function runs. The optimization uses symbolic abstraction and execution to judge if different runtime configurations of possibly stateful functions are semantically equivalent for arbitrary RPC streams. Our experiments show that AppNet can express common ANFs in only 7-28 lines of code. Its optimizations lower RPC processing latency by up to 82%.

1 Introduction

Modern cloud applications are composed of a collection of microservices instead of monolithic binaries [10, 50, 56]. This architecture speeds up software development because each microservice can use the language and libraries best suited to its task. It also improves scaling because each service can be scaled (via replicas) independently.

However, when applications are disaggregated, many simple function calls become remote procedure calls (RPCs) over the network. To secure and manage this communication, developers build *application networks*, also known as service meshes, using chains of application network functions (ANFs) that implement authentication, fault tolerance, logging, load balancing, etc. Enabled by RPC traffic processing platforms (e.g., Envoy proxy [19], RPC library plugins [18]) and by controllers such as Istio [39], application networks are ubiquitous today. Nearly all cloud application developers use them [7].

In theory, it should be easy to build performant application networks that are specialized to application needs. After all, an application network is meant to support known application(s) and functionality, in contrast to the Internet which is meant to support arbitrary applications. But the current reality is that implementing even simple application-specific functionality, such as access control or routing based on RPC fields, is challenging [1, 70, 76], and application networks can increase RPC latency as well as CPU usage by 2-7x [3, 5, 6, 66, 76, 86].

These shortcomings stem from the low-level programming model of current application networks. Developers can either write custom ANFs in general-purpose languages such as C++ and Rust or use generic HTTP modules. Implementing application-specific functionality is difficult either way. Custom ANFs are difficult to write because high-level abstractions do not exist, and a separate ANF must be written for each RPC processing platform. That is why HTTP modules, used after wrapping RPC traffic in HTTP, are more popular. But using them requires modifying the microservice code to expose relevant RPC information as HTTP headers [25, 70, 76]. Such modifications are onerous, especially when the microservice code is authored by third parties (a common case).

Low-level programming of application networks also hurts performance. There are different locations, including the client, the server, or another remote host [13], and different platforms where ANFs can be run. Different options offer different trade-offs [70, 76]—e.g., RPC libraries have low overhead but cannot be used with untrusted microservice code; and rate limiting total requests to a microservice is better done at the server-side to eliminate the coordination overhead among clients. Today, developers must explicitly configure the order, location, and platform for each ANF. However, it is difficult to manually determine high-performance configurations because performance depends on the interactions between various ANFs and on the (dynamically varying) number of client and server replicas. Since the semantics of ANFs are unknown to current network controllers, they must execute exactly what the user provides, even if semantically-equivalent configurations with significantly better performance exist.

*Now with UIUC.
†Now with Peking University.

Our goal is to enable expressive, easy-to-build, and high-performance application networks. We explore an approach based on high-level programming, inspired by works that achieve a similar goal for layer-3/4 networks [43,44,65,68,80] and switch forwarding [46,47,54,74]. We develop AppNet, a framework for building application networks which has (1) a high-level language to specify ANFs, (2) a compiler that generates an optimized configuration across multiple platforms, and (3) a controller that instantiates the configuration.

With AppNet, developers specify network functionality as a chain of *elements*, where each element expresses some type of RPC processing (e.g., load balancing). Our language offers a generalized match-action primitive where matches can be performed on RPC fields, state, or results of built-in functions (e.g., *random*) and actions can be reading or writing RPC fields and state. This primitive is expressive enough for RPC processing (unlike others we tried, such as SQL dataflow [9]) and yet can be compiled to high-performance code for multiple platforms. AppNet also provides primitives for managing shared state between multiple instances of the same element and shared state with the control plane. These primitives make it trivial to express functionality such as global rate limiting that may require coordination across element instances.

Based on element specifications, the AppNet compiler outputs runnable modules for various platforms. The mapping between elements and runnable modules is not one-to-one and is optimized for performance. A module may contain multiple elements to reduce overhead, the element order may be altered (e.g., moving elements that drop RPCs upstream reduces work for downstream elements), elements may be migrated based on state synchronization costs, or elements may be co-located in selected locations to completely bypass the (expensive) invocation of some platforms.

It is critical that the output of the compiler be semantically equivalent to the user's specification. Determining equivalence is challenging because elements are stateful (thus, processing an RPC impacts subsequent RPCs) and RPCs can be dropped or reordered (which can impact downstream state). AppNet uses a form of symbolic abstraction and symbolic execution [59] to determine when two chains of elements, each a possible runtime configuration, are equivalent for arbitrary streams of RPCs. It abstracts each element as symbolic transfer functions that capture its side effects (e.g., RPC field modifications and state updates). It then uses symbolic execution over these functions to compute the processing semantics of a chain, including changes to the element state and auxiliary outputs such as logs. The outputs of this execution help determine whether two chains are equivalent.

As part of its compilation, AppNet allows users to opt for weak consistency and improve performance. It can weaken consistency in two ways: *state consistency*, which controls the consistency of shared state among element instances, and *observation consistency*, which controls whether auxiliary outputs such as logs record the same information. Weak consistency may not always be appropriate, but applications that can tolerate them can experience higher performance. Such modulation of network behavior based on application characteristics is a unique opportunity for application networks.

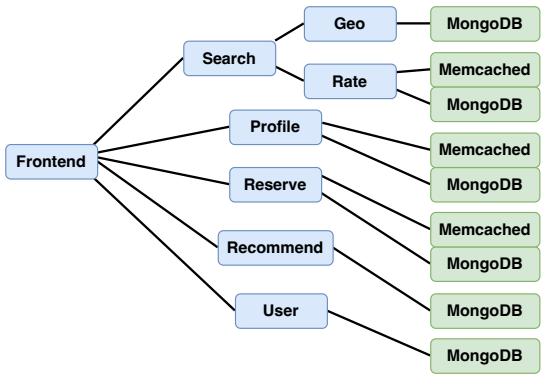


Figure 1: Hotel Reservation application's microservices [53].

Our current implementation of AppNet [16] supports three popular RPC processing platforms: EnvoyNative [31], EnvoyWasm [40] (respectively, C++ and WebAssembly-based execution environments in Envoy), and gRPC interceptors [29]. The Envoy platforms support deployment as a proxy located with the client or server ("sidecar") or as a remote proxy.

We evaluate AppNet by implementing 14 ANFs and studying a range of chain specifications. We find that 12 common ANFs need only 7–28 lines of code, and 2 more complex ANFs which integrate load balancing and routing—Meta's ServiceRouter [70] and Google's Prequal [82]—need only 62 and 88 lines. AppNet's compiler optimizations lower RPC processing latency by up to 82% and CPU usage by up to 75%, compared to unoptimized chain configurations.

Our results show that it is possible to build application networks that are both expressive and performant. Decoupling network specification from the running implementation opens up new capabilities too. One such capability, which AppNet already supports, is consistently upgrading the network when specifications change or microservices scale. The upgrade ensures that no RPC ever experiences a mix of old and new network policies, which can lead to performance anomalies and policy violations [4,69]. We are exploring others, including offloading RPC processing to kernel or hardware and serializing RPCs based on fields accessed in the network.

2 Background

Microservices are the dominant paradigm for modern cloud applications. Figure 1 shows a typical microservices-based application. Intended to enable hotel reservations, its functionality is split across eight types of microservices. Clients contact (a replica of) the Frontend service, which then communicates with a middle-tier service depending on the query. The middle-tier services handle tasks such as searching for hotels, completing hotel bookings, updating user profiles, and provid-

Function	Description
Fault Injection [23]	Randomly drop an RPC
Cache [22]	Generic cache for RPC responses
Rate Limiting [24]	Control the rate of RPC requests
Load Balancing [26]	Load balance across service replicas
Logging [20]	Record the RPC content to disk
Mutation [30]	Modify the RPC content
Application Firewall [15]	Rejects an RPC based on pattern matching
Metrics [28]	Monitor the RPC latency and success rate
Admission Control [45]	Drop RPC requests based on success rates
Encryption [12]	Encrypt certain RPC fields
Bandwidth Limit [21]	Control the amount of data to the service
Circuit Breaking [32]	Limit the impact of undesirable network peculiarities

Table 1: Common ANFs for microservices.

ing recommendations. They may contact other microservices and storage infrastructure. Production settings can have anywhere from 1 to 100s of replicas of each microservice [10,56], communicating via remote procedure calls (RPCs).

To manage communication between microservices, developers build application networks with rich functionality. Table 1 lists 12 common ANFs which we identified by surveying popular platforms to build application networks [11,14,30,31,39].

As Figure 2 shows, ANFs can be run in different locations and using different platforms. Common platforms include: (1) RPC library [18,70], which is embedded directly into the client or server microservice's code to manage communication without external proxies, (2) sidecar [19,35], a proxy co-located with the client and server that intercepts all traffic, and (3) remote proxy [13,76], a standalone proxy that handles traffic between services. Simple RPC processing may be located in the kernel (using eBPF) as well [17]. We will extend our work to this platform in the future.

ANFs are implemented in a platform-specific manner, and developers typically invoke them via *service mesh* controllers [35,39]. The controllers configure the target platform and also configure RPC traffic interception and redirection. Regardless of whether ANFs run inside an RPC library, a proxy, or both, developers must manually decide their execution order, platform, and placement.

This decision is complex and significantly impacts performance. As an example, consider the communication between two replicas of Search and one of Geo (Figure 3). The developer wants to run Logging, Load Balancing, Rate Limiting, and Application Firewall ANFs from Table 1 between the microservices. Rate Limiting uses a stateful token bucket to control the RPC request rate to a Geo replica. Figure 3a shows a possible configuration provided by the developer.

However, factoring in the properties of the platforms and ANFs' semantics, configurations with better performance exist. We could move Rate Limiting to the server side, which eliminates the need for synchronizing state between multiple ANF instances because the state depends on server replica, and we could move Logging to the RPC library because that platform has lower overhead. Figure 3b shows an optimized configuration for the chain.

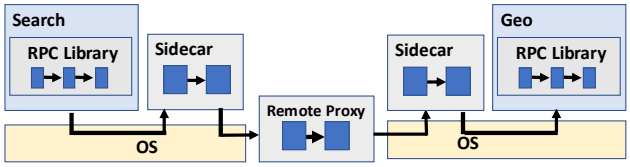
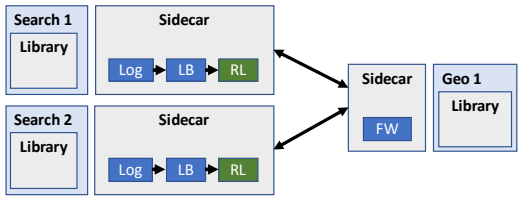
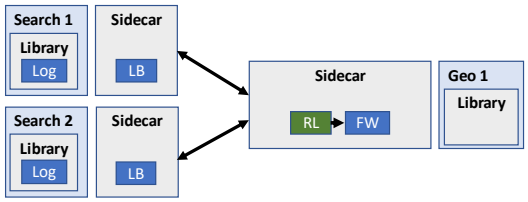


Figure 2: RPC processing between Search (client) and Geo microservices. Dark blue rectangles are ANFs.



(a) User-provided configuration



(b) Optimized, semantic-preserving configuration

Figure 3: Two possible configurations for an application network. 'Log' is Logging, 'LB' is Load Balancing, 'FW' is Application Firewall, and 'RL' is Rate Limiting (stateful).

Not all performance-enhancing configurations preserve the semantics of the input chain. Identifying optimized configurations that preserve semantics requires careful reasoning about semantics and state dependencies. For example, reordering Application Firewall and Rate Limiting and moving the Application Firewall to the client side can improve performance by preventing RPCs that would eventually be dropped from traversing the network. Yet, this reordering would violate the input semantics, as Rate Limiting's internal state could be affected depending on whether Application Firewall drops a packet before or after it.

The current architecture of application networks—with low-level, platform-specific ANFs whose ordering, location, and platform are manually determined—has evolved organically, and it has the limitations mentioned earlier: difficult-to-express application-specific functionality and poor performance. We ask what a principled design for a framework

to build application networks might look like and if it can effectively address these limitations.

3 AppNet Overview

AppNet provides high-level programming abstractions for application networks. The abstractions simplify the specification of desired network functionality and enable optimizations that help realize the functionality with low overhead. This section overviews the abstractions, the optimizing compiler, and the runtime controller.

3.1 Programming abstractions

Like high-level languages for layer-3/4 networks [43], the AppNet language focuses on the data plane, which has complex, performance-sensitive processing. Some ANFs have a control plane as well. For load balancing, the data plane forwards the RPCs to different replicas, and the control plane maintains the list of active replicas. AppNet provides shared state abstractions that enable the control plane to communicate easily with the data plane, but it does not provide high-level abstractions for control plane logic.

With AppNet, users specify network functionality between a pair of microservices using chains of *elements*, where each element encodes an ANF. Key aspects of the specification language are:

Generalized match-action rules RPC processing is specified as match-action rules that operate over RPC fields (not arbitrary low-level code for standard protocols), state, and built-in functions. RPC fields (both metadata and payload) are represented as key-value pairs, which the elements can easily read/write without worrying about (de)serialization. The state is also represented as key-value pairs.

Match-action rules are natural for RPC processing, and they give the compiler visibility into which RPC fields are accessed and what the state depends on. At first glance, the match-action paradigm may seem limiting for layer-7 processing, but we have found it to be rich enough to encode a wide range of ANFs. This encoding is made possible by allowing richer match expressions and actions than switch languages like P4 [46]. Match expressions go beyond static table lookups and can use a suite of operators and built-in functions, and actions can read/write state multiple times.

Shared-state management Shared state, between multiple instances of the same ANF (e.g., for global rate limiting) or between control and data planes, is common in application networks. Developers manage this state manually today, which is onerous. AppNet abstracts away this state management and provides APIs to read-write state. This abstraction lets us reduce synchronization costs by placing elements and state based on what the state depends on. If the state depends on server replica, we prefer placing the element on the server side so there will be only one reader-writer to this shard of the state; and if we can do that, we can locate the shard on the server as well. Shared state management also lets us trade off consistency and performance, as we discuss next.

Performance-consistency tradeoff AppNet lets users trade consistency for performance in two ways. First, they can opt for weak (eventual) consistency for shared state. This trade-off is well-known for distributed systems [49,85]. Users can also specify weak observation consistency. Two network configurations have strong observation consistency when all outputs—the microservices themselves and auxiliary outputs like logs—get the same content in both configurations. Weak observation consistency permits the auxiliary output channels, but not microservices, to get different content, e.g., logs will differ when logging occurs before versus after an ANF that modifies the RPC's *geo-region* field. Applications that can tolerate weak observation consistency can get higher performance because it creates more optimization opportunities.

Location & platform agnostic elements, paired elements AppNet does not require users to specify an element's location or platform explicitly (though they can pin if they want). This under-specification enables the compiler to determine the location and platform based on performance. Thus, while AppNet uses element chains as a specification primitive like application networks today, a key difference is that AppNet chains specify processing semantics, not execution order.

AppNet also lets users define pairs of elements that must be used in concert (e.g., compression/decompression). For paired elements, AppNet ensures that one element is allocated to the client side and the other to the server side, and conflicting elements are not placed between them. This primitive simplifies specification and prevents errors such as forgetting the decompression element or placing compression between encryption and decryption [1].

3.2 Compiler

The AppNet compiler takes in the network specifications and RPC definitions (e.g., Protobuf [37]) used by the application code. Using RPC definitions during compilation frees the developer from having to write RPC (de-)serialization code, helps validate the specifications (e.g., RPC field names), and ensures that the generated code is up-to-date with RPC definitions. After parsing and sanity-checking the specifications, the compiler tags each state variable with what it depends on (e.g., client-replica, server-replica, global, etc.) based on what the state is indexed on. This information is used for optimizing element and state placement.

The compiler outputs modules that can run on one or more platforms (e.g., RPC library, sidecar). The heart of this process is computing a performant runtime configuration that is semantically equivalent to the user input. In the output configuration, elements may be reordered and combined into one module. The challenge in this computation is that elements can be stateful, which means that we must analyze equivalence for arbitrary streams of RPCs, not individual RPCs;

state updates caused by an RPC impact all subsequent RPCs. The analysis must also account for RPCs getting dropped or reordered—a stateful element that is upstream of a rate limiter, which drops RPCs, will see a different RPC stream than when it is moved downstream, leading to different state updates. Statefulness and the possibility of dropped/reordered streams make it hard for us to reuse equivalence analysis developed in domains such as stateless packet forwarding [43] or instruction reordering [77].

AppNet solves this challenge by first symbolically abstracting elements. The abstraction captures all of an element's side-effects and dependencies using transfer functions, without capturing the exact processing logic. For the rate limiter, it will capture that it may drop an RPC, that its internal state is updated when an RPC appears, and that this update does not depend on the RPC content. AppNet then uses symbolic execution over chains of element-level transfer functions to determine end-to-end transfer functions of a chain, including output RPCs, state updates, and what is sent to auxiliary outputs such as logs. Two chains are deemed equivalent if their end-to-end transfer functions are equivalent.

3.3 Controller

The AppNet controller maintains an up-to-date, global view of all service replica locations by integrating with cluster managers such as Kubernetes [33]. It orchestrates the deployment of runnable modules generated by the compiler, ensuring they are deployed to the appropriate replicas and platforms. It also ensures that configuration updates are consistent.

4 AppNet design

We now describe the AppNet language and compiler in detail. Appendix A describes code generation and consistent updates.

4.1 Specification language

Figure 4 shows the core of the AppNet grammar, without type annotations and syntactic features that enhance usability. Figure 5 show an example specification. At the highest level, RPC processing between two microservices is described using four sub-chains of elements: two for those that must run on the client or server sides, one for those that can run anywhere, and one for paired elements (e.g., encrypt/decrypt). For paired elements, the first element is run on the client side and the second on the server side. It is not mandatory to have an element in any of these sub-chains. Users can include all elements in the "any" sub-chain. The implied processing order for elements is client, pair(client), any, pair(server), server; the compiler may re-order the elements, but it will preserve this semantics. Users can optionally specify weak observation consistency. Figure 5a shows the AppNet chain specification for the example in Figure 3, where only the "client" and "any" chain are specified.

An AppNet element specification has four sections. The `state` section declares variables that hold local or shared state

$$
\begin{aligned}
\textit{Chain} ::= &\left[\begin{array}{l}\textbf{client}: \textit{Element}^* \\ \textbf{any}: \textit{Element}^* \\ \textbf{server}: \textit{Element}^* \\ \textbf{pair}: (\textit{Element}, \textit{Element})^* \\ [\textbf{weak}]\end{array}\right.\\
\textit{Element} ::= &\left[\begin{array}{l}\textbf{state}: \textit{Decl}^* \\ \textbf{init}(\textit{Var}^*): \textit{Assign}^* \\ \textbf{req}(\textit{Var}): \textit{Action}^* \ [\textit{MatchAction}] \\ \textbf{resp}(\textit{Var}): \textit{Action}^* \ [\textit{MatchAction}]\end{array}\right.\\
\textit{Decl} ::= &\ \textit{Var}\ [\textbf{shared}\ [\textbf{weak}\ [\textbf{sum}]]]\\
\textit{MatchAction} ::= &\ \textbf{match}(\textit{Expr})\ \textit{Case}^+\ [\texttt{'*'} \Rightarrow \textit{Action}^+]\\
\textit{Case} ::= &\ \textit{Literal} \Rightarrow \textit{Action}^+\\
\textit{Action} ::= &\ \textit{Assign} \mid \textit{Send} \mid \textit{Foreach} \mid \textit{Return}\\
\textit{Assign} ::= &\ \textit{Var} = \textit{Expr} \mid \textbf{set}(\textit{Var}, \textit{Expr}^+, \textit{Expr})\\
\textit{Send} ::= &\ \textbf{send}(\textit{Message}, \textit{Channel})\\
\textit{Foreach} ::= &\ \textbf{foreach}(\textit{Var}, \textit{LambdaFunc})\\
\textit{Return} ::= &\ \textbf{return}\ [\textit{Expr}]\\
\textit{Message} ::= &\ \textit{Var} \mid \texttt{'error'}\\
\textit{Channel} ::= &\ \textbf{down} \mid \textbf{up} \mid \textit{Var}\\
\textit{Expr} ::= &\ \textit{Literal} \mid \textit{Var} \mid \textbf{get}(\textit{Var}, \textit{Expr}^+[, \textit{LambdaFunc}])\\
&\mid \text{BuiltinFunc}(\textit{Expr}^*)\\
\textit{LambdaFunc} ::= &\ \textbf{lambda}(\textit{Var}+) \Rightarrow \textit{Action}^*\ [\textit{MatchAction}]\\
\textit{Var} \in &\ (\text{set of variable names})\\
\textit{Literal} \in &\ (\text{literal values, e.g. } 0.1, 42, \textbf{true})
\end{aligned}
$$

Figure 4: AppNet specification language.

among all running instances of the element (serving different microservice replicas). Shared state declarations optionally accompany desired consistency and aggregation methods. By default, all reads and writes to shared state are synchronous. If users specify weak consistency, AppNet maintains a local copy and syncs this copy in the background (thus providing eventual consistency). During the sync, the default semantics is last-writer-wins, but users can specify custom aggregations (akin to CRDTs [73]). Sum (useful for counters) is a supported aggregator currently, and we will add more as needed.

The `init` section initializes the state variables. Initialization may use user input. The `req` and `resp` sections have match-action-based processing for RPC requests and responses. AppNet invokes these methods with a map that represents the content of the RPC payload and metadata (which includes RPC sender, destination, and method type). Users can match on any valid expression (Exp), and they can specify multiple actions for each case. Actions either update state or send a message. The message content can be the RPC (with its content represented by Var) or an Error. Messages can be sent downstream (`down`) to the next element or, if it is the last element in the chain, to the microservice; they can also be

```
client: logging()->load_balancing()
any: rate_limiting()->firewall()
```

(a) An element chain specification with four elements.

```
1  state:
2      bucket shared
3
4  init():
5
6  req(rpc):
7      key = get(rpc, 'dst')
8      has_token = get(bucket, key, lambda(token):
9          match token > 0:
10             true =>
11                 token = token - 1
12                 return true
13             false =>
14                 return false
15     )
16
17     match has_token:
18         true =>
19             send(rpc, down)
20         false =>
21             send(err('rate limited'), up)
22
23 resp(rpc):
24     send(rpc, up)
```

(b) The element specification for rate limiting.

Figure 5: Example AppNet specifications.

sent upstream (`up`) to the previous element, which is used for sending back errors or cached responses. AppNet checks at runtime if transmitted content can be cast into the expected RPC. In addition to the `up` and `down` channels, users can declare (in `state` section) and use auxiliary channels (e.g., for logging to disk).

To access state, AppNet provides `get` and `set` functions, enabling the compiler to infer state dependencies and determine whether operations should be local or remote. The `get` function can accept an optional lambda for computations, with updates to shared state being atomic. When a lambda is provided, the original return value of the `get` function is passed as an argument to the lambda. AppNet also supports various computations on RPC fields and states using built-in functions, including mathematical operators, random number generators, and utilities for encryption and compression.

Figure 5b shows the AppNet specification for an element that limits the rate of RPCs to each server replica using a token bucket. The `state` section defines a shared state variable called bucket that tracks available tokens. This element assumes that the control plane (not shown) refreshes the token bucket periodically by writing to this shared state. During request processing (Lines 6-21), the element checks if a token is available in the specified key of the bucket (state variables are key-value maps). The third argument of the `get` function specifies the logic that the data store should execute during the `get` operation. If a token is available, it is decremented.

4.2 Optimization

The AppNet compiler finds a high-performance configuration (location, platform, and execution order of each element) that respects users' constraints and is semantically equivalent to the input specification. We face a few challenges in computing such a configuration. First, there is a large space of possible configurations. For a chain of n elements with k placement options (location/platform combination) for each, there are $O(n! \cdot n^k)$ possible configurations. Second, we need to know which configurations are semantically equivalent to the user input, which is difficult for reasons mentioned in § 3.2. Third, multiple competing concerns (e.g., network traffic, CPU usage, latency, synchronization cost) make performance estimation difficult. For instance, positioning elements that drop RPCs early in the chain can reduce CPU usage and network traffic, but it can increase synchronization costs by preventing the consolidation of stateful elements.

AppNet addresses these challenges using a multi-start simulated annealing framework because it works well with large, intractable search space. Starting from an initial configuration with all elements randomly mapped to locations and platforms, the algorithm iteratively applies semantic-preserving mutations to the current solution and evaluates the new solution using a cost function. It decides whether to adopt the new solution based on the cost difference and a probability function. Finally, it periodically resets the temperature to avoid getting trapped in local optima.

We describe the cost function below, which is a key determiner of the final output. Appendix E provides more details on mutation strategies, termination conditions, and evaluation results on the optimality and scalability of the framework.

Cost function. Our cost function is a heuristic that combines six preferences. In priority order, they are: (1) Prefer platforms with lower processing costs (e.g., RPC library). (2) Prefer chains where all elements are located on a single platform, as this enables bypassing other platforms (e.g., server-side sidecar), significantly reducing processing overhead. (3) Prefer placements that align with state dependency to avoid synchronization cost. So, elements sharing a state indexed on a client replica should be located on the client side. (4) Prefer co-locating elements that require strong state synchronization (because dependencies are not aligned). Co-location facilitates the consolidation of state synchronization messages, reducing synchronization costs. (5) Same preference, but for elements with weak state consistency. (6) Prefer placing elements that may drop RPCs earlier because it reduces work for downstream elements.

We combine these preferences into a cost function in three steps. First, we compute element-level costs, which are represented as tuples with three components: the estimated frac-

tion of RPCs reaching the element, a binary value indicating whether the element requires strong state synchronization, and a binary value for weak state synchronization. The fraction of RPCs reaching an element is determined by the expected drop rates of upstream elements, set to 0 for elements that do not drop RPCs and a configurable value for those that might drop them. The default value is 0.05, but it can be provided by the developer or informed via runtime telemetry.

Second, we divide the input chains into sub-chains based on location and platform, and compute the cost of each sub-chain by aggregating the element-level cost tuples into a new tuple (with the same three components). The first component of the tuple sums the RPC fractions, reflecting the total processing cost. The second and third components apply a boolean *or*, indicating that synchronization costs are paid once per platform, regardless of how many elements require synchronization.

Finally, we aggregate sub-chain costs into the final chain-level cost. We convert the sub-chain cost tuples into scalar values by treating the binary components as 0/1 and computing a weighted sum of the three values using weights (0.5, 2, 1) for the three dimensions. We then add the baseline platform processing cost of S_p if there are any elements in the sub-chain for platform p. The chain-level cost is the sum of these five sub-chain-level costs. We use $S_p = 2$ for the out-of-process proxy (either sidecar or remote proxy) and $S_p = 0.1$ for the RPC library. We have found our results to be robust to exact values as long as the ordering between parameters is maintained. In the future, we plan to inform these parameters based on runtime profiling of different overheads [86].

4.3 Equivalence checking

There are three steps for deciding if two chains being considered during the optimization are equivalent: (1) symbolically abstracting what each element does; (2) symbolic execution of the chains based on the abstract elements; (3) comparing the outputs of this execution for two chains. We provide an overview of these steps below. Appendix B has the details and a proof of correctness.

Symbolic abstraction of elements This analysis creates an abstract representation of an element that captures the side effects of the element based on its dependencies. The side effects include modifications to the RPC, updates to the internal state, and writes to auxiliary output channels. The representation does not capture the actual processing.

The abstract representation is a transfer function whose parameters include the incoming RPC and the pre-state (prior to RPC processing). We model RPCs using their metadata and payload fields plus two special fields called *drops* and *reorders*. These fields capture whether the RPC could have been dropped or reordered.

Figure 6 (left) shows the transfer functions for three elements inside the green boxes. It assumes that incoming RPCs have two fields *dst* (for destination address) and *user* (for caller's identity) and the special field *drops* (we elide the special field *reorder* for simplicity). Logger logs the incoming RPC to an auxiliary output channel *out*, and so its transfer function appends the incoming RPC to the channel. Load Balancer is a round-robin load balancer that determines the destination using a state variable *idx* that indexes into a list of replicas and is incremented (with wrap around) after each RPC. Load Balancer has two transfer functions. One updates *RPC.dst* based on *idx*—$\mathbf{LB_R}$ is an opaque function that captures this dependency. The other updates *idx* based on an opaque function of *idx* and the *RPC.drops* field because if the RPC were dropped *idx* will not be updated. The transfer function of Firewall updates *RPC.drops* based on *RPC.user*.

We derive these transfer functions using static analysis on the element specification. For each RPC field and state variable modified by the element, a transfer function is generated that replaces the corresponding value with an opaque function representing the internal logic of the element, together with all symbols that the element reads, which serve as the dependencies for the write. For each element that may drop RPCs, a transfer function is generated that puts a new symbol representing the potential dropping event into the RPC special field. Reordering is handled in a similar way.

Symbolic execution of chains Symbolic execution of a chain produces a chain-level transfer function for the output RPC and for auxiliary channel outputs and state updates for all elements. This analysis essentially combined the element-level transfer functions.

Figure 6 (left) shows the end result of an RPC passing through that example chain. The auxiliary channel *out* outputs the exact RPC that came in; the internal state *idx* of the load balancer is updated by the opaque function $\mathbf{LB_S}$ whose parameters are the value of *idx* prior to RPC's arrival and the empty set for *RPC.drops* field (because the update will be different based on whether the RPC was dropped); and the final RPC has field values that depend on two opaque functions ($\mathbf{LB_R}$ and **FW**).

Comparing two chains Comparing two chains means comparing all of the chain-level transfer functions (result of symbolic execution). When the transfer functions have opaque functions, they are deemed equal iff their parameters are (recursively) equal; thus, this is basically a syntactic check. For strong observation consistency, we look for all transfer functions being equal. For weak observation consistency, we ignore the transfer functions for auxiliary channel outputs.

Figure 6 shows the results of this comparison when the chain on the left is compared against two other options with different element ordering. Option 1 is deemed weakly equivalent because the two transfer functions differ only for the auxiliary output of Logger. These transfer functions differ because the Logging element will record RPCs with a different *dst* field value once it is placed downstream of the Load Balancer. Option 2 is deemed not equivalent even though it

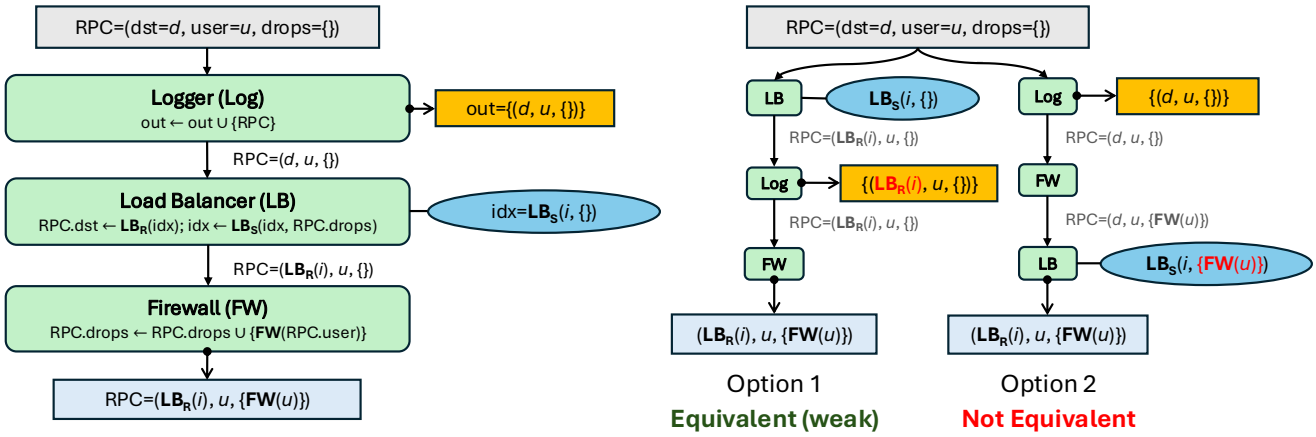


Figure 6: Equivalence-checking example. The transfer functions of the elements are inside the rounded green boxes. The rectangular boxes are channel outputs, and the oval ones are state updates determined by symbolic execution. Equivalence with strong semantics checks all outputs and state updates, and equivalence with weak semantics ignores yellow boxes (auxiliary outputs). The symbols marked red are those different from the original ones.

has identical transfer functions for Logger output and the final output RPC. The problem is that the transfer function for the state update for Load Balancer changes when it is placed downstream of the Firewall. This difference is surfacing the subtlety that Load Balancer will send subsequent RPCs to different destinations because the behavior of its round-robin logic is impacted by dropped RPCs.

5 Implementation

We implemented the AppNet compiler using 16K lines of Python and the controller using 3.1K lines of Go that integrates with Istio [39] and Kubernetes [33]. Developers use a custom Kubernetes resource [34] to provide AppNet specifications. The controller watches for changes to this resource and the application deployment (e.g., a new replica is launched) and updates the data plane platform appropriately.

AppNet currently supports three RPC processing platforms: (1) gRPC, a popular RPC library, for which we produce modules in Go that can be deployed as gRPC interceptors [29]; (2) EnvoyNative, for which we produce modules that C++ Envoy filters [31]; and (3) EnvoyWasm, a WebAssembly-based execution environment provided by Envoy [40], for which we produce modules in Rust that are later compiled to WebAssembly using the Rust compiler.

The overhead of these platforms increases in the listed order and the platforms differ in other ways as well (which makes the choice of the right platform a complex one). The gRPC modules run within the client or server address space, and so this platform cannot be used to run elements for which microservice code cannot be trusted. gRPC and EnvoyWasm allow dynamically updating running modules (without downtime); EnvoyNative does not. EnvoyNative and EnvoyWasm can be co-located with the client or server (as a sidecar proxy) or run in a remote proxy [13]; gRPC cannot be executed via a remote proxy.

AppNet assumes that its target platform provides access to RPC headers and payloads in plaintext (i.e., without mTLS). gRPC interceptors are always executed before mTLS is applied, and Envoy (in both sidecar and remote deployments) can be configured to terminate the mTLS connection [8]. In scenarios where end-to-end mTLS is required, AppNet will limit its target platforms to gRPC interceptors to ensure compliance with encryption requirements.

To synchronize shared state with strong consistency, we adopted the approach used by Envoy's global rate limiter [27] and StatelessNF [57]. Specifically, we use Redis [38] and route every state access through it. The synchronization for weak state consistency uses the same approach, but the synchronization is moved to the background.

Compiler benchmarks Finding the optimal runtime configuration takes 1.4s for 5-element chains that we use in § 6.2. Generating modules for gRPC interceptors and EnvoyWasm takes a few seconds, but it takes 1-2 minutes for EnvoyNative because that involves compiling the Envoy codebase.

6 Evaluation

Our evaluation aims to answer the following: (1) Can AppNet easily express common ANFs? (2) Can it reduce the overhead of application networks? (3) Does that reduction improve application performance?

6.1 Expressiveness

To evaluate the expressiveness of AppNet, we implement the 12 common ANFs in Table 1 which we identified based on a survey of common ANFs. As examples of more complex functionality, we also implement routing and load balanc-

	Shared state	LoC (AppNet)
Fault Injection [23]		11
Cache [22]	✓	14
Rate Limiting [24]	✓	25
Load Balancing [26]	✓	13
Logging [20]		10
Mutation [30]		7
Application Firewall [15]	✓	12
Metrics [28]		12
Admission Control [45]		21
Encryption [12]		28
Bandwidth Limit [21]		21
Circuit Breaking [32]		16
ServiceRouter [70]	✓	62
Prequal [82]	✓	88

Table 2: ANFs implemented in AppNet.

ing in Meta's ServiceRouter [70] and Google's Prequal [82]. Appendix F details these implementations.

We are able to express all 14 ANFs in AppNet. As a proxy for ease of use, Table 2 shows the lines of code (LoC) of each. The 12 common ANFs need only 7-28 lines, and ServiceRouter and Prequal need only 62 and 88 lines. Some elements have a control plane component (e.g., for maintaining the replica list for load balancing). These lines do not include those components. We find it encouraging that so few lines are needed to express the data plane of ANFs.

For comparison, we also implement the same ANFs in the native languages of the platforms (Go for gRPC, C++ for EnvoyNative, Rust for EnvoyWasm). These ANFs need 5-60x more lines than AppNet elements. C++ modules are the worst, needing 4x more lines than Go and Rust modules.

High-level languages can hurt performance if the generated code is not as efficient as hand-optimized code. We find that the processing time and CPU usage of AppNet elements is only 1-4% higher than similar filters [31] bundled with Envoy. See Appendix D for details.

The value of AppNet specifications lies not only in their compactness but also in enabling optimizations that lower overhead. We evaluate this next.

6.2 RPC Processing Overhead

To assess how well AppNet reduces the RPC processing overhead, we need a corpus of network specifications used in practice. But such a dataset does not exist to our knowledge, so we create a wide array of network specifications randomly and benchmark the range of overhead reduction. This method helps understand specification characteristics that lead to higher or lower overhead.

We create a dataset by randomly picking elements in the chain from the 12 common ANFs in Table 2. We study chains of sizes 3 and 5 and randomly assign each element to a subchain (client, server, or any). We also randomly assign platform constraints, which include "do not run in an RPC library" and "requires dynamic upgrades." The first constraint disallows gRPC, and the second constraint disallows EnvoyNative. Modulo these location and platform constraints, AppNet is free to optimize the chain as it sees fit.

Experimental setup We consider three metrics: (1) Service Time is the time to complete an RPC request when the service has a minimal load (at most one outstanding request); (2) Tail Latency is the 90th percentile latency with moderate load (30-40% CPU utilization); and (3) CPU Usage is the number of virtual cores used for processing RPCs.

We use Echo Server as the application which has a frontend microservice that sends an echo request and a backend microservice that responds with an echo response. This simple application helps us microbenchmark RPC processing overhead. We study more complex applications later. All applications in this paper use Go and gRPC, and all experiments use five replicas per microservice. The results are qualitatively similar for different replica counts.

We quantify AppNet's overhead reduction by comparing against two baselines: (1) *NoOpt* randomly assigns elements while honoring explicit location or platform constraints, (2) *LocalOpt* favors platforms with lower processing costs (first gRPC, then EnvoyNative, then EnvoyWasm) and prioritizes placements that align with state dependencies (e.g., placing elements that depend on client replicas on the client side). NoOpt captures what may happen when developers do not attempt any performance optimization, and LocalOpt captures what an informed developer might do—optimize for individual elements—excludes chain-wide optimizations that are difficult to do manually. Both baselines maintain the element ordering of the input specification and make placement decisions for elements in order. Thus, choices available for an element depend on the choices made for upstream elements.

We use Cloudlab [51] machines with two 16-core Intel Xeon Gold 6142 CPUs (2.6 GHz) and 384GB RAM, Ubuntu 20.04 (Linux kernel v5.4.0), Kubernetes v1.28.13, and Envoy 1.30.5. We disable TurboBoost, CPU C-states, and dynamic CPU frequency scaling to reduce measurement variance. We use wrk [41] and wrk2 [42] for load generation as well as high-precision measurement of latency. The remote proxy is shared by all replicas of the client and server microservices.

6.2.1 Overhead reduction

Figure 7 compares the performance of LocalOpt and AppNet with strong and weak consistency against NoOpt. It plots the reduction in the three metrics for 30 randomly selected 5-element chains. The absolute metrics of NoOpt (not shown in the graph) are 0.8–2.0 ms for service time, 2.5–12.1 ms for tail latency, and 10.9–28.6 virtual cores for CPU usage. These numbers exclude the processing time and CPU usage of the Echo Server application itself.

Both LocalOpt and AppNet show improvements across all three metrics. LocalOpt achieves a median reduction of 25% in service time, 27% in tail latency, and 22% in CPU usage.

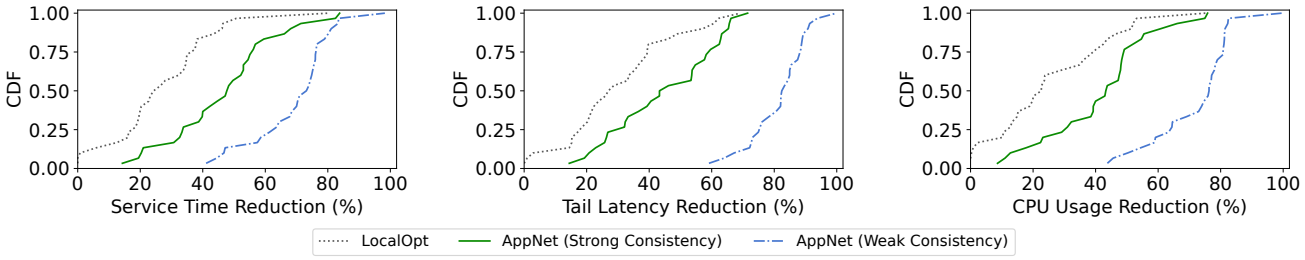


Figure 7: Reduction in RPC processing overhead compared to NoOpt. There are 5 elements in each chain.

AppNet, even when enforcing strong consistency, performs substantially better, with median reductions of 47% in service time, 44% in tail latency, and 42% in CPU usage. With weak consistency, AppNet further reduces overhead. Median reduction is 74-83%.

The degree of overhead reduction varies significantly across chains, influenced by which elements are present and location, platform constraints. By looking at the underlying data, we draw the following conclusions: the presence and placement of shared state elements within the chosen chain determine the potential for reducing synchronization overhead. The greatest reductions occur when multiple shared state elements are present and can be co-located and combined. Chains that allow more flexible element placement tend to yield higher gains. In contrast, chains with strict placement constraints limit the ability to group elements on one side and bypass a platform (e.g., a sidecar) on the other. Noticeable gains are also observed when elements that drop RPCs are positioned early in the chain, enabling them to execute sooner. AppNet's optimization algorithm is able to sift through these opportunities and find performant configurations. In contrast, LocalOpt uses a simplistic approach—selecting platforms based on processing costs and aligning state dependencies—and misses cross-element and cross-platform optimization opportunities.

6.2.2 Impact of available optimization opportunities

To shed light on overhead reduction as a function of available optimization opportunities, we now consider deployment environments that are limited to a single platform (but still multiple locations) or have shorter chains. These experiments are interesting also because some organization may not employ multiple platforms. The optimization opportunities that AppNet can tap in each platform differ. RPC libraries, which can only be used with fully trusted applications, are always in the path and have low overhead. So, they only benefit from reordering and moving elements between clients and servers (e.g., based on state synchronization costs). EnvoyNative has additional benefits when it can be completely bypassed (by moving all elements to one location). EnvoyWasm, which is used when dynamic upgrades are preferred, additionally benefits from consolidating multiple elements into one module because that reduces the number of invocations of the WebAssembly VM.

Figure 8 plots the overhead reduction for all three single-platform environment and the all-platforms environment. The experimental methodology is similar to the last section except that there are no constraints related to application trust and dynamic upgrades. In addition to 5-element chains, it plots 3-element chains which offer fewer optimization opportunities.

For 5-element chains, the median overhead reduction is 3–48% for the median, p10 is 0–24%, and p90 is 27–76%. The extent of the reduction varies across platforms due to the distinct characteristics of each as noted above. Even when reordering and client-vs-server placement are the only optimization opportunities (as in gRPC), we see a noticeable overhead reduction, especially for tail latency and CPU usage. AppNet significantly reduces overhead even for 3-element chains,though the reduction tends to be lower than for 5-element chains.

The reduction is generally higher as optimization opportunities increase. It is highest for the all-platforms environment because that also opens up the opportunity of using the platform with the least overhead when permitted by user policies. Due to implementation-level differences (Go vs. Rust), the state synchronization cost in gRPC is higher than in Envoy, which explains why, in some cases, gRPC optimizations are greater than those of Envoy.

LocalOpt often provides little to no benefit for single-platform environments. Here, it primarily aligns state dependencies, which helps only when an element is unconstrained and NoOpt (randomly) picks suboptimal placement. Such occurrences are rare.

6.2.3 Impact of weak state and observation consistency

Figure 9 shows the impact of using weak state consistency and weak observation consistency separately. It plots the reduction in overhead relative to strong consistency for both. For each curve, the set of chains is limited to those where that consistency type is relevant, i.e., there is at least one shared state variable for state consistency and at least one auxiliary output for observation consistency.

We see that weak state consistency leads to substantial reductions across a variety of chains, with median reduction

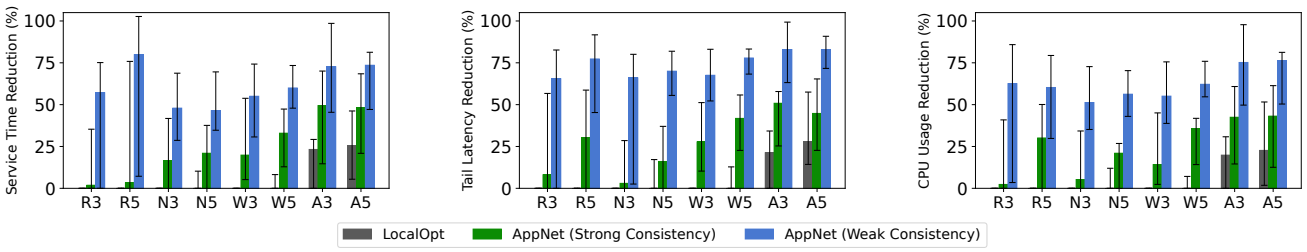


Figure 8: Reduction in RPC processing overhead compared to NoOpt. In the bar labels, the letter denotes available platforms (R = gRPC, N = EnvoyNative, W = EnvoyWasm, and A = all platforms), and the number denotes the chain length. The bar height denotes the median reduction and the error bars denote the 10th and 90th percentiles.

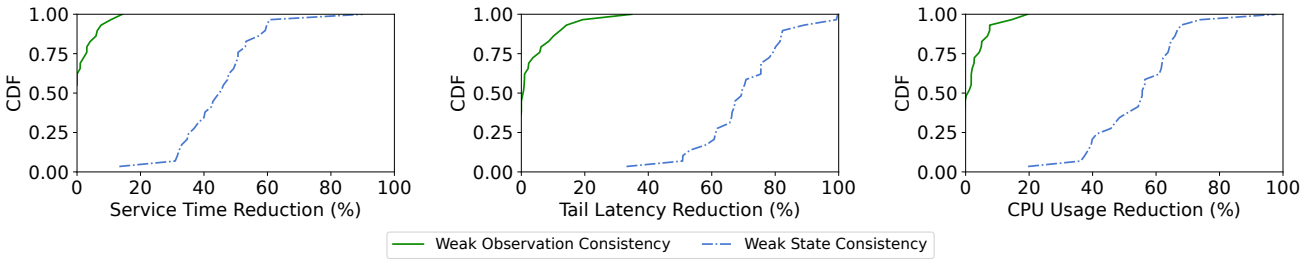


Figure 9: RPC processing overhead reduction with the two types of weak consistency relative to strong consistency.

ranging from 45% to 69%. This reduction stems from making state synchronization a background task. The degree of reduction depends on the number of shared-state elements included and whether they can be consolidated based on their placement constraints.

The overhead reduction from weak observation consistency is lower. A primary source of gain from weak observation consistency is moving a dropping element upstream, which reduces downstream work. But when the drop rate is low, as in our experiments, the gain is small. The gain will be higher in scenarios with higher drop rates (e.g., a rate limiter under heavy load). Despite low drop rates, we see a reduction in service time, tail latency, and CPU usage in 38-55% of the cases. Much of this reduction stems from weak observation enabling AppNet to reorder and consolidate more elements.

6.3 Application performance

We now evaluate the extent to which the reduction in RPC processing overhead in AppNet translates to end-to-end improvement in application-level performance. This section studies the Hotel Reservation application [53] (Figure 1), and Appendix C studies the Online Boutique application [36]. We use Hotel Reservation's search query, which retrieves available hotels based on location and check-in/check-out dates and involves calls to the Frontend, Search, Rates, Geo, Profile, Reserve microservices, and their associated MongoDB and Memcached storage systems. For each communication edge between microservices, we generate 5-element chains randomly (as in § 6.2.1). We measure end-to-end latency and total CPU.

Figure 10 shows the reduction in latency and CPU compared to NoOpt. The baseline metrics (not shown) are 7.2–14.1 ms for service time, 35.6–92.4 ms for tail latency, and 136.4-219.3 virtual cores for CPU usage.

LocalOpt demonstrates moderate performance gains across all metrics and chain configurations, with median reductions of 14% in service time and 11% in both tail latency and CPU usage. AppNet achieves more substantial improvements: with strong consistency, median reductions are 35% for service time, 29% for tail latency, and 26% for CPU usage. Weak consistency improves these numbers to 49%, 41%, and 42%, respectively.

Our results show that reducing RPC processing overhead leads to substantial improvements in end-to-end application performance. This is because this overhead is a substantial contributor toward application latency and CPU usage [5, 66, 71, 86]. Lowering this overhead directly helps applications by alleviating the tax of the microservices architecture.

7 Related Work

AppNet draws on four themes of prior work.

High-level network programming We draw heavily from the large body of work of high-level network programming, which has been explored in several contexts. Early work such as Click [60] focused on packet-processing in software switches. We borrow from this domain the idea of expressing

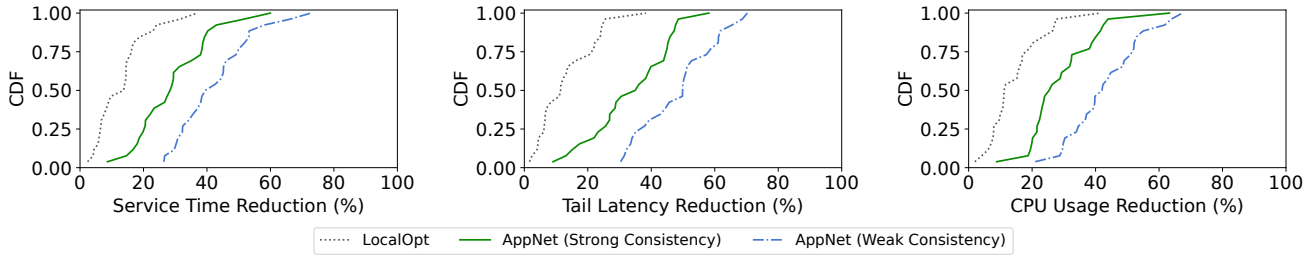


Figure 10: Performance improvement of Hotel Reservation compared to the NoOpt.

network functionality as a chain of elements. A key difference between AppNet and these systems is that AppNet includes abstractions for processing *inside* an element, which allows it to optimize the chain end-to-end.

A second wave focused on network-wide packet forwarding, catalyzed by software-defined networking (SDN) [43, 52, 68], where the focus was on layer-3/4 packet forwarding with capabilities like determining next hops and filtering. In contrast, AppNet targets rich, stateful functions and a setting where the network can scale dynamically.

The advent of programmable switches engendered work on high-level programming for them [46, 54, 75]. We borrow the match-action primitive from this work. Our match-action rules are richer, corresponding to the need for rich, layer-7 processing and aided by not being bound by hardware constraints. AppNet abstractions include shared state handling and optimizations across elements in a chain.

A parallel line of work focuses on NFs (network functions) in middleboxes. NetBricks [63] improves performance by composing multiple functions using a safe runtime, so they do not have to use separate VMs. It does not provide abstractions for individual functions, which allow us to safely combine multiple functions. Rubik [62] provides a language for efficient, low-level packet handling (e.g., assembling IP fragments and reconstructing TCP streams). AppNet abstractions focus on the RPC layer.

Reducing the overhead of application networks. There is broad awareness in the industry about the overhead of application networks [2, 5]. Several projects [13, 17] target this challenge by changing the software design of service meshes. Meta built ServiceRouter [70] to lower application networking overhead for specific functions, such as routing requests based on applications' sharding keys, and Alibaba built Canal Mesh [76] to address the overhead of sidecars by moving processing to a remote proxy. Unlike AppNet, none of these works focus on high-level programming to support diverse applications, deployment modes, and capabilities.

NF Optimizations. Optimizing NF chains for performance and resource efficiency has been an active area of research [48, 58, 64, 72, 78]. OpenBox [48] reduces redundancy in NF chains by eliminating overlapping logic across functions. Metron [58] builds on this by offloading parts of the merged logic to programmable switches. NFP [78] and Maestro [64] leverage parallelization to accelerate both inter- and intra-NF processing. In general, these systems focus on processing lower-level protocols and raw network packets, which have goals distinct from those of AppNet. Additionally, traditional NFs typically operate within dedicated middleboxes, while AppNet targets RPC processing tasks that can run in diverse environments, including RPC libraries, sidecar proxies, or middlebox-style remote proxies.

State management in NFs Works on scaling NFs show that managing state is a central challenge. Split/Merge [67], S6 [81], StatelessNF [57], SwiSh [84] manage different types of state based on its properties, and OpenNF [55] develops techniques to (approximately) infer properties of state in existing NFs. While these works do not focus on high-level message processing abstractions, they have influenced our design—our primitives enable a compiler to automatically infer important properties of state, and our state handling (placement and partitioning) draws on lessons from them.

8 Conclusions

AppNet enables expressive and high-performance application networks by decoupling the specification of ANF from its implementation. Its specification language is easy to use—common ANFs can be expressed in only 7–28 lines of code. It generates high-performance implementations based on an approach that reasons about estimated performance and semantic equivalence of possible implementations. Our experiments show that this approach lowers RPC processing latency and CPU usage by, respectively, 82% and 75%.

Acknowledgements

We thank the NSDI reviewers and our shepherd, Jiaqi Gao, for their feedback. This work was supported in part by UW FOCI and its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware), by NSF Grant 2402695 and 2402696, and by ACE, a center that is part of DARPA's JUMP 2.0.

References

[1] Enabling GZIP Response Compression with EnvoyFilter. https://karlstoney.com/enabling-gzip-compression-with-envoyfilter/, 2019.

[2] Performance Benchmark Analysis of Istio and Linkerd. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/, 2019.

[3] Benchmarking Linkerd and Istio: 2021 Redux. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/, 2021.

[4] Embracing eventual consistency in SoA networking. https://blog.envoyproxy.io/embracing-eventual-consistency-in-soa-networking-32a5ee5d443d, 2022.

[5] Istio: Performance and Scalability. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/, 2022.

[6] Performance Impacts of an Istio Service Mesh. https://pklinker.medium.com/performance-impacts-of-an-istio-service-mesh-63957a0000b, 2022.

[7] Service meshes are on the rise — but greater understanding and experience are required. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf, 2022.

[8] Understanding Istio Ambient Ztunnel and Secure Overlay. https://www.solo.io/blog/understanding-istio-ambient-ztunnel-and-secure-overlay, 2022.

[9] Use Dataflow SQL. https://cloud.google.com/dataflow/docs/guides/sql/dataflow-sql-intro, 2022.

[10] Better Load Balancing: Real-Time Dynamic Subsetting. https://www.uber.com/en-US/blog/better-load-balancing-real-time-dynamic-subsetting/, 2023.

[11] Go gRPC Middleware. https://github.com/grpc-ecosystem/go-grpc-middleware, 2023.

[12] gRPC Encryption. https://github.com/grpc/grpc-go/blob/master/examples/features/encryption/README.md, 2023.

[13] Introducing Ambient Mesh. https://istio.io/v1.15/blog/2022/introducing-ambient-mesh/, 2023.

[14] Linkerd Features. https://linkerd.io/2.14/features/, 2023.

[15] OWASP Coraza Web Application Firewall. https://coraza.io/, 2023.

[16] AppNet: Expressing, easy-to-build, and high-performance application networks. https://appnet.wiki/, 2024.

[17] Cilium Service Mesh. https://cilium.io/use-cases/service-mesh/, 2024.

[18] Cloud Service Mesh with proxyless gRPC. https://cloud.google.com/service-mesh/docs/service-routing/proxyless-overview, 2024.

[19] Envoy. https://www.envoyproxy.io/, 2024.

[20] Envoy: Application logging. https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/application_logging, 2024.

[21] Envoy: Bandwidth limiting. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/bandwidth_limiting, 2024.

[22] Envoy: Cache filter. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/cache_filter, 2024.

[23] Envoy: Fault Injection. https://istio.io/latest/docs/tasks/traffic-management/fault-injection/, 2024.

[24] Envoy: Global rate limiting. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/global_rate_limiting, 2024.

[25] Envoy: HTTP Routing. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/http/http_routing, 2024.

[26] Envoy: Load Balancing. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/load_balancing, 2024.

[27] Envoy: Rate limit. https://github.com/envoyproxy/ratelimit, 2024.

[28] Envoy Statistics. https://istio.io/latest/docs/ops/configuration/telemetry/envoy-stats/, 2024.

[29] gRPC Interceptors. https://grpc.io/docs/guides/interceptors/, 2024.

[30] gRPC-tools. https://github.com/bradleyjkemp/grpc-tools, 2024.

[31] HTTP filters. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters, 2024.

[32] Istio: Circuit Breaking. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/, 2024.

[33] Kubernetes. https://kubernetes.io/, 2024.

[34] Kubernetes: Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, 2024.

[35] Linkerd: the world's most advanced service mesh. https://linkerd.io/, 2024.

[36] Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo, 2024.

[37] Protocol Buffers. https://protobuf.dev/, 2024.

[38] Redis. https://redis.io/, 2024.

[39] The Istio Service Mesh. https://istio.io/, 2024.

[40] WebAssembly in Envoy. https://github.com/proxy-wasm/spec/blob/master/docs/WebAssembly-in-Envoy.md, 2024.

[41] wrk. https://github.com/wg/wrk, 2024.

[42] wrk2. https://github.com/giltene/wrk2, 2024.

[43] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. *ACM SIGPLAN notices*, 49(1):113–126, 2014.

[44] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 29–43, 2016.

[45] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. O'Reilly Media, Inc., 2016.

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

[47] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[48] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 511–524, 2016.

[49] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[50] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[51] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 1–14, 2019.

[52] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291, 2011.

[53] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[54] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 435–450, 2020.

[55] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid,

Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 44(4):163–174, 2014.

[56] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (ATC 23)*, pages 419–432, 2023.

[57] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, 2017.

[58] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron:NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, 2018.

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

[60] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[61] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. LemonNFV: Consolidating heterogeneous network functions at line speed. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1451–1468, 2023.

[62] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming network stack for middleboxes with rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 551–570, 2021.

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

[64] Francisco Pereira, Fernando MV Ramos, and Luis Pedrosa. Automatic parallelization of software network functions. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1531–1550, 2024.

[65] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, 2019.

[66] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 780–794, 2022.

[67] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, 2013.

[68] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN programming with pyretic. *Technical Report of USENIX*, 2013.

[69] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.

[70] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[71] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[72] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336, 2012.

[73] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types.

In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[74] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 15–28, 2016.

[75] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the ACM SIGCOMM 2021 Conference*, pages 731–747, 2021.

[76] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, et al. Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 860–875, 2024.

[77] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 65–76, 2014.

[78] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling network function parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 43–56, 2017.

[79] Wlamir Olivares Loesch Vianna, Leonardo Ramos Rodrigues, Takashi Yoneyama, and David Issa Mattos. Troubleshooting optimization using multi-start simulated annealing. In *2016 Annual IEEE Systems Conference (SysCon)*, pages 1–6. IEEE, 2016.

[80] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. *ACM SIGCOMM Computer Communication Review*, 43(4):87–98, 2013.

[81] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, 2018.

[82] Bartek Wydrowski, Robert Kleinberg, Stephen M Rumble, and Aaron Archer. Load is not what you should balance: Introducing Prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1285–1299, 2024.

[83] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, 2022.

[84] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

[85] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[86] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, et al. Dissecting Overheads of Service Mesh Sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 142–157, 2023.

A Additional AppNet Design

A.1 Code generation

The AppNet compiler takes in the optimized runtime configuration and produces platform-specific software modules as well as deployment scripts. To handle requests for strongly consistent state, it translates these accesses into blocking network calls to external storage. In contrast, reads from weakly consistent state are performed locally, and writes are applied asynchronously. The compiler further reduces overhead by consolidating co-located elements, effectively removing boundaries introduced by virtualization or sandbox approaches [61, 63]. This consolidation applies to both the processing logic and state synchronization messages.

The generated deployment scripts configure traffic flow for each communication edge (microservices pair) such that sidecars are bypassed when the runtime configuration does not place any element there. (Today, all traffic of a microservice is intercepted by the sidecar, which imposes unnecessary overhead for RPC traffic that does not require processing.)

A.2 Consistent Updates

Application networks evolve frequently due to policy and workload changes (scaling). Service meshes today provide atomic updates for individual ANF, but ensuring consistency across a communication edge with multiple ANFs is left to developers. This approach is vulnerable to application downtime or policy violations [4]. For example, when migrating an application firewall, the developer may first detach the module on one side and then attach it on the other, potentially causing policy violations.

AppNet ensures consistent configuration updates such that RPCs traversing a communication edge are processed entirely by the old configuration or the new, and never a blend of the two. It uses a two-phase update approach [69, 83]. Each RPC is tagged with a version number, and configurations are updated to process only RPCs with the corresponding version. To transition from one configuration to the next, AppNet controller first installs the new configuration, guarded by the next version number. Once that is done, it enables the new configuration by stamping new RPCs with the next version number. Finally, once all RPCs with the old version number have completed, the old configuration is removed.

The header to support consistent upgrades adds 23 bytes to RPCs.

B Equivalence Checking Formalization

This appendix describes equivalence checking of AppNet in more detail. Table 3 lists all types of symbols with their explanations.

Symbolic abstraction of an element We symbolically abstract an element as a set of transfer functions in two steps. In the first step, we convert the element into a tuple of properties (`rf`, `wf`, `rs`, `ws`, `drop`, `reorder`, `channel`), where

- `rf`(read fields), `wf`(write fields), `rs`(read states), `ws`(write states) are sets of RPC field/state names that the element reads/modifies.
- `drop`, `reorder` are boolean values indicating whether the element is likely to drop RPCs or reorder input RPC stream. An element may reorder the RPC stream if it does concurrent processes or delays RPCs by differing amounts.
- `channel` is a set of auxiliary output channels in the element. Elements such as Logging and Metrics will send RPC records to these channels.

All of these properties are obtained by analyzing the (post-parsing) IR of AppNet specifications.

The second step converts these properties into transfer functions. Algorithm 1 illustrates this process. **wf**, **ws**, **ord** and **dr** in the transfer functions are opaque function symbols representing the dependencies of field modification, state modification, reordering decision, and dropping decision, respectively. The meaning of symbol subscripts are explained in Table 3.

Algorithm 1 Symbolic Abstraction Algorithm

Input: *AppNet specification of an element*
Output: *A set of transfer functions for the element*

Function `SymbolicAbstraction`(e)**:**
  (`rf`, `wf`, `rs`, `ws`, `drop`, `reorder`, `channel`) ← get_property_tuple(e)
  $TFs \leftarrow \{\}$ # TFs is a set for collecting transfer functions
  $rv \leftarrow \{\tilde{r}.f : f \in \texttt{rf}\} \cup \{\tilde{s}.v : v \in \texttt{rs}\}$
  foreach $f \in$ `wf` **do**
    TFs.add($\tilde{r}.f \leftarrow \mathbf{wf}_{(\mathbf{i},\mathbf{f})}(rv)$)
  end
  foreach $v \in$ `ws` **do**
    TFs.add($\tilde{s}.v \leftarrow \mathbf{ws}_{(\mathbf{i},\mathbf{v})}(rv \cup \tilde{r}.drops \cup \tilde{r}.reorders)$)
  end
  foreach $ch \in$ `channel` **do**
    TFs.add($ch \leftarrow ch \cup \{\tilde{r}\}$)
  end
  if `reorder` = **true then**
    TFs.add($\tilde{r}.reorders \leftarrow \tilde{r}.reorders \cup \{\mathbf{ord}_{\mathbf{i}}(rv)\}$)
  end
  if `drop` = **true then**
    TFs.add($\tilde{r}.drops \leftarrow \tilde{r}.drops \cup \{\mathbf{dr}_{\mathbf{i}}(rv)\}$)
  end
  return TFs

Symbolic execution of a chain of elements Algorithm 2 shows the process of sequentially iterating over the chain and concatenating element-level transfer functions to obtain chain-level transfer functions. $\mathbf{F}_{\mathbf{f_x}}$ and $\mathbf{S}_{\mathbf{v_x}}$ are primitive symbols representing the initial values of RPC fields and state variables.

Comparing two chains Algorithm 3 shows how we compare two chains under desired consistency level. For weak consistency, we require the transfer functions of output RPC and final states to be identical. For strong consistency, we also compare the transfer functions of auxiliary channels.

Correctness Proof Two chains are equivalent if and only if for any input RPC stream, the contents of output RPCs, the outputs of auxiliary channels, and all RPC-related behaviors such as dropping, are identical. The only differences permitted are those that could occur between two copies of the same chain, such as the differences caused by randomness. In our formalization, we assume that all opaque functions additionally take an implicit seed that makes the function deterministic, and the same seed is used for both chains under comparison to eliminate randomness. We also assume that the RPC stream is not reordered (e.g., during network transmission—by explicitly including the network as a reordering element, we can remove this assumption) unless it

Type	Explanation	Examples
Properties	obtained from AppNet specifications, used for symbolic abstraction	`rf` (read fields, a set of field names), `ws` (write states, a set of state variable names), `channel`
Primitive Symbols	symbols representing initial values of fields and states	$\mathbf{F_f}$ (the initial value of $RPC.f$), $\mathbf{S_v}$ (the initial value of state variable v)
Function Symbols	symbols representing the opaque internal logic of elements, the parameters of which are symbolic values that it depends on	$\mathbf{wf_{(i,f)}}(S)$ (the logic of the i_{th} element writing to $RPC.f$, where S is a set of symbols the element reads)
Dictionary	A dictionary maps field name/variable name to its value symbol. If $a : b \in \tilde{s}$, then $\tilde{s}.a \triangleq b$.	$\tilde{r} = \{f_1 : \mathbf{F_{f_1}}, \cdots, f_n : \mathbf{F_{f_n}}, drops : \{\}, reorders : \{\}\}$.

Table 3: Symbol Explanation and Examples

Algorithm 2 Symbolic Execution Algorithm

Input: *a chain configuration $c = (e_1, \cdots, e_n)$, i.e., a list of AppNet specifications*
Output: *chain-level transfer functions of output RPC, states and auxiliary channels*

Function `SymbolicExecution`*(c)*:
  chainTFs $\leftarrow \{$
  $\tilde{r} \leftarrow (f_1 : \mathbf{F_{f_1}}, \cdots, f_n : \mathbf{F_{f_n}}, drops : \{\}, reorders : \{\})$,
  $\tilde{s} \leftarrow (v_1 : \mathbf{S_{v_1}}, \cdots, v_m : \mathbf{S_{v_m}})$
  $\}$
  foreach $e_i \in c$ **do**
    TFs $\leftarrow$ `SymbolicAbstraction`(e_i)
    chainTFs $\leftarrow$ concatenate(*chainTFs*, *TFs*)
  end
  rpcTFs, stateTFs, channelTFs $\leftarrow$ split(ChainTFs)
  return *rpcTFs, stateTFs, channelTFs*

Algorithm 3 Equivalence Checking

Input: *Two chain configurations c_1, c_2 and desired consistency level L*
Output: *A boolean indicating whether c_1 and c_2 are equivalent under L*

$rpc_1, state_1, channel_1 \leftarrow$ `SymbolicExecution`(c_1)
$rpc_2, state_2, channel_2 \leftarrow$ `SymbolicExecution`(c_2)
if L = *'weak'* **then**
  return $rpc_1 = rpc_2$ **and** $state_1 = state_2$
else
  return $rpc_1 = rpc_2$ **and** $state_1 = state_2$ **and** $channel_1 = channel_2$
end

meets elements with property $reorder =$ **true**.

We need to prove that for two chains, if the algorithm produces the same transfer functions, then the two configurations are equivalent. The proof below focuses on weak consistency, i.e., analyzing the output RPC and states; strong consistency just additionally checks the auxiliary channels, which follows a similar approach.

We start by showing that if two symbolic transfer functions are identical, then the concrete value or dropping/reordering decisions they represent are equivalent. We prove the statement by structural induction.

- Base case: primitive symbols include $\mathbf{F_{f_1}}, \cdots, \mathbf{F_{f_n}}$ and $\mathbf{S_{v_1}}, \cdots, \mathbf{S_{v_m}}$, which represent initial values of RPC fields and state variables. Identical primitive symbols indicate that they represent the exact same concrete initial value, which are equivalent.
- Inductive case: our symbolic execution system has two kinds of compound symbol structures:
  - $\mathbf{wf/dr/ord_{(i,f)}}(\{\tilde{r}.f : f \in rf\} \cup \{\tilde{s}.v : v \in rs\})$: these three symbols represent the modified field value that the i_{th} element writes into the RPC, the dropping decision, and the reordering decision, respectively. Considering that the essence of RPC processing inside an ANF is a black-box function that maps input RPCs and states to output RPCs and RPC-related operations, if all inner symbols in the parenthesis are identical, which means all the RPC fields and states the element reads are equivalent, then the modified fields or dropping/reordering decisions are also equivalent.
  - $\mathbf{ws_{(i,v)}}(\{\tilde{r}.f : f \in rf\} \cup \{\tilde{s}.v : v \in rs\} \cup RPC.drops \cup RPC.reorders)$: this symbol represents the modified value of state variables. State updates are different from RPC processing in that not only RPC contents and previous state values matter, but also whether an RPC arrives at the element and the order of RPC stream will influence the result. Therefore, our algorithm additionally includes the set of previous dropping/reordering decisions into the parentheses, and if all inner symbols are identical, then the modified state value should also be equivalent.

Therefore, if the transfer functions of output RPC are identical, then equivalence of single input RPC could be guaranteed.

The next step is to show why it is correct to apply analysis

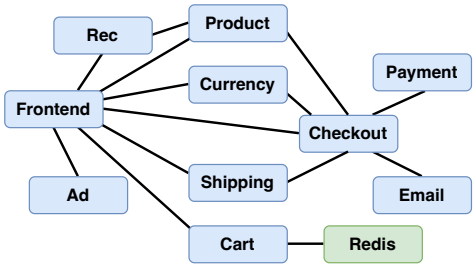


Figure 11: Online Boutique Microservice [36].

on single input RPC to stateful chains that receive a stream of RPCs. Let's consider two scenarios:

- The chain is stateless, i.e., there is no stateful element inside the chain. In this case, RPCs will not interfere with each other, and therefore reasoning about a stream of input RPCs is equivalent to reasoning about each RPC independently.

- The chain contains stateful elements. Considering that (1) The reordering events symbols inside the transfer functions of states are identical, which guarantees that stateful elements see RPCs in the same order in two configurations, and (2) The transfer functions of states, in a whole, are identical, which means given the same pre-states, the n_{th} RPC will produce the same post-states which serve as the pre-states of the $(n+1)_{th}$ RPC, we can inductively conclude that all RPCs in the stream will result in identical symbolic output RPCs.

Therefore, we can conclude that if the algorithm produces identical symbolic output RPC, dropping events and modified states, two chain configurations are equivalent.

C Additional Application Benchmark

In this section, we demonstrate AppNet's ability to reduce RPC processing overhead using the Online Boutique application [36]. It consists of 11 microservices that communicate via gRPC (Figure 11). Since it is implemented in multiple languages (Python, C#, Java, Node.js and Go) and AppNet supports only the Go version of gRPC, we port all services to Go. Our experimental methodology follows the same approach as in § 6.3.

Figure 12 presents the results, showing the performance improvements of the LocalOpt and AppNet relative to the "Fully Pinned" baseline. The baseline metrics (not shown) across all chain configurations range from 14.1–19.2 milliseconds for service time, 90.1–162.2 milliseconds for tail latency, and 106.4–163.8 virtual cores for CPU usage.

Consistent with the results from the Hotel Reservation benchmark, the LocalOpt achieves moderate performance gains across all metrics and chain configurations, with median reductions of 8% in service time and tail latency and 9% in CPU usage. In contrast, AppNet delivers greater improvements: under strong consistency, median reductions reach 22% for service time, 20% for tail latency, and 24% for CPU usage. When weak consistency is applied, these gains increase further to 36%, 37%, and 44%, respectively.

The overall improvements achieved by both the LocalOpt and AppNet are smaller in the Online Boutique application compared to the Hotel Reservation benchmark. This is primarily due to the larger baseline application latency and CPU usage, which reduce the relative impact of RPC processing optimizations. Nevertheless, AppNet still provide substantial performance benefits, demonstrating its effectiveness across different application workloads.

D Abstraction Tax

Systems based on high-level languages sometimes pay a performance tax when the generated code is not as efficient as hand-written code. We quantify this tax for AppNet by running our generated modules against two sets of handwritten modules. Each module is run in isolation (i.e., not as part of a chain), and we measure the same three metrics.

The first set of handwritten modules is Envoy's built-in filters. Envoy has a set of built-in filters written in C++ that users can configure to use with Envoy. Versions of all 12 filters in Table 2 are available except for encryption. Figure 13a shows how our generated EnvoyNative modules compare to these filters. We see that the tax is low. The median increase in latency and CPU is 1-4%, with the worst-case overhead being 3-7%.

The second set of handwritten modules are gRPC interceptors and Envoy WebAssembly filters that we wrote (because we were not able to find usable third-party implementations. Figure 13b presents the results. We see a similar tax to what we saw for Envoy built-in filters.

We investigated the sources of this tax and found that significant contributors include coarse-grained locking of shared state and the data structures used to maintain metadata for matching RPC messages across processing functions. We speculate that this can be reduced with a more careful analysis of the processing logic. Nevertheless, as demonstrated earlier, this low tax is more than compensated for by the optimizations enabled by AppNet's abstractions.

E Details of Optimization Algorithm

As the number of elements in the chain increases, the search space for potential optimal configurations soon becomes large and intractable. One of the possible approaches is to formulate the optimization into an integer programming (IP) problem and invoke a solver to generate solutions. However, solver-based methods are not scalable enough and struggle to incorporate validity checks as IP constraints. To address these challenges, AppNet leverages a multi-start simulated annealing (MSA) algorithm [79] to efficiently find high-quality solutions for long chains. Specifically, during each cooling

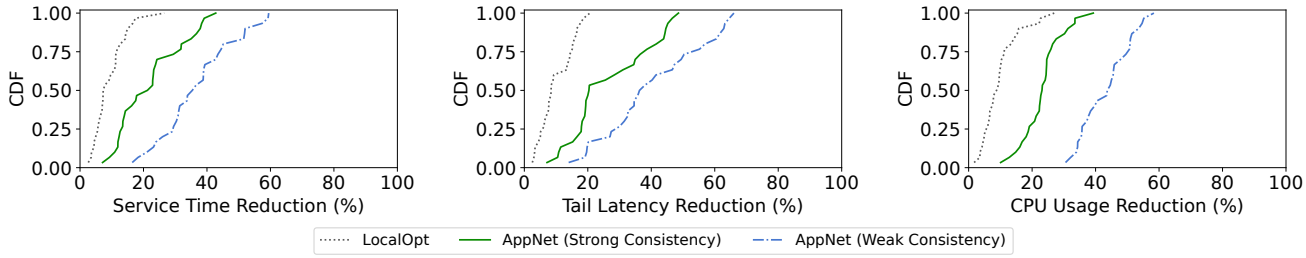
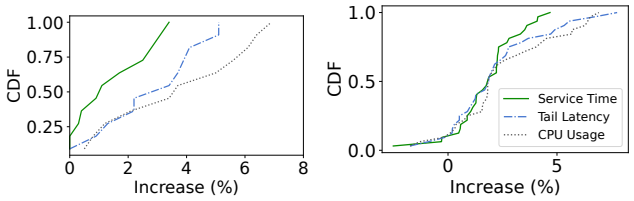
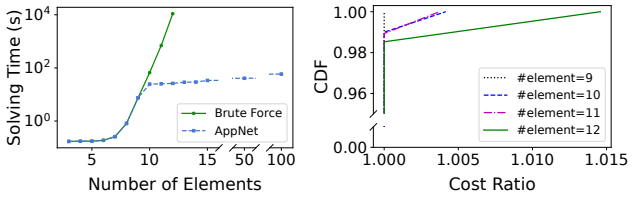


Figure 12: Performance improvement of Online Boutique compared to NoOpt.



(a) Envoy built-in. (b) Hand-coded.

Figure 13: Performance of AppNet generated elements compared to Envoy built-in and hand-coded elements.



(a) Solving time comparison (b) Cost ratio CDF

Figure 14: Efficiency and optimality comparison between multi-start simulated annealing and brute-force searching.

process, we iteratively consider valid (i.e., semantics preserving) mutations of the current solution and decide whether to adopt the new configuration according to some probabilities related to the cost difference and dropping temperature. Algorithm 4 shows our strategy of configuration mutation. To further mitigate the risk of local optima, we repeatedly restart the cooling process by resetting the temperature. The algorithm terminates when there's no improvement in cost over the last 15 cooling processes or the time budget runs out. Our final optimization algorithm combines MSA with brute-force searching: for chains with fewer than 10 elements, we exhaustively enumerate all possible combinations of configurations to guarantee optimality; for longer chains, we apply the MSA heuristics to achieve near-optimal solutions efficiently.

To evaluate the efficiency and solution quality of our approach, we compare multi-start simulated annealing against brute-force searching by randomly generating 100 specifications for each chain size ranging from 3 to 12 and recording the solving time and final cost. Figure 14a shows the speed

Algorithm 4 Solution Mutator in Simulated Annealing

Input: *The current solution*
Output: *A new valid solution mutated from the original one*

Function `SolMutation(s)`**:**
  repeat
    $s' \leftarrow s$
    switch $coin \leftarrow random(0,1)$ **do**
      case $0 \leq coin < 0.4$:
        change the position of one element in s'
      case $0.4 \leq coin < 0.8$:
        swap the positions of two elements in s'
      case $0.8 \leq coin \leq 1$:
        change the platform of one element in s'
    end
  until $PassValidityCheck(s')$;
  return s'

comparison. Brute-force searching becomes computationally prohibitive due to its factorial complexity, whereas MSA is 473x faster for 12-element chains and remains practical even when the chain size increases to 100. Figure 14b presents the CDF of the cost of multi-start simulated annealing relative to brute-force searching. MSA successfully identifies the optimal configuration in more than 98% of cases when the chain size is no more than 12, and the worst-case cost increase is lower than 1.5%.

F Details of ServiceRouter and Prequal Implementation

In this section, we describe the implementation details of ServiceRouter and Prequal in AppNet.

ServiceRouter [70] is Meta's global service mesh system. We implement the routing and load balancing as described in the paper. Specifically, it routes RPCs based on their key (for sharded services) and uses adaptive load estimation along with the power of two choices to balance load across service replicas. The AppNet implementation consists of a control plane component, a client-side element, and a server-side element. The control plane component maintains the shard information for each service and maintains an up-to-date global knowledge about the load of each service replica. The client-side

element maintains two states: a shard map and a load map, which store the mapping between request keys and shard replicas, and replicas to their current load, respectively. For each request, the client-side element first checks the freshness of its local states, queries the control plane if they are stale, and applies the power-of-two choices on the corresponding shard replicas. Upon receiving a response, the client-side element updates the load map using the piggybacked load information embedded in the response. The server-side element monitors load locally by maintaining a counter. This counter is incremented when a request is received and decremented when the application sends a response. The current load value is embedded as an RPC header in outgoing responses

Prequal [82] is a load balancer used by YouTube and other production systems at Google. It selects server replicas based on estimated latency and active in-flight requests. The AppNet implementation of Prequal also comprises three main components: a control plane, a client-side element, and a server-side element. The control plane actively probes and stores the load of each replica along with its request latency for various load levels. The client maintains a local cache of the control plane's state. For each request, it applies the power-of-N choices to sample a subset of replicas and retrieves their load information from either the local cache or the control plane, depending on the cache's freshness. It then uses the hot-cold lexicographic rule, as described in the paper, to select the replica for routing the request. The server-side component monitors the current load and maintains a mapping between load levels and the latencies of requests processed at each load point.

State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing

Qiongwen Xu[△], *Sebastiano Miano*[⊕], *Xiangyu Gao*[‡], *Tao Wang*[‡], *Adithya Murugadass*[△], *Songyuan Zhang*[△], *Anirudh Sivaraman*[‡], *Gianni Antichi* [Ω ⊕], *Srinivas Narayana*[△]

[△] *Rutgers University, USA* [⊕] *Politecnico di Milano, Italy* [‡] *New York University, USA* [Ω] *Queen Mary University of London, UK*

Abstract

With the slowdown of Moore's law, CPU-oriented packet processing in software will be significantly outpaced by emerging line speeds of network interface cards (NICs). Single-core packet-processing throughput has saturated.

We consider the problem of high-speed packet processing with multiple CPU cores. The key challenge is state—memory that multiple packets must read and update. The prevailing method to scale throughput with multiple cores involves state sharding, processing all packets that update the same state, e.g., flow, at the same core. However, given the skewed nature of realistic flow size distributions, this method is untenable, since total throughput is limited by single-core performance.

This paper introduces *state-compute replication*, a principle to scale the throughput of a single stateful flow across multiple cores using replication. Our design leverages a *packet history sequencer* running on a NIC or top-of-the-rack switch to enable multiple cores to update state without explicit synchronization. Our experiments with realistic data center and wide-area Internet traces show that state-compute replication can scale total packet-processing throughput linearly with cores, independent of flow size distributions, across a range of realistic packet-processing programs.

1 Introduction

Designing software to handle high packet-processing loads is crucial in networked systems. For example, software load balancers, CDN nodes, DDoS mitigators, and many other middleboxes depend on it. Yet, with the slowdown of Moore's law, software packet processing has struggled to keep up with line speeds of network interface cards (NICs), with emerging speeds of 200 Gbit/s and beyond [20]. Consequently, there have been significant efforts to speed up packet processing through better network stack design, removing user-kernel crossings, running software at lower layers of the stack (*e.g.,* NIC device driver), and designing better host interconnects.

We consider the problem of scaling software packet processing by using multiple cores on a server. The key challenge is that many packet-processing applications are *stateful*, maintaining and updating regions of memory across many packets. If multiple cores contend to access the same memory regions, there is significant memory contention and cache bouncing, resulting in poor performance. Hence, the classic approach to multicore scaling is to process packets touching distinct states, *i.e.,* flows, on different cores, hence removing memory contention and synchronization. For example, a load balancer that maintains a separate backend server for each 5-tuple may send all packets of a given 5-tuple to a fixed core, but process different 5-tuples on different cores, hence scaling performance with multiple cores. Many prior efforts have implemented and optimized such sharding-oriented solutions [35,46,54,63,68].

However, we believe that the existing approaches to multicore scaling have run their course (§2). Realistic traffic workloads have heavy-tailed flow size distributions and are highly skewed. With sharding, large "elephant flows" must be processed by one CPU core, reducing overall performance due to the low throughput of a single CPU core. With emerging 200 Gbit/s—1 Tbit/s NICs, a single packet processing core may be too slow to keep up even with a single elephant flow. Additionally, with the growing scales of volumetric resource exhaustion attacks, packet processors must gracefully handle attacks where adversaries force packets into a single flow [43].

This paper introduces a scaling principle, *state-compute replication (SCR)*, that improves the software packet-processing throughput for a *single, stateful flow* with additional cores, while avoiding shared memory and contention. Figure 1 shows how SCR scales the throughput of a single TCP connection for a TCP connection state tracker [40] with more cores, when other scaling techniques fail. A connection tracker may change its internal state with each packet.

SCR applies to any packet processing program that may be abstracted as a deterministic finite state machine. Intuitively, as long as each core can reconstruct the flow state by processing *all* the packets of the flow, multiple cores can successfully process a single stateful flow with zero cross-core synchronization. However, naively processing every packet with every CPU core cannot improve throughput with more

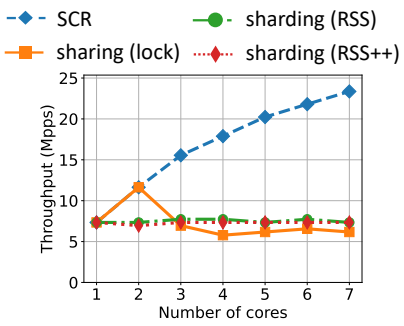


Figure 1: Scaling the throughput of a TCP connection state tracker for a *single TCP connection* across multiple cores. Sharing state across cores degrades performance beyond 2 cores due to contention. Sharding state (using RSS and RSS++ [35]) cannot improve throughput beyond a single CPU core (§2). In contrast, State-Compute Replication (§3) provides linear scale-up in throughput with cores.

cores. We need to meet two additional requirements. First, in systems where CPU usage is dominated by per-packet work, we must preserve the total number of packets moving through the system. Second, we must divide up *dispatch*—the CPU software labor of presenting the packets to the packet-processing program—across multiple cores (§3.1). This leads to the scaling principle below (more formal version in §3.1):
State-Compute Replication (informal). In a system bound by per-packet CPU work and software dispatch, *replicating* the state and the computation of a stateful flow across cores increases throughput *linearly* with cores, so long as we preserve the total number of packets traversing the system.

To meet the requirements for high performance in SCR, we design a *packet history sequencer* (§3.2), an entity which sees every packet and sprays packets across cores in a round-robin fashion, while piggybacking the relevant information from packets missed by a core on the next packet sent to that core. The sequencer must maintain a small bounded history of packet headers relevant to the packet-processing program. A high-speed packet-processing pipeline, running either on a programmable NIC or a top-of-the-rack switch, may serve as a sequencer. We present sequencer designs for two hardware platforms (§3.3), a Tofino switch, and a Verilog RTL module we synthesized into the NetFPGA-PLUS reference pipeline.

We evaluated the scaling efficacy of SCR using a suite of realistic programs and traffic (§4). SCR is the only technique we are aware of that monotonically scales the total processing throughput with additional cores regardless of the skewness in the arriving flow size distribution. However, no technique can continue scaling indefinitely. In §3.1 and §4, we demonstrate the limits of SCR scaling, along with the significant performance benefits to be enjoyed before hitting these limits. Additionally, we show the resource usage for our sequencer's RTL design, which meets timing at 340 MHz. We believe this design is simple and cheap enough to be added as an on-chip accelerator to a future NIC.

All our software is publicly available [31]. This work does not raise any ethical concerns.

2 Background and Motivation

2.1 High Speed Packet Processing

This paper considers packet-processing programs that must work at heavy network loads with quick turnarounds for packets. We are specifically interested in applications implementing a "hairpin" traffic flow: receiving a packet, processing it as quickly as possible, and transmitting the packet right out, at the highest possible rate. Such programs exhibit compute and working set sizes that are smaller in comparison to full-fledged applications (running in user space) or transport protocols (TCP stacks) running at endpoints.

Examples of the kinds of applications we consider include (i) middlebox workloads (network functions), such as firewalls, connection trackers, and intrusion detection/prevention systems; and (ii) high-volume compute-light applications such as key-value stores, telemetry systems, and stream-based logging systems, which process many requests with a small amount of computation per request [51]. A key characteristic of such applications is their need for high packet-processing rate (which is more important than byte-processing rate) and the fact that their performance is primarily bottlenecked by CPU usage [44, 69, 73].

The performance of such applications is mission-critical in many production systems. Even small performance improvements matter in Internet services with large server fleets and heavy traffic. As a specific example, Meta's Katran layer-4 load balancer [8] and CloudFlare's DDoS protection solution [44] process every packet sent to those respective services, and must withstand not only the high rate of request traffic destined to those services but also large denial-of-service attacks. More generally, the academic community has undertaken significant efforts for performance optimization of network functions, including optimization of the software frameworks [46], designing language-based high-performance isolation [61], and developing custom hardware offload solutions [64].

In this paper, we consider applications developed within high-speed packet processing software frameworks. Given the slowdown of Moore's law and the end of Dennard scaling, the software packet-processing performance of single CPU cores has saturated. Even expert developers must work meticulously hard to improve per-core throughput by small margins (like 10%) [37, 50, 52, 67]. The community has pursued various efforts to improve performance, such as re-architecting the software stack to make efficiency gains [21, 56, 69], introducing stateless hardware offloads working in conjunction with the software stack [19, 29], full-stack hardware offloads [30, 34], and kernel extensions [2]. This paper considers software frameworks that modify the device driver to enable programmable and custom functionality to be incorporated by developers at high performance with minimal intervention from the existing kernel software stack [50].

Specifically, we study performance and evaluate our tech-

niques in the context of kernel extensions implemented using the eXpress Data Path (XDP/eBPF [50]) within the Linux kernel. In §3, we will discuss how our observations and principles apply more generally to other high-speed software packet-processing frameworks, including those written with user-space libraries like DPDK [21].

2.2 Parallelizing Stateful Packet Processing

This paper considers the problem of parallelizing high-speed packet processing programs across multiple cores. The key challenge is handling *state*: memory that must be updated in a sequential order upon processing each packet to produce a correct result. Consider the example of the connection tracker [40], a program which identifies the TCP connection state (*e.g.,* SYN sent, SYN/ACK sent, *etc.*) using packets observed from both directions of a TCP connection. Each packet in the connection may modify the internal connection state maintained by the program. There are two main techniques used to parallelize such programs across cores.

Shared state parallelism. One could conceive a parallel implementation that (arbitrarily) splits packets across multiple cores, with explicit synchronization or retry logic guarding access to the shared memory, *i.e.,* the TCP connection state, to ensure a correct result.

Shared-state parallelism works well when the contention to shared memory is low. Specifically, shared-memory scaling could work well when (i) packets of a single flow arrive slowly enough, *e.g.,* if there are a large number of connections with a roughly-equal packet arrival rate, or (ii) when there are efficient implementations available for synchronization or atomic updates in software [47] or hardware [18, 25]. However, neither of these conditions are generally applicable. Many flow size distributions encountered in practical networks are highly skewed [36, 75] or exhibit highly bursty behavior [70], resulting in significant memory contention if packets from the heavier flows are spread across cores. Further, the state update operation in many programs, including the TCP connection tracker, are too complex to be implemented directly on atomic hardware, since the latter only supports individual arithmetic and logic operations (like fetch-add-write). Our evaluation results (§4) show that the performance of shared-state multicore processing plummets with more cores under realistic flow size distributions.

Sharded (shared-nothing) parallelism. Today, the predominant technique to scale stateful packet processing across multiple cores is to process packets that update the same memory at the same core, sharding the overall state of the program across cores. Sharding is achieved through techniques like Receive Side Scaling (RSS [4]), available on modern NICs, to direct packets from the same flow to the same core, and using shared-nothing data structures on each core.

However, sharding suffers from a few disadvantages. First, it is not always possible to avoid coordination through sharding. There may be parts of the program state that are shared across all packets, such as a list of free external ports in a Network Address Translation (NAT) application. On the practical side, the RSS implementations on today's NICs partition packets across cores using a limited number of combinations of packet header fields. For example, a NIC may be configured to steer packets with a fixed combination of source and destination IP addresses (but not a fixed source IP address) to a fixed CPU core. Further, the granularity at which the application wants to shard its state—for example, a key-value cache may seek to shard state by the key requested in the payload—could be infeasible to implement with the packet header sets supported by the RSS capabilities of the NIC [63].

Second, sharding state may create load imbalance across cores if some flows are heavier or more bursty than others, creating hotspots on single CPU cores. Skewed flow size distributions [36], bursty flow transmission patterns [70], and denial of service attacks [43] create conditions ripe for such imbalance. The research community has investigated solutions to balance the packet processing load by migrating flow shards across CPU cores [35, 63]. However, the efficacy of re-balancing is limited by the granularity at which flows can be migrated across cores. As we show in our evaluation (§4), the throughput of the heaviest and most bursty flows is still limited by a single CPU core, which in turn limits the total achieved throughput. Another alternative is to evenly spray incoming packets across cores [41, 71], assuming that only a small number of packets in each flow need to update the flow's state. If a core receives a packet that must update the flow state, the packet is redirected to a designated core that performs all writes to the state for that flow. However, the assumption that state is mostly read-only is not universal, *e.g.,* a TCP connection tracker may update state potentially on every packet. Further, packet reordering at the designated core can lead to incorrect results [35].

2.3 Goals

Given the drawbacks of existing approaches for multi-core scaling discussed above (§2.2), we seek a scaling technique that achieves the following goals:

1. *Generic stateful programming.* The technique must produce correct results for general stateful updates, eschewing solutions that only work for "simple" updates (*i.e.,* fitting hardware atomics) or only update state for a small number of packets per flow.
2. *Skew independence.* The scaling technique should improve performance independent of the incoming flow size distribution or how the flows access the state in the program.
3. *Monotonic performance gain.* Performance should improve, not degrade or collapse, with additional cores.

3 State-Compute Replication (SCR)

In §3.1, we present scaling principles for multi-core stateful packet processing, to meet the goals in §2.3. In §3.2 through §3.4, we show how to operationalize these principles.

3.1 Scaling Principles

To simplify the discussion, suppose the packet-processing program is *deterministic*, *i.e.,* in every execution, it produces the same output state and packet given a fixed input state and packet (we relax this assumption in §3.4).

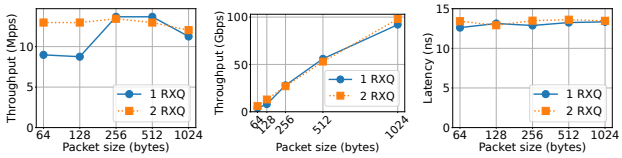
Principle #1 (Replication for correctness). Sending every packet reliably to every core, and *replicating* the state and computation on every core, produces the correct output state and packet on every core *with no explicit cross-core synchronization,* regardless of how the state is accessed by packets.

This principle asks us to treat each core as one replica of a replicated state machine running the deterministic packet-processing program. Each core processes packets in the same order, without missing any packets. With each incoming packet, each core updates a private copy of its state, which is equal to the private copy on every other core. There is no need to synchronize explicitly. Further, replication provides the benefit that the workload across cores is even regardless of how the state is accessed by packets, *i.e.,* skew-independent.

One way to apply this principle naively is to broadcast every packet received *externally* on the machine to every core: with k cores, for each external packet, the system will process k *internal* packets, due to k-fold packet duplication. However, artificially increasing the number of packets processed by the system will significantly hurt performance. In CPU-bound packet processing, smaller packets typically require the same computation that larger packets do. That is, the total amount of work performed by the system is proportional to the packets-per-second offered, rather than the bits-per-second [50, 69].

So how should one use replication for multi-core scaling? Understanding the dominant components of the per-packet CPU work in high-speed packet-processing frameworks offers insight. There are two parts to the CPU processing for each packet after the packet reaches the core where it will be ultimately processed: (i) *dispatch*, the CPU/software labor of presenting the packet to the user-developed packet-processing program, and signaling the packet(s) emitted by the program for transmission by the NIC; and (ii) the *program computation* running within the user-developed program itself. Dispatch often dominates the per-packet CPU work [50].

While these observations are known in the context of high-speed packet processing, we also benchmarked a simple application on our own test machine to validate them. Consider Figure 2, where we show the throughput (packets/second (a), bits/second (b)), and latency (c) of a simple packet forwarder written in the XDP framework running on a single CPU core. Our testbed setup is described in much more detail later (§4),



(a) Packets/second (b) Bits/second (c) Latency (ns)

Figure 2: The nature of CPU work in high-speed packet processing: Consider the throughput of a simple packet forwarding application (packets/second (a), bits/second (b)) running on a single CPU core clocked at 3.6 GHz, as the size of the incoming packets varies. The average latency to execute the XDP program is also shown in nanoseconds (c). CPU usage is tied to the number of packets (not bits) processed per second. Further, significant time elapses in *dispatch:* CPU work to present the input packet to and retrieve the output packet from the program computation.

but we briefly note here that our device-under-test is an Intel Ice Lake CPU configured to run at a fixed frequency of 3.6 GHz and attached to a 100 Gbit/s NIC. At each packet size below 1024 bytes, the CPU core is fully utilized (at 1024 bytes the bottleneck is the NIC bandwidth). The achieved packets/second is stable across all packet sizes which are CPU bound (see the *2 RXQ* curve). With a processing latency of roughly 14 nanoseconds at all packet sizes (measured only for the XDP forwarding program), back-to-back packet processing should "ideally" process $(14 * 10^{-9})^{-1} \approx 71$ million packets/second. However, the achieved best-case throughput ($\approx$ 14 million packets/second) is much smaller—implying that significant CPU work is expended in presenting the input packets to and extracting the output packets from the forwarder and setting up the NIC to transmit and receive those packets. This is not merely a feature of the framework we used (XDP); the DPDK framework has similar dispatch characteristics [50].

Our key insight is that it is possible to replicate program computation without replicating dispatch, enabling multi-core performance scaling. This leads us to the next principle:

Principle #2 (State-Compute Replication). Piggybacking a *bounded recent packet history* on each packet sent to a core allows us to use replication (#1) while equalizing the external and internal packets-per-second CPU work in the system.

Principle #2 states that replication (principle #1) is possible without increasing the total internal packet rate or per-packet CPU work done by the system. Suppose it is possible to spray the incoming packets across cores in a round-robin fashion. If there are k cores, each core receives every k^{th} packet. Then:

1. It is unnecessary for each core to have the most up-to-the-packet private state at all times. For correctness, it is sufficient if a core has a state that is "fresh enough" to make a decision on the current packet that it is processing.
2. With each new packet, suppose the core that receives it also sees (as metadata on that packet) all the $k-1$ packets from the last time it processed a packet, *i.e., a recent packet history*. The core can simply "catch up" on the computations required to obtain the most up-to-the-packet value

for its private state.

3. If packets are sprayed round-robin across cores, the number of historic packets needed to ensure that the most updated state is available to process the current packet is equal to the number of cores. Further, just those packet bits required to update the shared state are necessary from each historic packet, allowing us to pack multiple packets' worth of history into a single packet received at a core.

As a simple model, suppose a system has k cores, and each core can dispatch a single packet in d cycles and runs a packet-processing program that computes over a single packet in c cycles. For each piggybacked packet, the total processing time is $d + (k \times c)$. When dispatch time dominates compute time ($d \gg c$), with k cores, the total rate at which externally-arriving packets can be processed is $k \times \frac{1}{d+(k\times c)} \approx k/d$. Hence, it is possible to scale the packet-processing rate linearly with the number of cores k. In Appendix A, we show that a model like this indeed accurately predicts the empirical throughput achieved with a given number of cores.

Intuitively, doing some extra "lightweight" program computation per packet enables scaling the "heavyweight" dispatch computation with more cores, while maintaining correctness.

Principle #3 (Scaling limits). Principle #2 provides a linear scale-up in the packets-per-second throughput with more cores, so long as dispatch dominates the per-packet work.

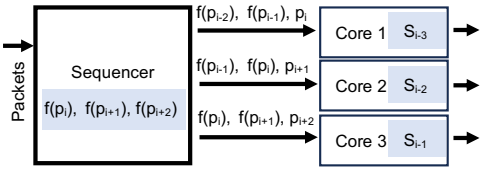
The scaling benefits of principle #2 taper off beyond a point. Dispatch can be overtaken as the primary contributor to per-packet CPU work, for example, when (i) the compute time $k \times c$ for each piggybacked packet becomes sizable; (ii) the per-packet compute time c itself increases due to overheads in the system, *e.g.,* larger memory access time when a core's copy of the state spills into a larger memory; or (iii) other components such as the NIC or PCIe become the bottleneck rather than the CPU. When this happens, the approximation in our simple linear model ($d \gg c$) no longer holds, and the system's packet rate no longer scales linearly with cores.

3.2 Operationalizing SCR

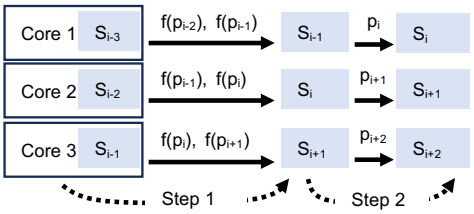
Operationalizing the scaling principles discussed above (§3.1) conceptually requires two pieces.

A reliable packet history sequencer (§3.3). We require an additional entity in the system, which we call a *sequencer,* to (i) steer packets across cores in round-robin fashion, (ii) maintain the most recent packet history across all packets arriving at the machine, and (iii) piggyback the history on each packet sent to the cores. After the packet is processed by a CPU core, its piggybacked history can be stripped off on the return path either at the core itself or at the sequencer. The size of the packet history depends only on the number of cores and metadata size (§3.1) and is independent of the number of active flows.

The NIC hardware or the top-of-the-rack switch are natural points to introduce the sequencer functionality, since they observe all the packets flowing into and out of the machine. Today's existing fixed-function NICs do not implement the functionality necessary to construct and piggyback a reliable packet history. However, we have identified two possible instantiations that could, in the near future, achieve this: (i) emerging NICs, *e.g.,* with programmable pipelines [1, 3, 27], implementing the full functionality of the reliable sequencer; or (ii) a combination of a NIC implementing round-robin packet steering [4, 7] and a programmable top-of-the-rack switch pipeline [17, 23, 38] for maintaining and piggybacking the packet history. We believe that either of these instantiations may be realistic and applicable given the context: for example, high-speed programmable NICs are already common in some large production systems [45], as are programmable switch pipelines [60]. We will show two possible hardware designs in §3.3. Hereafter, for brevity, we refer to both of these designs as simply sequencers.



(a) The sequencer stores relevant fields from the packet history, and piggybacks the history on packets sprayed round-robin across cores.



(b) Each core fast-forwards its private state and then handles its packet.

Figure 3: An example illustrating the scaling principles. p_i is the i^{th} packet received by the sequencer, $f(p_j)$ are relevant fields from p_j, and S_i is the state after processing packets $p_1, \ldots, p_i$ in order.

An SCR-aware packet-processing program. The packet-processing program must be developed to replicate the program state and keep private copies per core. Further, the program must process the packet history first before computing over the current packet. In Appendix C, we show an example that demonstrates how to transform a single-threaded program to its SCR-aware variant. We believe that these program transformations can be automated in the future.

An example showing scaling principles in action. Consider Figure 3, where a sequencer and three cores are used to run a packet-processing program. As shown in Figure 3a, the sequencer sprays packets (*i.e.,* $p_i, p_{i+1}, \ldots$) in a round-robin fashion across $k = 3$ cores (*i.e.,* $core_1, core_2, core_3$). Further, the sequencer stores the recent packet history consisting of the packet fields from the last k packets which are relevant to evolving the flow state. We denote the relevant part of a

packet p_i by $f(p_i)$. For example, in a TCP connection tracking program, this includes the TCP 4-tuple, the TCP flags, and sequence and ACK numbers. Note that this packet history is updated only by the sequencer and is never written to by the cores. In the example in Figure 3a, the packet history supplied to $core_1$ processing packet p_i is $f(p_{i-2}), f(p_{i-1})$. As shown in Figure 3b, each core updates its local private state, first fast-forwarding the state by running the program through the packet history $f(p_{i-2}), f(p_{i-1})$, and then processing the packet p_i sprayed to it.

If the packet-processing program is deterministic (§3.1), an SCR-aware program is guaranteed to produce the correct output state and packet if every CPU core is guaranteed to (losslessly) receive the packets sent to it by the sequencer. We show how to handle non-determinism and packet loss in §3.4.

3.3 Packet History Sequencer

The primary goal of the sequencer is to maintain and propagate recent packet history to CPU cores to help replicate the computation with the correct program state (§3.2). We assume the NIC is already capable of spraying packets across CPU cores [4, 7], and hence do not discuss that functionality further. We describe the rest of the sequencer's functions in terms of the following: (i) designing a packet format that modifies existing packets to piggyback history from the sequencer to the CPU cores; (ii) designing a hardware data structure that maintains a recent bounded packet history at the sequencer, and enables reading out the history into metadata on the packet. The packet fields that are maintained in the sequencer history depend on the specific fields used by the packet-processing application. The number of historic packets that must be tracked depends on the degree of parallelism that is sought, *e.g.,* the number of available CPU cores over which scaling is implemented.

We have implemented sequencing hardware data structures on two platforms, the Tofino programmable switch pipeline [23] and a Verilog module that we integrated into the NetFPGA-PLUS project [12].

3.3.1 Packet format

The key question answered in this subsection is: given a packet, what is the best place to put the packet history on it? While this may initially appear to be just an engineering detail, the packet format has important implications to the design of hardware data structures on the sequencer and the SCR-aware program.

As shown in Figure 4a, we choose to place the packet history close to the beginning of the packet, before the entirety of the original packet. Relative to placing the packet history between headers of the original packet, this placement simplifies the hardware logic that writes the history into the packet, as the write must always occur at a fixed address (0) in the packet buffer. Further, for reasons explained in §3.3.2, we include a pointer to the metadata of the packet that arrived the earliest among the ones in the piggybacked history. The earliest packet does not always correspond to the first piece of metadata when reading the bytes of the packet in order.

Keeping all the bytes of the original packet together in one place also simplifies developing an SCR-aware packet-processing program. The packet parsing logic of the original program can remain unmodified if the program starts parsing from the location in the modified packet buffer which contains all the bytes of the original packet in order.

Finally, we also prefix an additional Ethernet header to the packet in instantiations of the sequencer which run outside of the NIC, *i.e.,* a top-of-the-rack switch. Adding this header helps the NIC process the packet correctly: without it, the packet appears to have an ill-formatted Ethernet MAC header at the NIC. Our setup also uses this Ethernet header to force RSS on the NIC [4] to spray packets across CPU cores (our testbed NIC (§4.1) supports hashing on L2 headers). This additional Ethernet header is not needed in a sequencer instantiation running on the NIC.

3.3.2 Hardware data structures for packet history

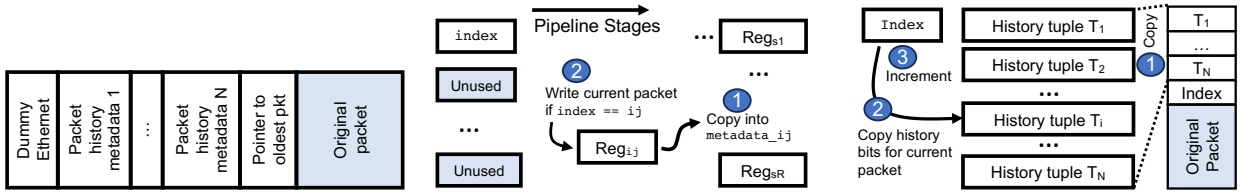
We show how to design data structures to maintain and update a recent packet history on two high-speed platforms, a Tofino programmable switch pipeline [23] and a Verilog module integrated into the NetFPGA-PLUS platform [12]. These designs are specific to the platform where they are implemented, and hence we describe them independently.

A key unifying principle between the two designs is that although the items in the maintained packet history change after each packet, we only update a small part of the data structure for each packet. Conceptually, a ring buffer data structure is appropriate to maintain such histories. Hence, in both designs, we use an *index pointer* to refer to the current data item that must be updated, which corresponds to the head pointer of the abstract ring buffer where data is written.

Tofino. We use Tofino registers [26], which are stateful memories to hold data on the switch, to record the bits of each historic packet relevant to the computation in the packet-processing program. We use the Tofino parser to extract these bits, which restricts our design only to support historic fields up to 4 Kilobits deep in the packet [38].

Suppose the pipeline has s match-action table stages, R registers per stage, and b bits per register. For simplicity in this description, we assume there is exactly one packet field of size b bits used in the computation in the packet-processing program. Our data structure can maintain a maximum of $(s-1) \times R \times b$ bits of recent packet history, *i.e.,* history for $(s-1) \times R$ packets, as shown in Figure 4b. We have successfully compiled the design to the Tofino ASIC.

First, we use a single register in the first stage to store the index pointer. The pointer refers to the specific register in the



(a) Packet format (b) Tofino sequencer (c) RTL sequencer

Figure 4: Hardware data structures. (a) Packets modified to propagate history from the sequencer to CPU cores. The sequencer prefixes the packet history to the original packet, which allows for a simpler implementation in hardware (§3.3) and simpler transformations to make a packet-processing program SCR-aware (Appendix C). In instantiations where the sequencer is partly implemented on a top-of-the-rack switch (§3.2), we further prefix a dummy Ethernet header to ensure that the NIC can process the packet correctly. (b) The data structure used to maintain and propagate packet history on the Tofino programmable switch pipeline (§3.3.2). Inset shows the specific actions performed on each Tofino register. (c) The data structure used to maintain and propagate packet history in our Verilog module on NetFPGA-PLUS (§3.3.2).

subsequent stages that must be updated with a header field from the current packet. The index pointer is incremented by 1 for each packet, and wraps back to 0 when it reaches the maximum number of packets required in the history. The pointer is also carried on a metadata field on the packet through the remaining pipeline stages.

Next, register ALUs in subsequent stages are programmed to read out the values stored in them into pre-designated metadata fields on the packet. If the index pointer points to this register, an additional action occurs: rewrite the stored contents of the register by the pre-designated history fields from the current packet.

Finally, all the metadata fields, consisting of the packet history fields and the index pointer, are deparsed and serialized into the packet in the format shown in Figure 4a. We also add an additional Ethernet header to ensure that the server NIC can receive the packets correctly (§3.3.1).

Recent work explored the design of ring buffers to store packet histories in the context of debugging [53], reading out the histories from the control plane when a debugging action is triggered. A key difference in our design is that reading out histories into the packet is a data plane operation, occurring on every packet.

NetFPGA. To show the possibility of developing high-speed fixed-function hardware for sequencing, we also present a sequencer design developed in Verilog in Figure 4c.

Suppose we wish to maintain a history of N packets, each packet contributing b bits of information. A simple design, for small values of N and b (we used $N = 16$ and $b = 112$), uses a memory which has N rows, each containing a tuple of b bits. We also maintain a register containing the index pointer (p bits), initialized to zero. At the beginning, the memory is initialized with all zeroes. When a packet arrives, it is parsed to extract the bits relevant to the packet history. Then the entire memory is read and put in front of the packet (moving the packet contents by a fixed size known beforehand, $N \times b + p$ bits). The information relevant to the packet history from the current packet is put into the memory row pointed to by the index pointer, and the index pointer is incremented (modulo the memory size). We have integrated this design into the NetFPGA-PLUS platform.

3.4 Discussion

In this section, we discuss how to generalize SCR to handle non-determinism, packet loss, and multiple packet-processing programs.

Handling Packet Loss. Packets can be lost either prior to the sequencer, after the sequencer but prior to processing at a CPU core, or after processing at a core. Among these, we only care about the second kind of packet loss, since this is the only one that is problematic specifically for SCR, introducing the possibility that flow states on the CPU cores might become inconsistent with each other.

We expect that SCR will be deployed in scenarios where the event of packet loss between the sequencer and CPU cores is rare. First, we do not anticipate any packet loss in an instantiation where the sequencer is running entirely on a NIC, since the host interconnect between the NIC and CPU cores uses credit-based flow control and is lossless by design [32, 59]. In an instantiation where the sequencer is running on a top-of-the-rack switch, it is possible to run link-level flow control mechanisms like PFC [6] (as some large production networks do [48]) to prevent packet loss between the switch and server cores. We discuss below how SCR can handle rare packet drop and corruption events while maintaining consistency among the states on CPU cores.

How should a CPU core that has lost a packet arriving from the sequencer synchronize itself to the correct flow state? There are two design options: the core can either explicitly read the full flow state from a more up-to-date core, or it can read the packet history from either the sequencer or a log written by a more up-to-date core, and then use the history to catch up its private state (akin to Figure 3). Since we operate in a regime where packet losses are rare, but the full set of flow states is large, we prefer to synchronize the packet history rather than the state. Further, to simplify the overall design, we avoid explicit coordination between the cores and the sequencer, synchronizing the history among the cores only.

Our objective is *atomicity:* any packet is either processed by all the cores or none of the cores. If a packet is sent by the sequencer to *any* core (in original or as part of the packet history), it should be processed in the correct order by *all* the cores. To achieve this, we (i) have the sequencer attach an incrementing sequence number to each packet released by it; (ii) use a per-core, lockless, single-writer multiple-reader log, into which each core writes the history contained in each packet it receives (including the relevant data for the original packet); and (iii) introduce an algorithm to catch up the flow state on each core upon detection of loss.

The algorithm proceeds as follows (more information is available in Algorithm 1 in Appendix B). Each CPU core c maintains a per-core log with one entry for each sequence number i. In a system with N cores, the history metadata of the packet with sequence i (say $history[i]$) will appear in packets with sequence numbers i through $i+N-1$; conversely, a packet with sequence number j contains $history[minseq], \cdots, history[j]$ where $minseq \triangleq max(1, j-N+1)$.

For core c and sequence number i, $log[c][i]$ takes on one of three possible values:

$$\begin{cases} history[i] & \text{if history for sequence } i \text{ was received at } c \\ NOT_INIT & \text{if the highest sequence received at } c \text{ is } j < i \\ LOST & \text{if } c \text{ has received sequence } j > i, \\ & \text{but sequence } i \text{ was not received at } c \end{cases}$$

At the beginning, for all cores c and sequence numbers i, $log[c][i]$ is set to NOT_INIT, to denote that each log entry is uninitialized at every core. When a (fixed) core c receives a packet with sequence number j, it first detects packet loss by comparing the max sequence number it has seen so far (say $max[c]$) with the earliest sequence number in the new packet it receives ($minseq$), assuming no reordering between the sequencer and the core. Then, c processes every sequence number k such that $max[c] < k \leq j$, in order of increasing k, as follows:

1. if $k < minseq$, *i.e.,* sequence k was lost between the sequencer and core c, the core updates $log[c][k] \leftarrow LOST$. For such packets, core c will read from the logs of other cores $c' \neq c$ in a loop, until c discovers either that (i) $history[k]$ is written in $log[c'][k]$, in which case c catches up its private state by reading this history; or (ii) $log[c'][k] = LOST$ on all cores $c' \neq c$, concluding that sequence k was never originally received on any core, and does not need to be recovered for atomicity;
2. if $minseq \leq k \leq j$, *i.e.,* sequence k is successfully received at core c (as part of the current packet), core c updates $log[c][k] \leftarrow history[k]$ available in the packet, and then proceeds with regular processing as in §3.2.

In Appendix B, we formally prove that, under some mild assumptions, this algorithm always terminates in a state that is eventually consistent across all CPU cores. Despite cores possibly waiting on one another, there will be no deadlocks. While the treatment above uses an infinitely large log, practically, our log implementation is a circular buffer with a fixed size (it is unnecessary to garbage-collect the log). The buffer must be sized large enough to recover from packet losses and transient speed mismatches across CPU cores, both of which are expected to be small in practical deployments.

Handling programs that depend on timestamps. The use of timestamps measured locally at each CPU core (*e.g.,* to implement a token bucket rate limiter) may result in the results of computations at different CPU cores diverging from each other. To handle this, we avoid measuring time locally at each CPU core, and instead have the sequencer attach a timestamp for each packet to the packet history. Modern NICs and programmable switches support high-resolution hardware timestamping over packets [13, 26].

Handling programs involving randomization. For SCR to produce a consistent state across cores, it is necessary that the state computations on all CPU cores agree on the result even if the computations involve random numbers. None of our current benchmark programs involve randomization. For those programs that do, we recommend to fix the seed of the pseudorandom number generator to the same value across different CPU cores.

Handling chained packet-processing programs. SCR can handle multiple packet-processing programs run sequentially (for example, for service function chaining [49]) by piggybacking the union of the historical packet fields for all the programs on each packet from the sequencer to the core. In addition to the program changes typically needed for SCR parallelism (Appendix C), the programs must also be rewritten to handle packets that include additional fields of history for the co-resident programs. We believe this can be accomplished by the design of a suitable automatic compiler. We leave the design of such compilers to future work.

4 Evaluation

We seek to answer two main questions through the experiment setup described in §4.1.
(1) Does state-compute replication provide better multi-core scaling than existing techniques (§4.2)?
(2) How practical is sequencer hardware (§4.3)?

4.1 Experiment Setup

Machines and configurations. Our experiment setup consists of two server machines connected back-to-back over a 100 Gbit/s Nvidia/Mellanox ConnectX-5 NIC on each machine. Our servers run Intel Ice Lake processors (Xeon Gold 6334) with 16 physical cores (32 hyperthreads) and 256 GB DDR4 physical memory spread over two NUMA nodes. The system bus is PCIe 4.0 16x. We run Ubuntu 22 with Linux kernel v6.5.

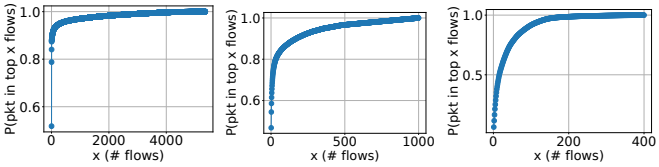
Program	State		Metadata size	RSS hash	Packet traces	Atomic HW	Lines of code
	Key	Value	(bytes/packet)	fields	evaluated	vs. Locks	(shard/RSS)
DDoS mitigator	source IP	count	4	src & dst IP	CAIDA, Univ DC	Atomic HW	168
Heavy hitter monitor	5-tuple	flow size	18	5-tuple	CAIDA, Univ DC	Atomic HW	141
TCP connection state tracking	5-tuple	TCP state, timestamp, seq #	30	5-tuple	Hyperscalar DC	Locks	1029
Token bucket policer	5-tuple	last packet timestamp, # tokens	18	5-tuple	CAIDA, UnivDC	Locks	169
Port-knocking firewall	source IP	knocking state (*e.g.,* `OPEN`)	8	src & dst IP	CAIDA, UnivDC	Locks	123

Table 1: The packet-processing programs we evaluated.

One machine serves as a packet replayer/generator, running a DPDK burst-replay program which can transmit packets from a traffic trace. We have tested that the traffic generator can replay large traces (1 million packets) at speeds of $\sim$ 120 million packets/second (Mpps), for sufficiently small packets (so that the NIC bandwidth is not saturated first). The traffic generator can be directed to transmit packets at a fixed transmission (TX) rate and measure the corresponding received (RX) packet rate. Our second server is the Device Under Test (DUT), which runs on identical hardware and operating system as the first server. We implement standard configurations to benchmark high-speed packet processing [50]: hyperthreading is disabled; the processor C-states, DVFS, and TurboBoost are disabled; dynamic IRQ balancing is disabled; and the clock frequency is set to a fixed 3.6 GHz. We enable PCIe descriptor compression and use 256 PCIe descriptors. Receive-side scaling (RSS [4]) is configured according to the baselines/programs, see Table 1. We use a single receive queue (RXQ) per core unless specified otherwise. On our setup, we have checked that the per-core packet-forwarding throughput is comparable to prior work benchmarking the XDP framework [50]. However, the absolute numbers are lower than the multi-core throughput reported in recent literature using DPDK. We believe this may be due to the differences in the packet-processing framework (XDP vs. DPDK).

The definition of throughput. We use the standard *maximum loss-free forwarding rate* (MLFFR [5]) methodology to benchmark packet-processing throughput. Our threshold for packet loss is in fact larger than zero (we count $< 4\%$ loss as "loss-free"), since, at high speeds we have observed that the software typically always incurs a small amount of bursty packet loss. We use binary search to expedite the search for the MLFFR, stopping the search when the bounds of the search interval are separated by less than 0.4 Mpps. Experimentally, we observe that MLFFR is a stable throughput metric: we get highly repeatable results across multiple runs. We only report throughput from a single run of the MLFFR binary search.

Traces. We are interested in understanding whether SCR provides better multi-core scaling than existing techniques on realistic traffic workloads. We have set up and used three traces for throughput comparison: a university data center trace [36], a wide-area Internet backbone trace from CAIDA [11], and a synthetic trace with flows whose sizes and inter-arrivals were sampled from a hyperscalar's data center flow characteristics [33]. These traces are highly dynamic, with flow states being created and destroyed throughout—an aspect that we believe is crucial to handle in real deployment environments (*i.e.,* the programs are not simply processing a stable set of active flows). Further, we ensure that all TCP flows that begin in the trace also end, by setting TCP SYN and FIN flags for the first and last packets (respectively) of each flow in the trace. This allows the trace to be replayed multiple times with the correct program semantics. The flow size distributions of the traces are shown in Figure 5.



(a) University DC (b) Internet backbone (c) Hyperscalar DC

Figure 5: Flow size distributions of the packet traces we used. We used real packet traces captured at (a) university data center [36] and (b) wide-area Internet backbone by CAIDA [11]. We also synthesized (c) a packet trace with real TCP flows whose sizes are drawn from Microsoft's data center flow size distribution [33].

The eBPF framework limits our implementations in terms of the number of concurrent flows that our stateful data structures can include. This is not a limitation of the techniques, but an artifact of the current packet-processing framework we use (eBPF/XDP). To account for this limitation, specifically for the CAIDA trace, we have sampled flows from the trace's empirical flow size distribution to faithfully reflect the underlying distribution, without over-running the limit on the number of concurrent flows that any of our baseline programs may hold across the lifetime of the experiment.

Baselines. We compare state-compute replication against (i) state sharing, an approach that uses either hardware atomic instructions (when the stateful update is simple enough) or eBPF spinlocks [10] (when it is not) to share state across CPU cores; (ii) state sharding using classic RSS; and (iii) sharding using RSS++ [35, 63], the state-of-the-art flow sharding technique to balance CPU load. RSS++ solves an optimization problem that takes as input the incoming load imposed by flow shards, and migrates shards to minimize a linear combination of load imbalance across CPU cores and the number of cross-core shard transfers needed. Both SCR and state sharing spray packets evenly across CPU cores. The packets sent to

each core for the sharding techniques depends on the configuration of RSS, which varies across the programs we evaluated (see below and Table 1). Running RSS++ over eBPF/XDP requires patching the NIC driver [9] to expose the RX hash on packets to XDP programs. Unless specified otherwise, we run SCR without loss recovery (§3.4) as we believe this is the most representative scenario in which SCR will be deployed. We evaluate loss recovery separately (§4.2).

Programs. We tested five packet-processing programs developed in eBPF/XDP, including (i) a heavy hitter monitor, (ii) DDoS mitigator, (iii) TCP connection state tracker, (iv) port-knocking firewall, and (v) a token bucket policer. Table 1 summarizes these programs. Each program maintains state across packets in the form of a key-value dictionary, whose size and contents are listed in the table. We developed a cuckoo hash table to implement the functionality of this dictionary with a single BPF helper call [15]. The packet fields in the key determine how RSS must be configured: packets having the same key fields must be sent to the same CPU core. However, today's NICs do not allow RSS to steer packets on arbitrary sets of packet fields [63]. We pre-process the trace to ensure that RSS hashing indeed shards the flow state correctly. For example, on the NIC in our testbed, the source (`srcip`) and destination (`dstip`) IP addresses may be used together, but not separately, as the RSS key to hash a packet to a core. For a program that maintains flow state at the granularity of `dstip`, an RSS key (`srcip, dstip`) could steer two packets with the same `dstip` but different `srcip` to different CPU cores, violating sharding at the granularity of `dstip`. To prevent this, and to evaluate our sharding baselines fairly, we pre-process our traces (*e.g.,* modifying packets such that every `srcip, dstip` combination in the trace hashes to a core that only depends on `dstip`) to ensure that RSS hashing indeed shards the flow state correctly. For the connection tracker, since both directions of the connection must go to the same CPU core, we use the keyed hash function prescribed by symmetric RSS [74].

4.2 Multi-Core Throughput Scaling

In this section, we compare the MLFFR throughput (§4.1) of several packet-processing programs (Table 1) scaled across multiple cores using four techniques: SCR (§3), state sharing with packets sprayed evenly across all cores, sharding using RSS, and sharing using RSS++ (§2). Since the TCP connection tracking program requires packets from the two directions of the connection to be aligned, we evaluated it on a synthetic but realistic hyperscalar data center trace (§4.1). For the rest of the programs, we report results from real university data center and Internet backbone traces.

We have ensured that these experiments reflect a fair comparison of CPU packet-processing efficacy. First, we truncated the packets in the traces to a size smaller than the full MTU, to stress CPU performance with a high packets/second (Mpps) workload (§3.1). Further, we fix the packet sizes used across all baselines for a given program. We used a fixed packet size of 256 bytes for the connection tracker and 192 bytes for the others. The packet size limits the number of items of history metadata that can be piggybacked on each packet. Since the metadata size changes by the program (Table 1), the maximum number of cores we can support for a fixed packet size also varies by program (we support 7 cores for the token bucket, heavy hitter detector, and connection tracker, and 14 for the DDoS mitigation and port-knocking firewall).

Throughput results. Figure 6 and Figure 7 show the throughput as we increase the number of packet-processing cores. SCR is the only multi-core scaling technique that can monotonically scale the throughput of all the stateful packet-processing programs we evaluated across multiple cores, regardless of the flow size distribution (§2.3). The throughput for SCR increases linearly across cores in all of the configurations we tested. Somewhat surprisingly, SCR provides even better absolute performance than hardware atomic instructions in the case of the heavy hitter and DDoS mitigation programs. However, the performance of lock-based sharing falls off catastrophically with 3 or more cores.

The throughput of sharding using RSS depends on how the RSS hash function steers flows to cores. On our experimental setup, RSS can split flows evenly across CPU cores. However, RSS neither splits individual heavy flows, nor does it intelligently distribute heavy flows across CPU cores to balance the load. When the packet rates of individual heavy flows in the workload are within the processing capabilities of a single CPU core, RSS++ [35] can improve upon RSS by spreading the heavy flows across CPU cores. By measuring and adapting to the workload on each CPU core, RSS++ can scale throughput effectively with additional CPU cores, as has been shown in prior work [35, 63]. However, when the workload includes heavy flows whose (individual) packet rates exceed the processing capability of a single CPU core, and the number of such flows is smaller than the number of available CPU cores, simply spreading out the heavy flows across cores is insufficient to improve throughput with additional cores. This is the situation with the real packet workloads we tested (§4.1). Such situations also arise under attacks that force sharded packets to be handled on the same CPU core [43, 62]. It becomes necessary to process individual heavy flows using multiple cores, which is what SCR does.

Our results also show that RSS++ is not always better than RSS. Re-balancing load by migrating a flow shard across cores requires bouncing the cache line(s) containing the flow states across cores, an action that can degrade performance if done too frequently. On the other hand, rebalancing too infrequently may fail to mitigate skew across cores, since the per-shard load used to make CPU-balancing decisions may become stale relative to the current load.

Why does SCR scale better than other techniques? Figure 8 shows detailed performance metrics measured from Intel's

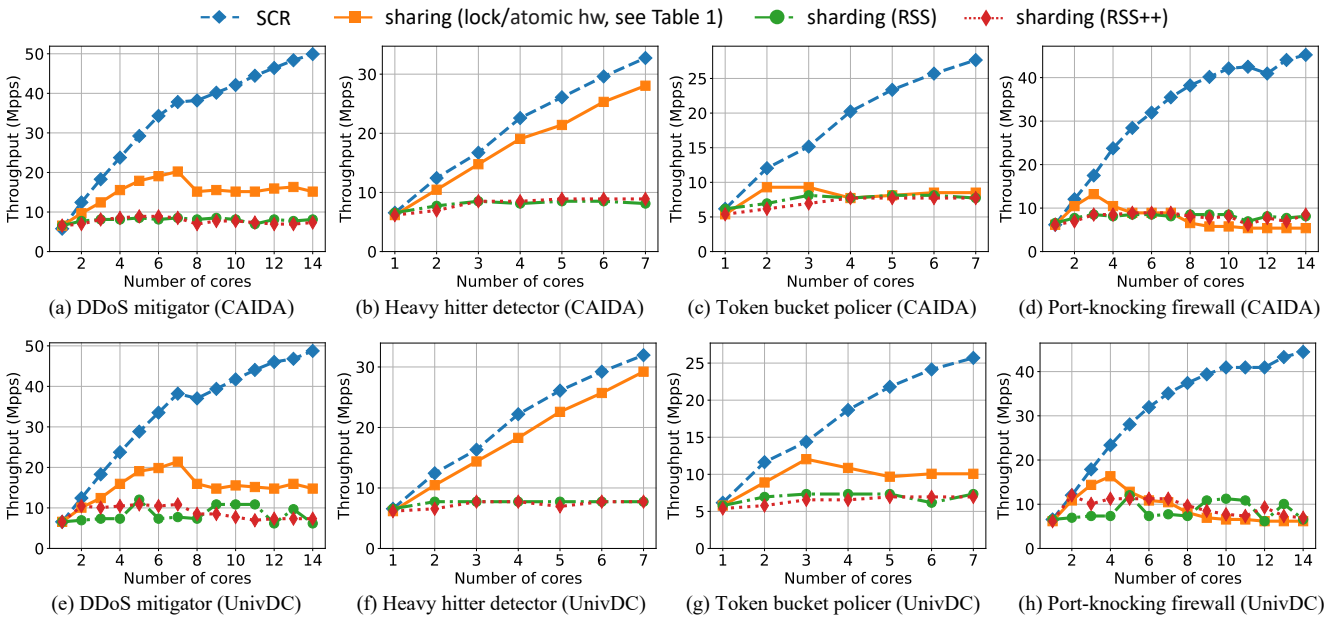


Figure 6: Throughput (millions of packets per second) of four stateful packet-processing programs implemented using state-compute replication (§3), shared state, and sharding (§2). Packet traffic is replayed from real data center and Internet backbone traces (§4.1).

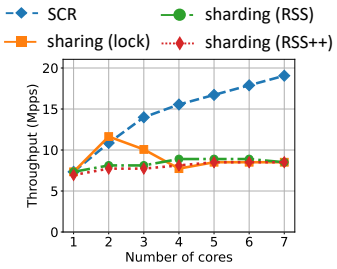


Figure 7: Throughput of TCP connection tracking parallelized using four techniques, SCR (§3), shared state, sharding with RSS, and sharding with RSS++ [35], on a hyperscalar data center trace (§4.1).

performance counter monitor (PCM [24]) and BPF profiling [14]. We measure the L2 cache hit ratios, instructions retired per cycle (IPC), and the program's computation latency (only the XDP portion, excluding the dispatch functionality in the driver), as the load offered to the system increases, when our token bucket policer is run across different numbers of cores (2, 4, or 7). The numbers show the averages for these metrics across the cores running the program. Error bars for IPC show the min and max values across cores. IPC is a meaningful metric to evaluate "CPU goodput" for eBPF/XDP programs: unlike high-speed packet processing frameworks like DPDK which poll the NIC and exhibit persistently high IPC [42], eBPF/XDP drivers adapt CPU usage to load through a mix of polling and interrupts.

Lock-based sharing in general suffers from lower L2 cache hit ratios ((a)–(c)) and higher latencies ((g)–(i)) due to lock and cache line contention across cores—a trend that holds as the offered load increases and also with additional cores

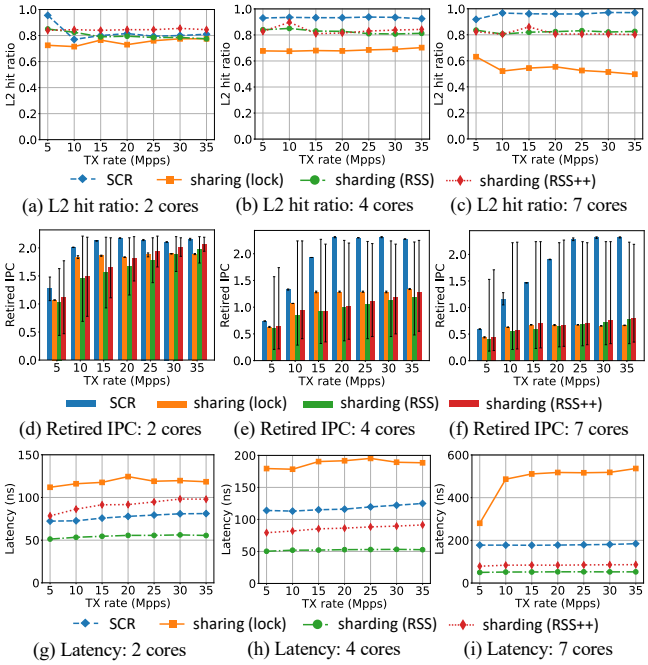
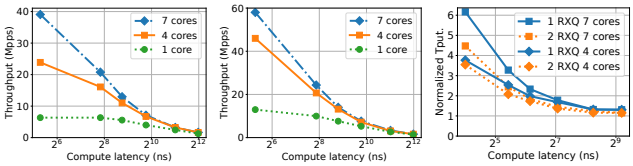


Figure 8: Hardware performance metrics drawn from Intel PCM while executing the token bucket program. As the offered load increases, we show the program's compute latency (measured purely for the XDP portion), the L2 hit ratio, and the number of instructions retired per CPU clock cycle (IPC), when the program is scaled to 2, 4, or 7 cores. Packet traffic is from a university data center (§4.1).



(a) Packets/sec (1 RXQ) (b) Packets/sec (2 RXQ) (c) Normalized to 1 core

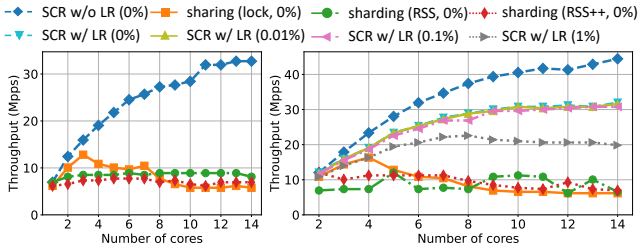
Figure 9: The throughput scaling of a stateless program using SCR, as the compute latency of the program varies but the dispatch latency remains constant ((a) and (b) in packets/second, and (c) normalized against single-core throughput at the same compute latency). As discussed in §3.1, the more the dispatch time dominates compute time, the more effective the multi-core scaling from SCR.

at the same offered load. As we might expect, IPC increases with the offered load ((d)–(f)), since the cores get busier with packet processing. While the sharding approaches (RSS and RSS++) effectively use the CPU with a high average IPC for 2 cores, their average IPC values drop significantly with additional cores, with large variation across cores (see error bars). This indicates a severe imbalance of useful CPU work across cores: Flow-affinity-based sharding approaches are unable to balance packet-processing load across cores, leaving some cores idle and others heavily used.

In contrast to sharing and sharding approaches, SCR has a consistently high IPC with more cores and higher offered loads. SCR has higher packet-processing latency ((g)–(i)) than RSS-sharding since it needs to process the history for each packet. RSS++ sometimes incurs higher compute latency than SCR due to its need to monitor per-shard load, which requires additional memory operations. SCR's more effective usage of the CPU cores results in better throughput scaling than RSS and RSS++ (Figure 6 (g)).

Limits to SCR scaling. SCR suffers from two kinds of scaling limitations. First, as discussed in §3.1, as the compute latency increases in comparison to the dispatch latency, the effectiveness of SCR's multi-core scaling reduces. This is because more time is spent "catching up" state, incurring significant duplicated work across CPU cores. We evaluate how the throughput of a stateless program varies as the compute latency of this program increases, shown both in packets/second (Figure 9a, Figure 9b) and normalized to the single-core throughput for that compute latency (Figure 9c). With a small compute latency (left of the graph), using N cores provides $\approx N\times$ throughput relative to a single core, but this relative benefit diminishes with increasing compute latency. Latency profiles of our benchmarks show compute latencies smaller than 70 nanoseconds (Table 4, Appendix A).

Second, SCR's attachment of histories to packets incurs non-negligible byte/second overheads. Adding to the number of bytes per packet increases L3 cache pressure due to higher DDIO cache occupancy [22] and incurs additional PCIe transactions and bandwidth [59]. Further, when packet histories are appended outside the NIC (*e.g.,* top-of-the-rack switch),



(a) Adding packet history externally (b) Impact of loss recovery

Figure 10: (a) The throughput of a token bucket policer on the university data center trace (§4.1), while truncating all packets in the trace to 64 bytes, with only SCR adding metadata to packets before feeding them to the NIC. (b) The throughput of a port-knocking firewall on the university data center trace. SCR is run with and without loss recovery (§3.4) at multiple packet loss rates.

Rows	LUT			Flip-flops	
	Usage	Logic	%	Usage	%
16	1045	646	0.060	2369	0.069
32	1852	1444	0.107	3158	0.091
64	2637	2229	0.153	4707	0.136
128	3390	2982	0.196	7786	0.226

Table 2: Sequencer resource usage after synthesis into the NetFPGA-PLUS reference switch and meeting timing at 340 MHz.

SCR may saturate the NIC earlier than other approaches. We compare SCR against the shared and sharded approaches, when SCR alone adds history metadata before packets are fed into the NIC while the packets for the latter approaches are truncated to 64 bytes. Figure 10 (a) shows the throughput of the token bucket program with the university data center trace. After 11 cores, the CPU is no longer the bottleneck for SCR. This prevents SCR from scaling to a higher packets/second throughput. Yet, SCR saturates at a throughput much higher than the other techniques.

Overheads of SCR's loss recovery handling. We evaluate how SCR's loss recovery algorithm impacts throughput with and without packet loss. Figure 10 (b) compares a version of SCR without incorporating the loss recovery algorithm, against a version that incorporates loss recovery at different artificially-injected random packet loss rates (0%, 0.01%, 0.1% and 1%). We also show the performance of existing scaling techniques (shared state, RSS, RSS++). The mere inclusion of the loss recovery algorithm impacts performance due to the additional logging operations. Moreover, SCR's throughput degrades with higher loss rate due to recovery-related synchronization (§3.4). However, SCR still outperforms and outscales existing multi-core scaling techniques.

4.3 Practicality of Sequencer Hardware

We integrated our Verilog module implementing the sequencer (§3.3.2) into the NetFPGA-PLUS [12] reference

Resource	Avg%	Resource	Avg%
Exact match crossbars	23.31%	SRAM	9.69%
VLIW instructions	9.11%	TCAM	0.00%
Stateful ALUs	93.75%	Map RAM	15.62%
Logical tables	23.96%	Gateway	23.44%

Table 3: Resource usage (average % across stages) of a Tofino implementation of the sequencer that uses as many stateful ALUs as possible to store packet history, amounting to 44 32-bit fields.

switch, which is clocked at a maximum frequency of 340 MHz with a 1024-bit wide data bus, providing a bandwidth of 348 Gbit/s. We use the Alveo U250 board, which contains 1728000 lookup tables (LUTs) and 3456000 flip-flops.

We synthesized our sequencer design with different numbers of memory rows (§3.3.2), corresponding to the size of the packet history (in number of packets). Each row is 112 bits long, enough to maintain a TCP 4-tuple and an additional 16-bit value (*e.g.,* a counter, timestamp, *etc.*) for each historic packet. Table 2 shows the resource usage. If each packet history metadata in the program is smaller than a row (112 bits), parallelizing across N cores requires N rows. For such programs, our design can meet timing (340 MHz) while scaling to 128 cores. The LUT and flip-flop hardware usage is negligible compared to the FPGA capacity at all row counts measured. We believe that our sequencer design may be simple and cheap enough to be added as an on-chip accelerator to a future NIC.

We have also implemented a stateful-register-based design of the sequencer on the Tofino programmable switch (§3.3.2). The resource usage of this design is shown in Table 3. Our implementation was designed to use as many stateful registers and ALUs as possible (our design uses 93% on average across stages) to hold the largest number of bits of packet history. Our design holds 44 32-bit fields, sufficient to parallelize (Table 1) the DDoS mitigator over 44 cores, the port-knocking firewall over 22 cores, the heavy hitter and token bucket over 9 cores, or the connection tracker over 5 cores. The small number of stateful ALUs on the platform, as well as the limit on the number of bits that can be read out from stateful memory into packet fields, restrict the Tofino sequencer from scaling to a larger number of CPU cores.

5 Related Work

Frameworks for network function performance. The problem of scaling out packet processing is prominent in network function virtualization (NFV), with frameworks such as split/merge [68], openNF [46], and Metron [54] enabling elastic scaling. There have also been efforts to parallelize network functions automatically [63] and designing data structures to minimize cross-core contention [47]. These efforts are flow-oriented, managing and distributing state at flow granularity. In contrast, SCR scales packet processing for a single flow.

General techniques for software parallelism. Among the canonical frameworks to implement software parallelism [57], our scaling principles are most reminiscent of Single Program Multiple Data (SPMD) parallelism, with the program being identical on each core but the data being distinct. The sequencer in SCR makes the data distinct for each core.

Parallelizing finite state machines. A natural model of stateful packet processing programs is as finite state automata (the state space is the set of flow states) making transitions on events (packets). There have been significant efforts taken to parallelize FSM execution using speculation [65, 66] and data parallelism [58]. In contrast, SCR exploits replication.

Parallel network software stacks. There has been recent interest in abstractions and implementations that take advantage of parallelism in network stacks, for TCP [55, 72] and for end-to-end data transfers to/from user space [39]. SCR takes a complementary approach, using replication rather than decomposing the program into smaller parallelizable computations.

6 Conclusion

It is now more crucial than ever to investigate techniques to scale packet processing using multiple cores. This paper presented state-compute replication (SCR), a principle that enables scaling the throughput of stateful packet-processing programs monotonically across cores by leveraging a packet history sequencer, even under realistic skewed packet traffic.

Acknowledgments

The work was supported in part by the U.S. National Science Foundation (NSF) grants CNS #1910796, CAREER #2340748, CNS #2008048, FMitF #2019302, and FMitF #2422076, the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, the partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"), and gifts from the Network Programming Initiative (NPI) and Xilinx. We thank our shepherd Luis Pedrosa for his thoughtful comments on the final versions of this paper and the anonymous NSDI reviewers for their revision feedback. We are grateful to David Walker, Jennifer Rexford, Sanjay Rao, and Santosh Nagarakatte for helpful feedback on previous versions of this paper. We also appreciate the support of the Rutgers Laboratory for Computer Science Research (LCSR) staff, Kevin Conover, Tim Hayes, and Chuck Hedrick, for their thoughtful support of research infrastructure.

References

[1] Intel IPU. [Online, Retrieved Feb 21, 2023.] https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[2] Linux Socket Filtering aka Berkeley Packet Filter (BPF). [Online, Retrieved Nov 05, 2020.] https://www.kernel.org/doc/Documentation/networking/filter.txt.

[3] NVIDIA BlueField DPU. [Online, Retrieved Feb 21, 2023.] https://www.nvidia.com/en-us/networking/products/data-processing-unit.

[4] Receive Side Scaling. [Online, Retrieved Feb 21, 2023.] https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[5] Benchmarking Methodology for Network Interconnect Devices. [Online, Retrieved Sep 17, 2023.] https://www.rfc-editor.org/rfc/rfc2544, 1999.

[6] IEEE 802.1Qbb – Priority-based Flow Control. [Online, Retrieved May 02, 2024.] https://1.ieee802.org/dcb/802-1qbb/, 2011.

[7] How to set up Intel Ethernet Flow Director. [Online, Retrieved Sep 17, 2023.] https://www.intel.com/content/www/us/en/developer/articles/training/setting-up-intel-ethernet-flow-director.html, 2017.

[8] Open-sourcing Katran, a scalable network load balancer. [Online, Retrieved Sep 17, 2023.] https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/, 2018.

[9] A kernel patch to support RSS++. [Online, Retrieved Apr 29, 2024.] https://github.com/rsspp/linux/commit/4e09cf8be6ac5b0a06cc5b92c62f758f29e3b6aa, 2019.

[10] Concurrency management in eBPF. [Online, Retrieved Sep 17, 2023.] https://lwn.net/Articles/779120/, 2019.

[11] The CAIDA UCSD Anonymized Internet Traces - 2019. [Online, Retrieved Sep 17, 2023.] https://www.caida.org/catalog/datasets/passive_dataset, 2019.

[12] NetFPGA-PLUS. [Online, Retrieved Sep 17, 2023.] https://github.com/NetFPGA/NetFPGA-PLUS, 2021.

[13] Nvidia ConnectX-6 DX. [Online, Retrieved Apr 14, 2024.] https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/ConnectX-6-Dx-Datasheet.pdf, 2021.

[14] Measuring BPF performance: Tips, tricks, and best practices. [Online, Retrieved Sep 17, 2023.] https://developers.redhat.com/articles/2022/06/22/measuring-bpf-performance-tips-tricks-and-best-practices, 2022.

[15] BPF helpers manual page. [Online, Retrieved Sep 17, 2023.] https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html, 2023.

[16] BPF maps. [Online, Retrieved Sep 17, 2023.] https://www.kernel.org/doc/html/latest/bpf/maps.html, 2023.

[17] Broadcom Trident 4: BCM56880 Series. [Online, Retrieved Jul 22, 2023.] https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2023.

[18] Built-in Functions for Memory Model Aware Atomic Operations. [Online, Retrieved Sep 17, 2023.] https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/_005f_005fatomic-Builtins.html, 2023.

[19] Checksum Offloads. [Online, Retrieved Sep 17, 2023.] https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html, 2023.

[20] ConnectX-7 400G adapters. [Online, Retrieved May 1, 2024.] https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet, 2023.

[21] Data Plane Development Kit. [Online, Retrieved Jul 22, 2023.] https://www.intel.com/content/www/us/en/developer/topic-technology/networking/dpdk.html, 2023.

[22] Intel Data Direct I/O technology. [Online, Retrieved Sep 17, 2023.] https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2023.

[23] Intel Tofino. [Online, Retrieved Jul 22, 2023.] https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2023.

[24] Introduction to Intel Performance Counter Monitor (PCM). [Online, Retrieved Sep 17, 2023.] https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html, 2023.

[25] Legacy __sync Built-in Functions for Atomic Memory Access. [Online, Retrieved Sep 17, 2023.] https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/_005f_005fsync-Builtins.html, 2023.

[26] Open Tofino: P4 Intel Tofino Native Architecture - Public version. [Online, Retrieved Sep 17, 2023.] https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, 2023.

[27] Pensando infrastructure accelerators. [Online, Retrieved Jul 22, 2023.] https://www.amd.com/en/accelerators/pensando, 2023.

[28] Port knocking. [Online, Retrieved Jul 22, 2023.] https://help.ubuntu.com/community/PortKnocking, 2023.

[29] Segmentation Offloads. [Online, Retrieved Sep 17, 2023.] https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html, 2023.

[30] TCP Offload Engine: Chelsio Communications. [Online, Retrieved Sep 17, 2023.] https://www.chelsio.com/nic/tcp-offload-engine/, 2023.

[31] SCR experimental scripts and software. [Online, Retrieved Oct 03, 2024.] https://github.com/smartnic/bpf-profile, 2024.

[32] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2022.

[33] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[34] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[35] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. Rss++: Load and state-aware receive side scaling. In *Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, 2019.

[36] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *Internet Measurement Conference (IMC)*, 2010.

[37] Daniel Borkmann and Martynas Pumputis. K8s Service Load Balancing with BPF & XDP. [Online. Retrieved Jan 23, 2021.] https://linuxplumbersconf.org/event/7/contributions/674/attachments/568/1002/plumbers_2020_cilium_load_balancer.pdf, 2020.

[38] Pat Bosshart et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM*, 2013.

[39] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: Disaggregating the host network stack. In *ACM SIGCOMM*, 2022.

[40] Arthur Chiao. Connection Tracking (conntrack): Design and Implementation Inside Linux Kernel. [Online, Retrieved Sep 17, 2023.] https://arthurchiao.art/blog/conntrack-design-and-implementation/, 2020.

[41] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NIC-Driven Tail-Aware Balancing of us-Scale RPCs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[42] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[43] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[44] Arthur Fabre. L4drop: Xdp ddos mitigations. [Online, Retrieved Jul 25, 2023.] https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/.

[45] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[46] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling

Innovation in Network Function Control. *ACM SIGCOMM*, 2014.

[47] Massimo Girondi, Marco Chiesa, and Tom Barbette. High-speed connection tracking in modern servers. In *IEEE International Conference on High Performance Switching and Routing (HPSR)*, 2021.

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

[49] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 2015.

[50] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[51] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[52] Jonathan Corbet. Accelerating networking with AF_XDP. [Online. Retrieved Jan 20, 2021.] https://lwn.net/Articles/750845/, 2018.

[53] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[54] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[55] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *ACM EuroSys*, 2019.

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

[57] Samuel Midkiff. *Automatic parallelization: an overview of fundamental compiler techniques*. Springer Synthesis Lectures on Computer Architecture, 2012.

[58] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-Parallel Finite-State Machines. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

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

[60] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM*, 2021.

[61] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[62] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated synthesis of adversarial workloads for network functions. In *ACM SIGCOMM*, 2018.

[63] Francisco Pereira, Fernando M.V. Ramos, and Luis Pedrosa. Automatic parallelization of software network functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[64] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[65] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[66] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling Scalability-Sensitive Speculative Parallelization for FSM Computations. In *International Conference on Supercomputing*, 2017.

[67] Quentin Monnet. Optimizing BPF: Smaller Programs for Greater Performance. [Online. Retrieved Jan 20, 2021.] https://www.netronome.com/blog/optimizing-bpf-smaller-programs-greater-performance/, 2020.

[68] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[69] Luigi Rizzo. Netmap: a novel framework for fast packet I/O. In *USENIX annual technical conference (ATC)*, 2012.

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

[71] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. A case for spraying packets in software middleboxes. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2018.

[72] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[73] Nikita V. Shirokov. XDP: 1.5 years in production. Evolution and lessons learned. In *Linux Plumbers Conference*, 2018.

[74] Shinae Woo and KyoungSoo Park. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. Technical report, KAIST, 2020.

[75] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *ACM SIGCOMM*, 2002.

Application	t	c_2	d	c_1
DDoS mitigator	126	13	101	25
Heavy hitter monitor	138	17	105	32
Token bucket policer	153	22	102	51
Port-knocking firewall	128	15	101	27
TCP connection tracking	140	39	71	69

Table 4: The throughput model parameters (in nanoseconds) for packet-processing applications we evaluated.

Appendixes

The appendixes include supporting material that has not been peer-reviewed.

A Predicting Throughput

This section provides a model that predicts the throughput of SCR (§3.2) for any number of cores, given latency measurements made with one and two cores at low load. We also check the agreement between the empirically-measured throughput and our model.

Suppose a system has k cores, where each core can dispatch a single packet in d cycles, and run a packet-processing program that computes over a single packet in $c = c_1 + (k-1) \times c_2$ cycles, where c_1 is the time required to process the current packet and c_2 the time for a state update using one item of packet history. The time c_2 is smaller than c_1, as the state transition is a fragment of the computation extracted from the program which processes one packet in entirety. For each piggybacked packet, the total processing time is $d + c_1 + (k-1) \times c_2$. We define $t \triangleq d + c_1$, the time to process one packet including the dispatch and the program computation. When t dominates the state-computation time (*i.e.,* $t \gg c_2$), with k cores, the total rate at which externally-arriving packets can be processed is $k \times \frac{1}{t+(k-1)\times c_2} \approx k/t$. Table 4 lists the parameters we measured for packet-processing applications we evaluated. It shows that $t \approx 3.6 - 9.9 \times c_2$. Hence, it is possible to scale the packet-processing rate linearly with the number of cores k.

We applied the parameters in Table 4 to the throughput model and compared the predicted throughput to the actual throughput. Figure 11 shows that they match well.

B Loss Recovery Algorithm

This section provides the detailed pseudocode and a proof of correctness of the packet loss recovery algorithm outlined in §3.4.

Before we start the correctness proof of loss recovery algorithm, we define the following notations.

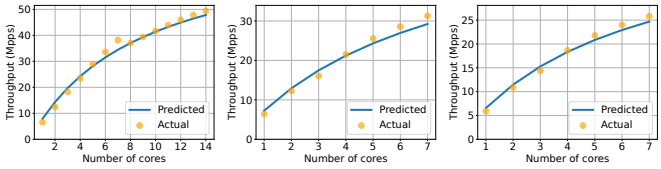
1. sp: an SCR packet. sp_i: the i^{th} SCR packet sent from the sequencer to a core (please refer to §3.3.1 for more details).

Algorithm 1 SCR loss recovery

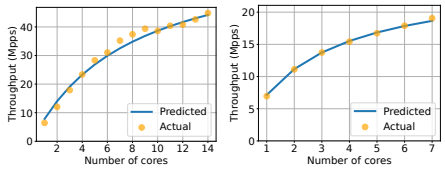
Input: the received SCR pkt (sp), current core (c)

```
1:  Initialize:
        max[c] ← 0, ∀i : log[c][i] ← NOT_INIT
2:  function scr_loss_recovery(sp, c)
3:      maxseq ← sp.seq_num
4:      minseq ← max(1, maxseq − N + 1)
5:      for i ← max[c] + 1 to maxseq do
6:          if i < minseq then
7:              log[c][i] ← LOST
8:              handle_loss_recovery(i, c)
9:          else
10:             history ← get history of i from sp
11:             log[c][i] ← history
12:         end if
13:     end for
14:     max[c] ← maxseq
15: end function

16: function handle_loss_recovery(i, c)
17:     C_others ← C \ {c}
18:     C_lost ← ∅
19:     while true do
20:         for each c′ ∈ C_others do
21:             if log[c′][i] is NOT_INIT then
22:                 continue
23:             else if log[c′][i] is LOST then
24:                 C_lost ← C_lost ∪ {c′}
25:                 if C_lost = C_others then return
26:                 end if
27:             else
28:                 history ← log[c′][i]
29:                 update state using history
30:                 return
31:             end if
32:         end for
33:     end while
34: end function
```



(a) DDoS mitigation (b) Heavy hitter detector (c) Token bucket policer



(d) Port-knocking firewall (e) TCP connection tracking

Figure 11: Predicted and actual throughput (§4.1) in millions of packets per second (Mpps) of five stateful packet-processing programs implemented using SCR (§3). The workloads of (a)-(d) and (e) are from a university data center and a hyperscalar data center (§4.1) separately.

2. p: a regular packet. p_i: the i^{th} regular packet received by the sequencer (in original or as part of the packet history, please refer to §3.3.1 for more details).
3. C: the collection of all cores.

We want to prove that no core will be deadlocked by loss recovery, *i.e.,* every core will *start processing* (execute line 6) and *finish processing* (finish executing line 6-12) every regular packet, under the assumptions that (1) each core will receive at least one SCR packet after packet loss, (2) we have infinite memory, and (3) packet number monotonically increases.

Theorem 1. *For any regular packet p_i, if every core has received p_j ($j \geq i$), every core will finish processing p_i.*

Proof. We will first prove that any core will process p_1 to p_i in order. The order of packets to process follows the order at line 5 (from $max[c]+1$ to $maxseq$), *i.e.,*, after a core finishes processing p_k, it will start processing p_{k+1}. Each core will be triggered to process all packets including p_1 to p_i, since each core has received p_j ($j \geq i$),

Given the order of packets to process, we now prove all cores can finish processing p_i. If $i = 1$, all cores will start processing p_1 (line 6) and then finish processing p_1 (Lemma 1). If $i > 1$, according to the order of a single core processing packets and Lemma 1, we get the induction hypothesis that if all cores have started processing p_k, all cores will finish processing p_k and start processing p_{k+1}. Using induction hypothesis for i times, all cores will start and then finish processing p_i. □

Lemma 1. *For any regular packet p_i, if all cores have started processing p_i, then all cores will finish processing p_i.*

Proof. For any core c, no matter p_i is lost or received, c will finish processing it.

If p_i is received by c, after c updates $history[i]$ in its log (line 10-11), c finishes processing p_i.

If p_i is lost at c (detected at line 6), c will wait for other cores to update p_i in their logs until c gets $history[i]$ or confirm p_i is lost at all the other cores (line 19-33). c will not be deadlocked in waiting, since p_i will be updated to $history[i]$ or *LOST* in the logs of all cores which have started processing p_i (if line 6 is executed for p_i, line 7 or line 11 will be executed). □

In a practical implementation, logs are finite and sequence numbers wrap around. We handle these concerns with a sufficiently large log and a sequence space. Our current implementation uses the values 1,024 and 842,185 for the aforementioned two quantities (respectively).

C SCR-Aware Multi-Core Programming

Consider a packet-processing program developed assuming single-threaded execution on a single CPU core. The question we tackle in this subsection is: how should the program be changed to take advantage of multi-core scaling with state-compute replication? We walk through the process of adapting a program written in the eBPF/XDP framework [50], but we believe it is conceptually similar to adapt programs written in other frameworks such as DPDK.

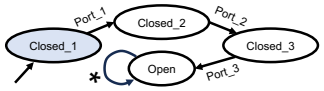


Figure 12: A state machine for a simple port-knocking firewall.

We describe the program transformations necessary for SCR through a running example. Suppose we have a port-knocking firewall [28] with the state machine shown in Figure 12. The program runs a copy of this state machine per source IP address. If a source transmits IPv4/TCP packets with the correct sequence of TCP destination ports, then all further communication is permitted from that source. All other packets are dropped. Any transition not shown in the figure leads to the default `CLOSED_1` state, and only the `OPEN` state permits packets to traverse the firewall successfully. A simplified XDP implementation of this single-threaded firewall is shown below.

```
/* Definition of program state */
struct map states {
  /* assume we define a dictionary with keys
     as source IP addresses and values as firewall
     states among CLOSED_{1, 2, 3} and OPEN. */
}

/* State transition function. See Figure 12  */
int get_new_state(int curr_state, int dport) {
  /* A function that implements the state machine for
     the port knocking firewall. */
  if (curr_state == CLOSED_1 && dport == PORT_1)
    return CLOSED_2;
```

```
  if (curr_state == CLOSED_2 && dport == PORT_2)
    return CLOSED_3;
  if (curr_state == CLOSED_3 && dport == PORT_3)
    return OPEN;
  if (curr_state == OPEN)
    return OPEN;
  return CLOSED_1;
}

/* The main function */
int simple_port_knocking(...) {
  /* Assume the packet is laid out as a byte array
     starting at the address pkt_start. Suppose the
     packet is long enough to include headers up to
     layer 4. First, parse IPv4/TCP pkts. */
  struct ethhdr* eth = pkt_start; // parse Ethernet
  int l3proto = eth->proto; // layer-3 protocol
  int off = sizeof(struct ethhdr);
  struct iphdr* iph = pkt_start + off;
  int l4proto = iph->protocol; // layer-4 protocol
  if (l3proto != IPv4 || l4proto != TCP)
      return XDP_DROP; // drop non IPv4/TCP pkts
  int srcip = iph->src; // source IP addr
  off += sizeof(struct iphdr);
  struct tcphdr* tcp = pkt_start + off;
  int dport = tcp->dport; // TCP dst port

  /* Extract & update firewall state for this src. */
  int state = map_lookup(states, srcip);
  int new_state = get_new_state(state, dport);
  map_update(states, srcip, new_state);

  /* Final packet verdict */
  if (new_state == OPEN)
      return XDP_TX; // allow traversal
  return XDP_DROP; // drop everything else
}
```

The program's state is a key-value dictionary mapping source IP addresses to an automaton state described in Figure 12. The function `get_new_state` implements the state transitions. The main function, `simple_port_knocking` first parses the input packet, dropping all packets other than IPv4/TCP packets. Then the program fetches the recorded state corresponding to the source IP on the packet, and performs the state transition corresponding to the TCP destination port. If the final state is `OPEN`, all subsequent packets of that source IP may traverse the firewall to the other side. All other packets are dropped.

To enable this program to use state-compute replication across cores, this program should be transformed in the following ways. We believe that these transformations may be automated by developing suitable compiler passes, but we have not yet developed such a compiler.

(1) *Define per-core state data structures and per-packet metadata structures.* First, the program's state must be replicated across cores. To achieve this, we must define per-core state data structures that are identical to the global state data structures, except that they are not shared among CPU cores. Packet-processing frameworks provide APIs to define such per-core data structures [16].

Additionally, we must define a per-packet metadata structure that includes any part of the packet that is used by the program—through either control or data flow—to update the state corresponding to that packet. For the port-knocking firewall, the per-packet metadata should include the `l3proto`, `l4proto`, `srcip`, and `dport`.

The data structures that maintain packet history on the sequencer correspond to this per-packet metadata (§3.3).

(2) Fast-forward the state machine using the packet history. The SCR-aware program must prepend a loop to "catch up" the state machine for each packet missed by the CPU core where the current packet is being processed. By leveraging the recent history piggybacked on each packet, at the end of this loop, the CPU core has the most up-to-the-packet state.

```
/* Assume the pointer 'data' locates where the
   per-pkt metadata begins in the byte array of
   the packet (Figure 4a ). Suppose 'index'
   is the offset of the earliest packet §3.3.2 ,
   and NUM_META is the number of packets in the
   piggybacked history.
*/
int l3proto, l4proto, srcip, dport, i, j;
for (j = 0; j < NUM_META; j++) {
  i = (index + j) % NUM_META; // ring buffer
  struct meta *pkt = data + i * sizeof(meta);
  l3proto = pkt->l3proto;
  l4proto = pkt->l4proto;
  srcip   = pkt->srcip;
  dport   = pkt->dport;
  if (l3proto != IPv4 || l4proto != TCP)
     continue; // no state txns or pkt verdicts
  /* Update state for this srcip and dport: */
  /* map_lookup; get_new_state; map_update. */
  /* Note: No pkt verdicts for historic pkts. */
}
pkt_start = data + NUM_META * sizeof(struct meta)
              + sizeof(index);
```

A few salient points about the code fragment above. First, the semantics of the ring buffer of packet history (§3.3) are implemented by looping over the packet history metadata starting at offset `index` rather than at offset 0. The decision to implement the ring buffer semantics in software makes the hardware significantly easier to design, since only a small part of the hardware data structure needs to be updated for each packet (§3.3.2). Second, the loop must implement appropriate control flow before the state update to ensure that only packets that should indeed update the flow state do. Note that the metadata includes parts of the packet that are not only the data dependencies for the state transition (`srcip, dport`) but also the control dependencies (`l3proto, l4proto`). Third, no packet verdicts are given out for packets in the history: we want the program to return a judgment for the "current"

packet, not the historic packets used merely to fast-forward the state machines. Finally, the code fragment conveniently adjusts `pkt_start` to the position in the packet buffer (Figure 4a) corresponding to where the "original" packet begins. The rest of the original program—unmodified—may process this packet to completion and assign a verdict.

What is excluded in our code transformations is also crucial. This program avoids locking and explicit synchronization, despite the fact that it runs on many cores, even if there is global state maintained across all packets.

With these transformations, in principle, a packet-processing program is able to scale its performance using state-compute replication across multiple cores.

MTP: Transport for In-Network Computing

Tao Ji
UT Austin

Rohan Vardekar
University of Illinois Chicago

Balajee Vamanan
University of Illinois Chicago

Brent E. Stephens
Google and University of Utah

Aditya Akella
UT Austin

Abstract

In-network computing (INC) is being increasingly adopted to accelerate applications by offloading part of the applications' computation to network devices. Such application-specific (L7) offloads have several attributes that the transport protocol must work with—they may mutate, intercept, reorder and delay application messages that span multiple packets. At the same time the transport must also work with the buffering and computation constraints of network devices hosting the L7 offloads. Existing transports and alternative approaches fall short in these regards. Therefore, we present MTP, the first transport to natively support INC. MTP is built around two major components: 1) a novel message-oriented reliability protocol and 2) a resource-specific congestion control framework. We implement a full-fledged prototype of MTP based on DPDK. We show the efficacy of MTP in a testbed with a real INC application as well as with comprehensive microbenchmarks and large-scale simulations.

1 Introduction

Datacenters today are experiencing ever-increasing demand for computing with the emerging of large-scale distributed applications and intensive workloads such as ML/AI and VR/AR. These applications run heavy computational logic and rely on infrastructure services (e.g., for caching and load-balancing) that themselves impose non-trivial computational needs. In-network computing (INC) has emerged as a promising approach to meet the resulting compute demand by leveraging specialized in-network hardware for offloading. With INC, application-specific and infrastructure processing functions—which we together refer to as L7 functions—can be run atop network devices. This can provide vastly improved application throughput and latency compared to approaches that rely on conventional server architectures [65].

We have seen a blossoming of ideas for offloaded L7 functions (or "L7 offloads"). Examples include key-value caches [35, 46, 75], file and storage systems [41, 48, 59], machine learning (ML) tasks [45, 55, 69], transaction and database systems [32, 34, 70], consensus [47], as well as load balancing [17, 81]. Alongside, we have seen a growing number of devices and platforms that can support INC, including programmable switches [1–3, 12, 40, 78, 79], and SmartNICs [10, 53, 54, 64]. While the growing number of showcases for INC is promising, we argue that there are fundamental problems that need to be addressed before INC can be utilized in a systematic and effective manner to accelerate datacenter computation. This paper focuses on one such problem, namely the design of an INC-compatible transport protocol.

Despite the wide diversity of INC possibilities and platforms, we argue that all of them need in common a transport protocol with two properties:

(1) The transport's basic functions—reliability and congestion control—should work in the face of key attributes of INC, namely that the L7 offloads can induce mutation, intercept, reordering and arbitrary delays upon application messages. For example, an HTTP load balancer can insert cookies to an HTTP request [17], and an in-network aggregator can delay producing the result until the straggler provides the input [78]. We detail all such attributes in §2.1.

(2) The transport should not impose additional transport-specific resource needs on INC hardware (e.g., needing significant buffering or state maintenance). Switches and SmartNICs that host INC have limited memory and compute power.

Most datacenters today use either TCP variants or RDMA as their transport [29, 72, 82], and new experimental transports have emerged, such as the receiver-driven Homa protocol [61]. Unfortunately, none of these transport protocols can accommodate the attributes of INC. When used alongside INC, they result in either broken reliability semantics, poor and ineffective congestion control, or both (§2.2). Alternative approaches either have prohibitive compute and memory overheads or are ad-hoc, eschewing reliability and/or congestion control while only supporting a subset of the INC attributes (§2.3), and hence are not widely applicable.

This paper presents MTP (Message Transport Protocol), the first INC-compatible transport protocol. It makes key design choices that elevate L7 offloads—how they function and how

they are used in datacenter networks—to first-class entities from a transport perspective. MTP works on *messages* as this aligns with the processing semantics of L7 offloads—all key decisions in MTP, be they for loss detection/recovery, congestion control or path selection, operate at the message granularity. To handle message mutation, intercept and reordering, MTP decouples key transport actions, e.g., loss detection and recovery, from the packet or byte sequence numbers (§4).

MTP recognizes and operates on *pathlets*, groups of offloads that share fate. For example, a pathlet can be either a single instance of an offload or multiple replicas of an offload that a NIC load balances across. MTP elicits *per-pathlet feedback* both for MTP's reliability protocol to set appropriate timeouts when messages are delayed by pathlets rather than dropped (§5) and for congestion control to take per-message decisions on when to transmit and which pathlets to use. This controls the load on pathlets, avoiding the overloaded ones and causing load imbalance (§6).

Additionally, using MTP has low overheads: pathlets need not maintain MTP-specific state and can easily leverage MTP primitives by generating appropriate ACK packets (§4–6). The endpoint-maintained state is constant and proportional to (an upper-bounded) number of in-flight messages and pathlets in use (Appendix B). The MTP header is only 20 B plus optionally a 5 B pathlet congestion feedback (Appendix A), and the ACK primitives have low bandwidth and endpoint processing overheads (§8.2.3).

To evaluate MTP, we implement a full-fledged MTP prototype based on DPDK [23] and perform end-to-end benchmarks by running on top of MTP NetCache [35], a real-world INC application. We show that MTP improves the throughput by over 15% (§8.1). We conduct comprehensive microbenchmarks (§8.2) showing MTP's effective loss recovery and congestion control in the face of INC. We also simulate MTP's operation at large scales (§8.3) and find that MTP achieves 65% less tail latency and can sustain 10–20% higher loads compared to TCP under heavy message reordering.

2 Motivation

In this section, we argue that a novel transport for INC is warranted. For that, we capture the essential message operations that INC performs (§2.1), describe how existing transports are not compatible with these operations (§2.2), discuss how the ways in which INC is made to work with transports today are not widely applicable (§2.3), and extract the requirements of a novel transport (§2.4).

2.1 On-Path Message Processing

INC enables applications to offload portions of their (L7) logic to NICs or switches. Traditional offloads, such as network address translation and transmit segmentation offload, have dedicated network- or transport-level logic and process the corresponding protocol headers in packets. In contrast, L7 offloads process application-level (L7) messages, which are the payloads that span one or more network packets. L7 offloads have more diverse functions. Examples include various key-value caches [35, 46, 49], intrusion detection/prevention [80], L7 load balancing [17, 81], transaction and database acceleration [32, 34, 70], as well as in-network aggregation (INA) for machine learning jobs [45, 55, 69, 78].

These offloads are "on-path": located on the communication path between end hosts, and multiple offloads that perform different processing (e.g., load balancing and intrusion detection) or different parts of the same processing (e.g., two levels of INA [45]) can form a "chain" on the path.

However, the operations that these offloads perform on L7 messages can often interact poorly with the underlying transport, potentially causing low performance and even breaking the transport protocol. We summarize the operations below and discuss their impact on transport in the next section.

Mutation. An offload can change the L7 message payload. At times this can result in changes in the message size and the number of packets that the message spans. For example, an HTTP load balancer [17] can inject cookies into HTTP responses, which makes the response messages longer. The in-network aggregator of SwitchML [69] replaces the per-worker gradients in the RDMA message with aggregated gradients.

Intercept. In some scenarios, an offload can explicitly drop a message so that it does not reach the transport's receiving endpoint. Some RPC load balancers, for example, can proactively drop the requests that will anyway violate the service-level objectives [42].

Reordering. An offload can reorder messages between the same sender and receiver for two reasons. First, the offload explicitly does so as part of its logic. Transaction triaging, for example, reorders messages to batch similar transactions, which improves server performance [34]. Second, many hardware architectures support running replicated instances of the same offload logic in parallel to improve performance [53, 54, 78]. Messages that take different amounts of processing time at various instances can finish in a different order.

Delaying. An offload can impose delays on a message in various cases, and such delays can be long and unpredictable. (1) Complex INC logic or wimpy hardware: FPGA-based intrusion detection, for instance, can incur a tail processing delay of ∼4x median for infrequent slow-path processing [80]. Some wimpy SmartNICs can even incur ∼80μs tail delay (∼2x average) on packet steering [54]. These delays are of the same order of magnitude as the network fabric delay (10s of μs [43]). Such offloads themselves are often bottlenecks and when experiencing high load may incur increased queuing delay. (2) In some cases, the processing of a message cannot complete if another message is delayed. E.g., in INA, gradients cannot be completely aggregated if a straggler worker delays its part, and mitigating a straggler can take 10s of ms, which is much higher than the usual aggregation [78] and network fabric delay.

2.2 Today's Transports Are Incompatible

We discuss three transports, i.e., TCP (including its variants such as QUIC [44]), RDMA [29, 82], and Homa [61]. All of them provide reliability and congestion control, which are essential transport-layer functions. TCP variants and RDMA are arguably the most widely adopted transports in production datacenters. Homa is an emerging receiver-driven transport providing message transfer semantics with efforts to deploy it in production [31, 63]. Unfortunately, these transports are largely incompatible with INC operations which can have a fundamental impact on the correctness and efficacy of the transports' reliability and congestion control.

The TCP variants and RDMA do not work in the face of mutation, intercept, or reordering. Consider these transports' reliable delivery: they both use a per-connection continuous byte/packet sequence number space, and all the bytes/packets transmitted by the sender must be acknowledged by the receiver. When an offload mutates a multi-packet message in a way that changes its length or intercepts it, the byte and/or packet counts that reach and are ACK'd by the receiver also change. In that case, the sender cannot correctly decide if a missing byte/packet needs retransmission, and can be confused if ACK'd bytes/packets are more than what it has transmitted. Also, reordering can cause severe out-of-order packet arrivals resulting in discontinuous received sequence, which both protocols can interpret as loss and trigger wasteful retransmission as well as reduction in congestion window.

Homa cannot properly perform loss recovery with mutation. Homa's receiver detects missing bytes with a timeout and asks the sender to retransmit the missing part. However, if the message is mutated to a different length, the range of bytes in the mutated message that the receiver asks for is likely not the same bytes in the original message. Note that Homa could work with intercept and reordering because the transmission of each message is independently controlled, and there is no cross-message state such as a sequence number that can be disrupted by dropping or reordering messages.

Finally, delaying can also disrupt congestion control in fundamental ways in all three protocols. TCP and RDMA rely on some end-to-end measurements such as ECN [8, 82] and RTT [43, 60] to quickly and accurately detect congestion. These measurements traverse the forward path and are eventually reflected back to the sender by the receiver. When messages can be delayed at an offload for unpredictably long, the measurement might not arrive at the sender in a timely manner, which prevents the sender from quickly reacting to congestion. To achieve line rate and low queuing, Homa's receiver tries to keep 1 BDP (bandwidth-delay product) of credits in flight. However, this assumes that both network RTT and bandwidth is known and constant. With INC, message mutation that causes changes in length, below-line rate bottleneck offloads and unpredictable delays can make it extremely difficult to accurately control the outstanding credits.

2.3 How INC Works With Transports Today

Some existing INC works have made simplifying assumptions or workarounds so that transports can correctly function with a specific offload. However, such approaches have important limitations.

One approach to be compatible with INC operations is for an offload to "terminate" the transport, running both the receiver- and sender-side protocol logic [28]. However, transport stacks need significant hardware resources on the offload for transport processing and message buffering. The hardware limitations of some INC platforms like switches preclude running transports altogether [55, 69], while the limitations of others like SmartNICs may limit scalablity and require careful parallel decomposition and many cores for performance [71].

Furthermore, the simplifying assumptions made by prior work are also usually unrealistic. For example, prior work has assumed that an RPC's length never exceeds one MTU (maximum transmission unit) and hence can always be carried by just one packet [35,46,49,53,64]. Other work has assumed that a new connection can be established for each RPC [81]. However, these don't generally hold: A significant proportion of RPCs in datacenters are indeed larger than the MTU [8,61], and RPCs typically reuse connections because connection establishment incurs overhead [4].

Some works have proposed workarounds, which usually cover only the subset of message operations required by the specific offload and are not generalizable. As a result, these solutions require case-by-case human efforts to apply to other use cases. For example, NetReduce [55] accommodates mutation in a way that is transparent to TCP and RDMA, but the message size must not change. As such, this approach is not compatible with an HTTP load balancer that adds cookies. Additionally, this approach does not support intercept or reordering. SwitchML [69] and ATP [45] build their own transport functions such as reliable delivery based on UDP, but neither work in the face of intercept. The transaction triaging offload [34] supports intercept and reordering but does not work amidst mutation of the message (transaction) body.

Others have relied on specific hardware architectures. The HTTP load balancer [17], for example, accommodates mutation that changes the message length based on NICs that have general-purpose cores. However, it must maintain per-connection state that correctly maps between the altered and original byte sequences, and it must buffer the mutated messages for retransmission. Maintaining such per-connection state can be expensive in hardware, and buffering payload might not be supported by some hardware architectures.

2.4 Design Requirements

We argue for a novel transport that is widely applicable to INC. Such a transport must meet these requirements:

- Provide correct reliable delivery in all scenarios where the packets or bytes are altered in the network as a result

of message mutation, intercept, or reordering;

- Provide effective congestion control for all potential bottlenecks in the network, including the network fabric and slow offloads, despite the delaying of messages.
- Support all kinds of INC hardware without assuming that special transport-specific state or buffering can be maintained by the offload.

We present MTP (Message Transport Protocol), the first transport designed for INC, satisfying all of these requirements.

3 Pathlets

Before describing MTP, we provide a brief overview of pathlets, which is a key building block. Then, we discuss the infrastructure support needed to allow MTP to leverage pathlets and our assumptions on how pathlets work.

3.1 Pathlets and Grouping

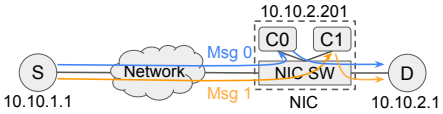
We observe that on the path of a message, between the transport sender and receiver, there can be multiple offloads, potentially with different types of computation. For example, an L7 load balancer [17, 81] can be followed by intrusion detection [80], if the requests can come from untrusted senders. In a two-level INA deployment, a message can go through two aggregators on different switches [45].

We refer to each such offload instance on the path as a pathlet [1]. Pathlets that perform the same computation are said to have the same pathlet type. An ordered chain of pathlet types defines the computation that the message goes through in the network.

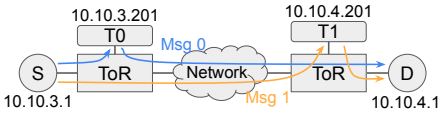
Instances of a pathlet type can be arranged in two forms. First, as shown in Figure 1b, offload instances can be at different network locations; in this case, as we describe below, MTP allows a sender to select which specific pathlet instance to use. Second, as shown in Figure 1a, the instances are co-located on the same NIC or switch, and an on-NIC/switch scheduler locally balances their loads [53]. Thus, these should not be viewed as individual pathlets as the scheduler hides internal load imbalance or partial failure. MTP groups the latter such instances together and views them as one pathlet.[2]

3.2 Pathlet Discovery and Routing

For the MTP transport to work with pathlets, the network must provide two services: (1) Pathlet discovery: The datacenter network (DCN) should determine the chain of pathlet types to be used for messages towards a given receiver, and provide for each type the specific pathlet instances for the transport to use. (2) Pathlet routing: The network should route the packets of a message to the transport-selected pathlets in the same order. We now describe how these services can be provided in an MTP-enabled DCN.



(a) Locally load-balanced offloads C0 and C1.



(b) Non-locally deployed offloads T0 and T1.

Figure 1: **Possible deployment models for offload instances that perform the same computation.**

Discovery. We assume that the applications that leverage INC capabilities follow the client-server model. When a client attempts to reach a service, it first queries for the server(s) with a service discovery protocol (SDP), such as [5, 6]. The SDP provides, in addition to server addresses, a list of pathlet types that client messages should go through toward the service. The client then queries a pathlet registry for the available instances of each pathlet type and the address of each instance. The replied pathlet information for the service is cached in the client host hypervisor or OS for fast retrieval.

This pathlet registry is similar to a service registry that would back a SDP today. It can be scaled up with standard sharding techniques that are used for service registries, such as employing a distributed key-value store [68, 76].

Routing. To reach the pathlets, we employ a form of source routing. When a client connects to a service, it configures MTP with the server and pathlet addresses that it queries from the SDP. When sending messages, MTP encodes the list of pathlet addresses it selects in the packet header, and the network forwards the packets through the hops with given addresses in order, and eventually to the server. This can be achieved with IPv6 segment routing [22] on commodity off-the-shelf network devices. We assume that each pathlet has an IPv6 address.

3.3 Pathlet Processing

We observe that there can be two ways in which the pathlet gathers data to process: 1) *full buffering*, where the pathlet buffers messages in full, and the processing can only be done when the whole message has arrived at the pathlet; and 2) *streaming*, where the pathlet starts processing with partial messages, and can receive and transmit packets of the message while processing. Full buffering can utilize *cut-through* forwarding, so the biggest difference between streaming and full buffering is in the buffering requirements.

We design MTP to cater for full buffering, which we argue is the most common case, for two reasons. First, applica-

[1] While inspired by prior art on "pathlet routing" [26] where a pathlet is a link or set of links, we use the term "pathlet" differently.

[2] A datacenter operator can conservatively decide to have just one instance for each pathlet type as default. However, if the operator chooses to expose multiple pathlets, MTP can offer improved performance.

tions can often break down large data into packet-sized messages for offload processing. For example, SwitchML [69] and ATP [45] encapsulate partial gradients in single-packet messages to fit in the programmable switch, each message with an L7 header as application context. Also, for many applications in datacenters, messages are small, with over 90% messages each fitting in one Ethernet jumbo frame, and many offloads also assume single-packet messages [35, 46, 51, 81].

Second, it is usually an application requirement or choice to process larger messages in whole, in which case the pathlet should already have the capability to achieve full buffering of multi-packet messages. For instance, NetReduce [55] works with large messages that contain ML gradients by relying on the transport to chunk the message, and the FPGA-based aggregator can reassemble the message based on transport sequence numbers.

We note that some applications employ limited-capability offload hardware, such as Tofino [3], where it is hard to perform processing that involves maintaining cross-packet (message) state. MTP is still applicable: such offloads can leverage MTP's reliability and congestion control primitives by simply generating appropriate ACK packets (detailed later) upon receiving a "single-packet message".

We discuss how MTP can be used as a building block to support streaming applications in §9.

4 Overview

We now present MTP, starting with an overview of its architectural components and a basic end-to-end workflow.

4.1 MTP Architecture

MTP consists of two major components: (1) a message-oriented reliability protocol that natively accommodates message mutation, intercept and reordering; and (2) a multi-pathlet congestion control framework that enables congestion control algorithms to operate effectively in the face of pathlets causing long and unpredictable delays. Neither component requires additional state at pathlets. As such, MTP meets all of the requirements in §2.4.

To design such a reliability protocol, we make a key observation about why existing transports fall short (§2.2): The receiver cannot tell whether a missing place in the received packet or byte sequence is because of loss or intercept/reordering; nor can it, in case of mutation, provide the before-mutation sequence number for the sender to retransmit. Our design choice is therefore to rely on the sender to detect losses with *per-message retransmission timeouts (RTOs)*: The receiver, upon the arrival of all packets of the message, sends an end-to-end (E2E) ACK, which disarms the timer. Otherwise the timer triggers retransmission of the whole message. We discuss the challenges that come with this design in §5.1.

To enable effective congestion control (CC) under long and unpredictable delay caused by INC, our framework provides two primitives. (1) Per-pathlet early congestion feedback: A pathlet has the choice of communicating its own congestion state, as well as reflecting any congestion signal along the path (such as ECN for link congestion), directly back to the sender in the form of a *dedicated packet*, that does not incur additional delay from the offload processing. (2) Per-message pathlet decision: If multiple same-type pathlets are available, the congestion control can *decide the instance to use on a per-message basis*. This allows quickly steering messages away from the congestion point. We show how these primitives can be leveraged to build effective congestion control in §6.

4.2 Basic Workflow

We now walk through an example (Figure 2) of end-to-end MTP message delivery with in-network mutation.

MTP is connection-based. Each connection endpoint is bound to an IP address and a port. A flow is defined by the 5-tuple as in TCP. We focus on transport operations over an established connection. MTP's connection establishment is achieved with a three-way handshake similar to TCP.

Before sending the first message, the sender queries the service and pathlet discovery protocols for pathlets, and CC remembers these pathlets. To send a message, the application posts to the MTP stack a message descriptor, which carries metadata such as the length and location of the message body. The MTP stack assigns a message number to the descriptor, which increments in the order of descriptors polled, and stores the descriptor in the connection state. CC then selects the pathlets to use for this and subsequent messages and stores the pathlet addresses in the descriptor.

Once CC determines that this message is admissible into the network because there is sufficient room in the congestion window, the MTP stack divides the message body into segments each up to the MSS (maximum segment size, 4096 bytes in this example). A message smaller than the MSS is sent in one segment of the message length. Each segment is identified by a segment number unique within the message. Then MTP encapsulates the pathlet addresses and segments into packets and transmits them.

Upon receiving the segment, pathlet A *immediately* generates a congestion feedback that potentially includes its own congestion signal and link congestion signal such as ECN. We explain how the feedback values are decided in §6. Pathlet A then performs message mutation that increases the message size by 8 bytes so that the message now consists of two segments.

When receiving the first arrived segment of a message, the receiver creates in its connection state a message descriptor that keeps track of the received segments of a message with a bit mask of the received segment numbers. To enable the receiver to determine when all segments are received, every segment packet carries the message length, which divided by MSS gives the number of segments in the message. Once all segments have arrived, the receiver sends an ACK for the

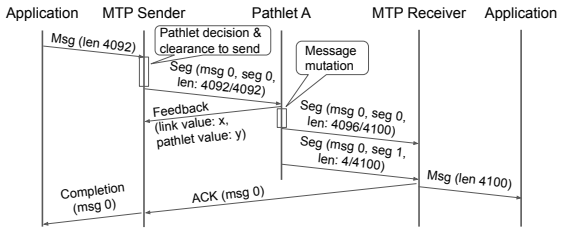


Figure 2: **The MTP basic workflow with mutation.**

(now mutated) message to the sender host. The MTP stack then provides the message to the receiver application and notifies the sender application of completion.

This example shows how MTP works with mutation. Note that the receiver is "passive": It only sends out the E2E ACK for fully received messages and does not check the continuity or order of arrived message numbers. Also, the receiver does not act on out-of-order arrived packets within a message, and can create the message descriptor based on any segment packet of a message that arrives first. Therefore, no special accommodation is needed for reordered messages and even reordered packets, which can be a result from fine-grained network load balancing [20]. However, such a passive receiver design makes it challenging to provide an exactly-once guarantee, which we discuss in §5.2.

By design, MTP does not guarantee in-order message delivery, since this is known to cause head-of-line blocking between messages, which hurts latency-sensitive applications [74]. Applications that require ordering can achieve this by adding their own ordering mechanism [74].

In MTP, a pathlet is expected to output at most one message for each incoming message. The pathlet can intercept a message by sending an E2E ACK to the sender as if the message was delivered to the receiver. Otherwise, the pathlet must forward the message towards the next pathlet or receiver. Any additional payload generated should be added to a message in the form of mutation. We leave the support for branching pathlets in MTP, i.e., multicast, to future work.

5 Reliability

In this section, we discuss the challenges to MTP's reliability protocol that arise from the design choices that we have made, and how we address them.

5.1 Delays and RTOs in MTP

We have previously made the design choice to rely on the sender to detect losses with per-message RTO. In a legacy DCN, relying on a constant RTO for loss detection is acceptable because the network fabric delay is bounded and the RTT is predictable, and the operator can configure a short RTO [9, 61]. However, as discussed in §2.1, an offload can incur a long and unpredictable delay that can be much higher than the fabric delay. Setting a long RTO for the worst-case scenario hurts the quickness of loss recovery, whereas a short RTO causes spurious retransmissions (§8.2.1).

To resolve this, our idea is to distinguish whether the message is in the network or in a pathlet that can incur long or unpredictable delays. With this knowledge, the sender can apply *different RTOs* that fit the respective cases. MTP achieves this with two dedicated types of ACKs as follows.

Pathlet ACKs. A pathlet can leverage two types of ACKs to notify the sender of the arrival and departure of a message, allowing the sender to apply different RTOs. When the pathlet receives all message segments, it sends a pathlet receive ACK, or *PRX ACK*, to the sender to indicate that the whole message is buffered. Until further notice, the sender assumes that the message is being processed, and does not retransmit until the long RTO (*"pathlet RTO"*) is triggered. When the pathlet finishes message processing and sends all the segments out, it sends a pathlet transmit ACK, or *PTX ACK*, to the sender, indicating that the message has left the pathlet. Upon PTX ACK, the sender restarts the short RTO timer for timely retransmission for fabric drops.

Loss of PRX and PTX ACKs does not impact correctness, since the sender always maintains one RTO timer and will retransmit upon timeout until eventually getting the E2E ACK for the message. We prioritize PTX and PRX ACKs, as well as the E2E ACK in the network so that they are rarely dropped.

Configuring the RTOs. With the pathlet ACKs, the fabric RTO can be set to the fabric RTT between the furthest hosts in the network under heavy congestion (i.e., a few hundred μs [43]), plus the message serialization delay, to prevent false positives, because such delay is nearly the worst case between the sender/any pathlet and the next pathlet/receiver. For generally underutilized networks or latency-sensitive messages, the fabric RTO can be set shorter but this risks spurious retransmissions under sudden congestion.

The pathlet RTO should be set to cover the processing delay and the buffering time under pathlet congestion, which is known to the pathlet owner and distributed with the SDP (§3.2), plus the message serialization delay. In practice, though, assuming the message is seldom lost once buffered by a pathlet, the pathlet RTO can be set loosely.

5.2 Exactly-Once Guarantee

Most applications are programmed under the assumption that the transport delivers a message to the receiving application exactly once, and it is important that MTP provides the same semantics. However, MTP's receiver design (§4.2) so far does not guarantee this. Suppose a message is successfully delivered to the receiving application, the MTP receiver removes the message descriptor from the connection state, and the E2E ACK is then dropped; the sender times out and retransmits the whole message. When the duplicate message arrives, the MTP receiver cannot realize that the message is duplicated, and will deliver it to the application again.

Challenge. For the receiver to detect and drop duplicate messages, it must track some state about the delivered messages.

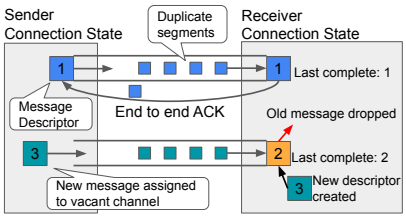


Figure 3: **Workflow of virtual channels.**

The approach must only occupy constant space so as not to exacerbate the connection state scalability issue in datacenters [37].

A strawman is to leverage a fixed-sized window with the lower bound representing the highest message number below which all messages are received and use a bitmap to represent message numbers received within the window. However, when messages can be heavily reordered like in an INC-enabled DCN, a slow-arriving head-of-line (HoL) message can block the window's stepping forward, preventing messages beyond the window from being processed.

Virtual Channels. We devise a scheme called virtual channels[3] to track the delivered messages, which uses constant per-connection state, and is tolerant to reordering. A virtual channel (Figure 3) essentially represents the correspondence between the sender-side state of a message and its receiver-side counterpart. Each inflight message uses a separate virtual channel, and each connection maintains a fixed number of virtual channels (the figure shows 2).

This scheme prevents the delivery of duplicate messages to the application. Each virtual channel maintains the last message number fully received. Suppose the E2E ACK from the receiver is lost, the sender retransmits the message (upper channel in the figure). Note that the message remains in the channel upon RTO. The receiver checks the message number specified in the packets against the last message number. If they are the same, or the just-received segments bear a smaller message number, then the just-received segments are duplicates of the last fully received message. The receiver drops them and retransmits the E2E ACK. On receiving the E2E ACK the sender removes the message state from the channel, which then becomes vacant.

When a new message comes from the application, the sender assigns it to a vacant virtual channel (lower channel in the figure). The virtual channel starts transmitting segments when CC provides the pathlets to use and clearance to send. When the receiver obtains the first arrived segment of the new message, it finds that a new message has arrived because the packet bears a greater message number than its last message number for this virtual channel, then it removes the state of the last message and creates the state of the new message.

This scheme can tolerate high degrees of reordering because each virtual channel independently performs message transfer, and the connection makes progress as long as one channel does. The number of virtual channels determines the maximum number of inflight messages and is communicated during connection establishment. Generally, this number should be set to allow at least one BDP of messages in flight to sustain the line rate. For example, with 4KB messages and a network of 40μs RTT (including the usual processing time of offloads) and 100Gbps line rate, $\sim$122 virtual channels should achieve the line rate.

[3]We use this term differently from existing works where it refers to the multiplexing of the physical interconnect in a computer [18, 39].

6 Congestion Control

MTP provides for congestion control algorithms (CCAs) a framework that consists of two mechanisms, per-pathlet early congestion feedback and per-message pathlet selection (§4.1). We now demonstrate how one can build an effective congestion control solution for INC-enabled DCNs combining these mechanisms with existing ideas for congestion control [8, 43, 67, 77].

6.1 Using MTP's Congestion Feedback

To leverage MTP's per-pathlet congestion feedback, it is important to unify the semantics in which different pathlets express their congestion conditions: Unlike network links, pathlets have vastly different implementation details and performance characteristics, based on which programmers may tend to expose distinct congestion information for each pathlet; however, doing so would require many different CCAs to interpret all the feedback, which is impractical. Also, mixing different CCAs for different pathlets on a path may cause interference among the CCAs.

To this end, we observe that in an INC-enabled DCN, each offload must maintain some sort of logical buffer or queue to temporarily store the pending workload for work conservation, and the buildup of the buffer or queue indicates offload congestion. Therefore, we can use 8 bits to map the pathlet queue or buffer occupancy to 256 quantiles. This is extending the existing idea of 2-bit queuing encoding [77]. The 8-bit encoding is then communicated to the sender with MTP's dedicated congestion feedback packet.

We then employ the Swift CCA to act on the pathlet feedbacks, inspired by how it has been used for endpoint congestion [14]. Swift [43] is a datacenter CCA that is highly effective, achieving very low loss rates at high loads, and deployed in production. It naturally fits because it reacts on RTT measurements, which are multi-bit indirect proxies of the queuing condition that MTP now directly provides, and because it is the state-of-the-art CCA based on multi-bit feedback. Our adapted algorithm essentially adjusts the congestion window (`cwnd`) based on the buffer occupancy feedback to maintain the occupancy at a moderate target level to achieve work conservation without overflowing, which is detailed in Appendix C with parameters we adopt in evaluation.

MTP maintains per-pathlet `cwnd`s and updates them in-

dependently to avoid conflicting feedbacks from different pathlets. It also uses a separate `cwnd` for link congestion. The receiver or next pathlet that provides feedback immediately reflects a marked CE bit to the sender in feedback packets. A message is only "cleared to be sent" when all pathlet `cwnds` and the link `cwnd` have sufficient room.

6.2 Proactive Pathlet Switching

So far we have assumed that all messages of a flow use the same sequence of pathlets. However, recall that the network can provide alternative instances of the same pathlet type (§3). It has been shown that mapping a flow to a fixed set of network resources, though the resources might be randomly chosen with approaches like ECMP, can result in hotspots [67], where many flows are mapped to the same resource, leaving alternative resources underutilized. In an INC-enabled DCN, pathlets are also resources that can become hotspots.

Therefore, it is important to enhance the CCA with the capability to resolve hotspots, which may further improve resource utilization. To that end, we complement our CCA with a mechanism called *proactive pathlet switching (PPS)*. The idea, inspired by PLB [67], is simple: When congestion at a pathlet cannot be quickly resolved by the CCA, which implies that the resource is likely a hotspot, the flow should be steered to use an alternative pathlet. Specifically, PPS monitors the congestion state of pathlets: If congestion causes the CCA to fail to decrease the buffer occupancy reflected in the multi-bit feedback of a pathlet below a threshold for some time, then PPS randomly chooses a pathlet from all the same-type pathlets provided by the DCN for future messages (Line 14 in Algorithm 1 in Appendix C). This is enabled by per-message pathlet selection.

7 Implementation

Our implementation has two types of components: an MTP stack that runs on end-hosts, and wrappers for making L7 offlaods MTP-compatible.

MTP Stack: We used DPDK [23] to implement an MTP stack prototype for testbed experiments. The prototype adopts a queue pair (QP)-based API (resembling RDMA's verbs [57]) for applications to communicate with the MTP stack. The MTP stack prototype is a process, similar to the Snap userspace networking system [56]. However, this is not fundamental, and the MTP stack could also be implemented as a library, in the kernel, or offloaded to the NIC.

Different from RDMA, the MTP stack prototype is a process, which, unlike the NIC, cannot access arbitrary memory in the application. Therefore, when creating the QP, the MTP stack also creates two shared memory regions, one for outgoing (TX) message bodies and one for incoming (RX), and the application maps the shared message body region to its address space. These memory regions are asynchronously accessed by both the application and MTP. To receive a message, the MTP stack must allocate space from the RX region to accommodate the message body; meanwhile, the occupied space of a message can only be freed by the application when it finishes using the received content, and vice versa for sending a message. The freeing of message space is not necessarily in the same order as allocation due to, for example, the reordering of messages. We implement a lock-free memory allocator to efficiently support this.

We encapsulate the MTP API into a shared library called `libmtp`, and implement a message generator that works with this library for evaluating MTP. The MTP stack, `libmtp`, and the message generator add up to ∼6K lines of C/C++ code.

Wrapper and Reference Offload: To allow commodity off-the-shelf L7 offloads to work with MTP, we implement our own MTP-compatible reference message processing wrappers, for complex L7 offloads or "middleboxes". The wrapper is implemented over DPDK with the ability to generate PTX and PRX ACKs, as well as providing early congestion feedback. We assume that routing packets from/to the middlebox is handled by separate switching logic in the INC hardware.

One key challenge is in efficiently utilizing the middlebox's buffer: With the assumption of full buffering, the middlebox cannot start processing a message until all segments have arrived. When one segment is dropped, the received segments occupy the buffer until retransmission comes. During this time other messages that would have been received in full might be dropped due to the lack of space. Our insight here is that a partially received message can be removed if the missing segments don't arrive within a short timeout. This is based on the protocol design that MTP messages are always sent in full and one can expect that the gap between packets of a message is predictable, only subject to serialization and fabric queuing delays. In our experiments, we conservatively set the receive timeout to 100μs.

To aid in our experiments, we embellish the middlebox wrapper with NetCache (§8.1) and a reference L7 offload. The reference offload emulates configurable processing time and buffer size. It reassembles message segments received from the wrapper, adds processing delay, segments the processed message and sends the segments out through the wrapper. This reference middlebox (wrapper and the reference offload) takes 600 lines of C code.

8 Evaluation

We first perform an end-to-end evaluation with NetCache [35] an existing INC application, showing that MTP improves its performance (§8.1). Then in §8.2, we carry out microbenchmarks on testbed to evaluate the two major design aspects of MTP that accommodate pathlet *delaying*: the pathlet ACK-based dual-RTO design for the reliability protocol, as well as MTP's congestion control that consists of pathlet feedback and proactive pathlet switching. We also study the overheads of pathlet ACKs. After that, we use large-scale simulation in

§8.3 to evaluate MTP's support for message reordering (at DCN scale), mutation, and intercept compared to alternatives.

8.1 Application Benchmarks

We adopt NetCache [35], a programmable switch-based key-value cache offload for this experiment. We run the NetCache client on one node in our testbed and run the NetCache backend server on another node. Each node has a Intel Xeon 1.8GHz 16-core CPU and 64GB DRAM. The two nodes are interconnected with 25Gbps Ethernet via a Tofino-based switch which runs the cache offload. We focus on read operations . As per our workflow, the vanilla client (which is also our baseline) encapsulates each read request RPC into a UDP packet toward the cache offload. For cache hits, the cache directly appends the value to the packet and sends it back to the client, and for cache misses, the offload forwards the requests to the backend server. The client recovers from losses of RPCs by retrying after a timeout, similar to key-value cache RPC in production [73]. The MTP-based client achieves the same workflow by establishing a connection to itself via the cache and backend, which are two *chained pathlets* in the path of this connection, and encapsulating each RPC into an MTP message. In this workflow, the messages are *mutated* with the appended values and can be *reordered* due to the cache's direct reply. Loss recovery is provided by the MTP stack.

The backend is allocated a single CPU core which runs at ∼1 Mops, and we use a query pattern of 50% hit rate, with which the system should be able to achieve a maximum throughput of ∼2 Mops. We add to both the cache and backend MTP header parsing logic, as well as to the backend pathlet feedback logic that reflects the RPC queue occupancy. We set both the UDP-based retry timeout and MTP RTO to 125μs, which is ∼25 times the RTT to the backend server. Note that this setup doesn't require dual RTOs for MTP because the delays at both pathlets are predictable.

We let the client generate read operations in an open-loop manner at variable percentages of the max throughput with Poisson inter-op arrival pattern, and observe the achieved throughput and tail op latency, as shown in Figure 4. The MTP-based client can sustain over 95% of system throughput whereas the UDP-based client starts falling below the offered load at 80%. Also, the MTP-based client sees no more than 4x p99 op latency relative to that at zero load, while the tail latency skyrockets with the UDP-based client starting from 75% load. This is because at high loads, the UDP-based client overflows the server and causes huge amounts of spurious retransmission (center figure) that wastes the server throughput, and this is not the case with MTP that provides pathlet CC, which limits the RPC buildup and results in negligible drops and retransmissions. At lower loads, the latency of the MTP-based client is slightly higher due to the extra processing of the MTP stack. Such a delay is necessary and inevitable in providing transport functions, including reliability and congestion control [33, 38, 58, 71].

Component	UDP-Based	MTP-Based
Cache (P4)	618	687
Client (C++)	825	423
Backend (C++)	323	323

Table 1: **Lines of code comparison between UDP-based and MTP-based NetCache components.**

Integration Cost. MTP is of low cost to integrate. Table 1 shows the lines of code (LoC) of UDP- and MTP-based NetCache components. MTP-based cache adds, in addition to the UDP-based, 69 lines of P4, which mainly includes the MTP header definition and parsing logic. The additional Tofino resource consumption is negligible with only 0.3% more crossbar and 2.3% more gateway usage. The MTP-based client is of much fewer LoC than UDP-based because the MTP-based client does not need the timeout-retry logic, which along with DPDK setup is handled by the MTP stack. Also, `libmtp` is light-weight to use. The MTP-based backend uses the same LoC as UDP-based: Coding packet IO with the MTP middlebox wrapper is of similar complexity to DPDK.

8.2 Microbenchmarks

For microbenchmarks we use CloudLab [21] as our testbed. We use nodes with AMD EPYC 3.0GHz 16-core CPUs and 128GB DRAM interconnected by 25 Gbps Ethernet. We place our reference middleboxes on the path between the network interface and the MTP stack emulating SmartNIC processing cores [52, 54]. We assign each middlebox on a dedicated CPU core. Packets that arrive at the host are first handled by the middlebox, which assembles the message and, after processing time, re-packetizes the processed message, and transmits the packets to the MTP stack via a ring buffer.

Though MTP makes no assumption on pathlets' service times, we evaluate MTP by making the middleboxes emulate two commonly-studied long-tail service time distributions: exponential and bimodal [19, 24, 36, 51, 54, 66].

8.2.1 Dual RTOs with PTX and PRX ACKs

MTP adopts PTX and PRX ACKs to determine whether the message is being buffered and processed by a pathlet, which enables using dual RTOs to detect fabric drops ("fabric RTO") and failures of message processing pathlets ("pathlet RTO"). We show the benefits of this with a testbed experiment.

We use our message generator (§7) to issue a flow of 128KB messages from one node to another in a closed loop. The receiver node enables our reference middlebox. The processing time is drawn from either an exponential distribution with a mean of 100 μs, or a bimodal distribution of a similar mean with 95%-ile of 50μs and 5%-ile of 1 ms. The receiver randomly drops 1% of the incoming packets from the interface, which is equivalent to fabric drops from the middlebox's and MTP stack's points of view.

Figure 5 (left) shows a comparison of goodput with a single per-message end-to-end RTO versus with dual RTOs based on

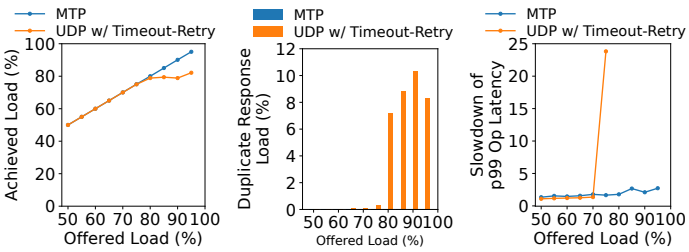


Figure 4: **Left, Center: achieved throughput and retransmission rate as percentage of system throughput, respectively; Right: p99 op latency slowdown relative to unloaded op latency.**

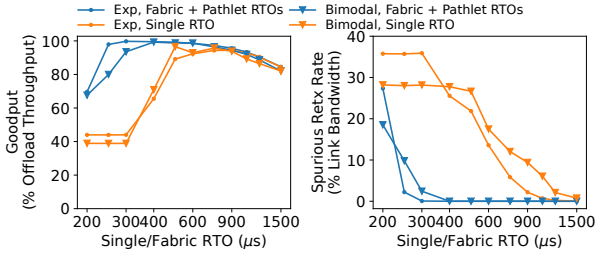


Figure 5: **Achieved throughput with respect to RTO (left) and spurious retransmission rate at the reference middlebox (right).**

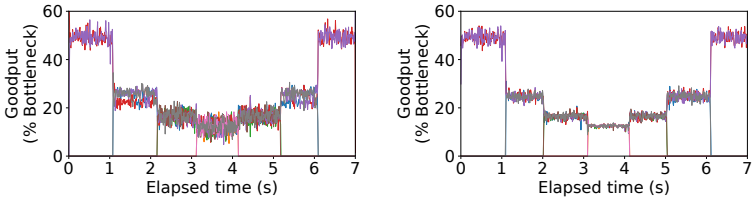


Figure 6: **Convergence under exponential pathlet processing time distriburion with legacy (left) and MTP (right) CC.**

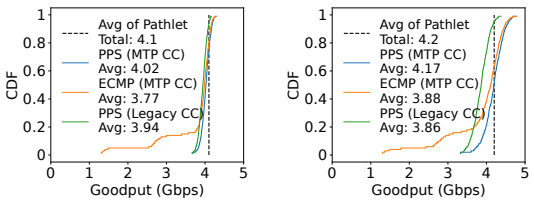


Figure 7: **Per-10ms total goodput with exponential (left) and bimodal (right) pathlet processing times.**

PTX and PRX ACKs. When dual RTOs are used, the X-axis is the fabric RTO because it is in effect as we set the pathlet RTO loosely to 4 ms.

Generally, both overly short and long RTOs cause goodput degradation. Long RTOs impair the sender's ability to quickly recover from losses. Short RTOs cause false positives, where the sender retransmits the messages when they are actually not lost. The right figure shows the rate of packets received at the offload that are spurious retransmissions (or "false positives") and hence dropped. The middlebox determines that a packet is a false positive if it has buffered the same packet of the same message. The message can be pending for processing (received in full), being processed, or waiting for other packets to arrive. We find that MTP's dual RTO design leads to a wider range (400-1050 μs) of fabric RTO choices where MTP achieves >90% goodput with zero false positive (hence no waste of link bandwidth) for both processing time distributions because the PRX and PTX ACKs allow the fabric RTO to not account for the middlebox's unpredictable processing time, whereas such range does not exist for single RTO.

8.2.2 Congestion Control

We compare MTP's CC solution with early multi-bit pathlet feedbacks, reacted by Swift-inspired CCA and PPS, to a legacy solution, where pathlets provide legacy ECN feedback, marking the CE bit in packets when they are congested (>50% buffer occupancy), and the receiver reflects the per-packet ECN back to the sender. We use DCTCP [8] to react on the ECN.

Convergence. We start 8 flows, 2 at each time, of 4KB messages from 4 sender nodes towards one receiver node that has a bottleneck middlebox. We configure the middlebox such that the processing time is drawn from either a exponential distribution averaging 12 μs or a bimodal distribution of 95% 6 μs and 5% 120 μs. The middlebox has a 32KB message buffer, which is roughly 4 times the bandwidth-delay product (BDP). Messages go through this middlebox before entering the receiver MTP stack.

Figures 6 compare how MTP flows converge based on the legacy and MTP's CC solution under exponential processing time distribution. The results for bimodal distribution can be found in Appendix D.1. Each line in the figures represents the throughput of a flow averaged every 10 ms. Most notably, as more flows share the middlebox, legacy CC fails to converge whereas MTP stably converges to a fair share. Early and multi-bit pathlet feedback allows MTP to perform AIMD based on the extent of congestion collected from each feedback than having to wait for at least a round trip to collect and smooth the congestion signals as with single-bit ECN [8].

Proactive Pathlet Switching (PPS). We show how PPS improves performance in addition to congestion control. We use a similar topology as above, with additionally a secondary middlebox of the same type on another node, and provide it to the sender-side MTP stack as an alternative pathlet. We configure the two middleboxes to be asymmetric such that the "secondary" incurs either exponential processing times of 24 μs or bimodal of 95%-ile 12 μs and 5%-ile 240 μs. We generate 4 closed-loop flows, all initially towards the "primary" pathlet, and expect PPS to distribute the flows across both pathlets. PPS randomly chooses a middlebox if the buffer occupancy is above 60% for 3 consecutive feedbacks (MTP CC) or round trips (legacy CC). We use ECMP as a baseline and rehash the flows every 100 ms.

Figure 7 shows the cumulative distributions of the total application-observed throughput every 10 ms. The achieved throughput can at times exceed the total of the pathlets' average throughput because of the long-tail processing time distributions. We find that for both distributions MTP with

PPS can achieve a total goodput that is ∼98% of the average total throughput of the two pathlets, whereas with ECMP can only achieve ∼90%. This is because PPS can quickly redistribute the flows based on congestion, despite random choice having a small chance of mapping all flows to the same pathlet, whereas ECMP cannot recover from such collision until next rehashing. The legacy CC combined with PPS achieves slightly lower average throughput than MTP due to the previously shown convergence problem.

This subsection shows that our CC solution based on MTP's primitives is effective in the face of typical processing time distributions, and we also evaluate MTP's CC with multiple pathlets in Appendix D.2. However, this is not necessarily the only possible way to use the primitives, and it is not our goal to innovate on the CCA. There are known limitations to DCTCP that have been addressed by other CCAs [11,15,25,27,50,60,61]. We leave a systematic study of CC for INC to future work, noting that the similar approaches to evolve from DCTCP may also be used to improve the CC performance in MTP.

8.2.3 ACK Overheads

MTP leverages PTX and PRX ACKs to communicate message states in the network and pathlet feedbacks, in addition to the E2E ACKs. These extra packets can impose overhead on both link bandwidth and the endpoints' processing. We quantify such overhead with an experiment, where one sender node sends messages to one receiver node in a closed-loop manner, with varying numbers of pathlets in between, as well as different message sizes and enabled ACK types (pathlet ACKs + feedbacks vs. feedbacks only). We use 2 CPU cores for the MTP stack in this experiment.

Figure 8 shows the percentage of link bandwidth consumed by ACK packets. The 0-pathlet bars reflect the overhead of E2E ACKs only. In general, the ACK bandwidth overhead is linear in the number of pathlets and the inverse of message size. This is expected because each pathlet ACKs on a per-message basis. Enabling both pathlet ACKs and feedbacks incurs around twice the bandwidth consumption of feedbacks only, because with the former, each pathlet ACKs twice (PRX, piggybacked by feedback, and PTX) per message, whereas with the latter the pathlet sends back only one feedback packet for each message. The worst case occurs with MTU-sized (4KB) messages, 2 pathlets and with both pathlet ACKs and feedbacks enabled, using 6% of link bandwidth, which is negligible. Note that most (if not all) existing L7 INC works place two or fewer pathlets in a path.

Figure 9 shows a breakdown of CPU cycles spent by the sender-side MTP stack with MTU-sized messages, as percentages of total CPU cycles. The total height of a bar represents the total cycles spent on the receive (RX) routine, which consists of four parts: 1) congestion control update (blue), which runs the CCA upon feedback; 2) message state update (orange), which resets the timer and switches between the fabric

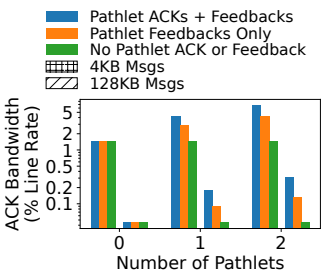


Figure 8: **Link bandwitdh overhead of MTP ACKs.**

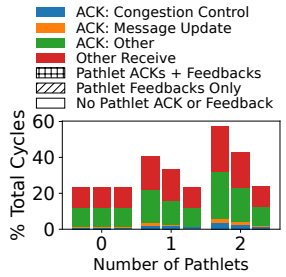


Figure 9: **Sender MTP stack CPU cycle breakdown.**

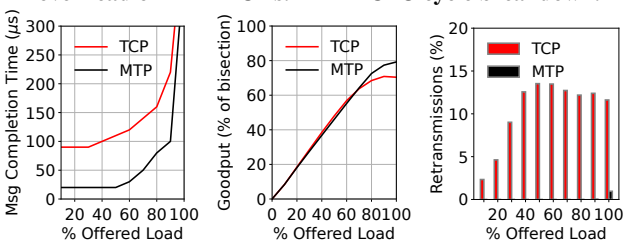


Figure 10: **Left: p99 message completion time; Center: average goodput achieved across the network; Right: retransmissions as percentage of the flows' original packets.**

and pathlet RTOs; 3) other ACK handling (green), which includes updating statistics and preparing CC input from statistics (pathlet feedback, ECN, etc. within a round trip); and 4) other RX logic (red), which involves packet header parsing and connection state lookup. Similar to link bandwidth usage, more CPU cycles are consumed with more pathlets and pathlet ACKs enabled, which incur higher ACK rates. Nonetheless, with 2 pathlets, the total CPU usage of RX is around 55%, still allowing the transmit logic to saturate the 25Gbps link. This is of higher throughput and message rate than some of the well-optimized user-space TCP [33] and software RDMA [58] stacks with the same number of cores.

In comparison, UDP does not have ACKs and hence does not incur similar bandwidth and CPU overheads. However, UDP also lacks essential transport functions such as reliability and congestion control. Building basic reliability upon UDP requires at least end-to-end ACKs which will incur overhead similar to MTP in the 0-pathlet case.

8.3 Large-Scale Simulation

To study MTP's support for reordering at scale, we use ns-3 simulations [7]. We simulate a fat-tree topology with 128 nodes with an over-subscription factor of two. We use 100 Gbps links with varying delays to model asymmetry—50% of aggregation-core links have a delay of 2 μs and the rest have a delay of 4 μs. The switches have a buffer size of 975 KB per port,which closely matches datacenter switches.

We generate *all-to-all* traffic with one-packet messages of 1 KB and a flow size of 1 MB. We use TCP as a baseline, and both TCP and MTP use the DCTCP [8] for congestion control (recall that MTP uses DCTCP for link congestion).

We leverage packet spraying to induce message reordering at scale (our messages are one-packet), instead of simulating L7 offloads. The reordering is exacerbated by the difference in path delays in our fat tree. Results are shown in Figure 10.

The left figure shows the 99%-ile message completion times with respect to load, and the center figure shows the goodput. MTP reduces tail message completion times by about 65% (speed up of 3.6x) across all loads. MTP is also able to operate at higher loads (i.e., beyond 80%) without saturating whereas TCP saturates saturates earlier (i.e., after 60% load).

To understand why MTP's performance is better, the right figure show the number of packet retransmissions as a percentage of the flows' original number of packets. We observe that TCP suffers from 10–15% packet retransmissions and MTP nearly eliminates such overhead as it is robust to reordering. Also, TCP saturates at around 60% load, so the percent of retransmitted packets does not increase after 60% as many flows do not finish and get backlogged indefinitely. We believe this is because MTP can precisely respond to congestion using per-message signals and not misinterpreting reordering as drops or congestion. Further, we show MTP's operation with message mutation and intercept in Appendix E.

9 Discussion

In this section, we discuss the problems that may arise when applying MTP to real-world production scenarios, and the potential approaches to tackle them.

Scalability. In §8.2.3, we have shown that MTP's ACK packets incur reasonable overhead, to both link bandwidth and CPU, with up to 2 pathlets. Though we have argued that this is the case with most existing L7 INC works, more pathlets may be required in a path with the emergence of more INC ideas, or with the mixing of offloads from different applications, which can result in higher ACK overheads. One potential approach to optimize the ACK overheads is reducing unnecessary ACKs. In our experiments, when the dual-RTO mechanism is enabled, all messages would trigger PTX and PRX ACKs. This is not always necessary if the pathlet's processing delay is short most of the time, in which case the pathlet can start sending pathlet ACKs only when long delay is imminent. Similarly, not all messages need to trigger congestion feedback, especially when the pathlet is not heavily loaded. Also, the CPU overhead of ACKs can be mitigated by offloading the MTP stack onto a dedicated ASIC, similar to RDMA.

Streaming. MTP does not natively support streaming, because doing so requires storing transport state in the pathlet for handling losses and out-of-order arrival of data segments, which would complicate MTP and prevent some offload hardware from supporting it. Nevertheless, pathlets that need streaming can first have the sender break down the stream into single-packet MTP messages. MTP can provide reliable per-message delivery and congestion control, on top of which reordering mechanisms can be implemented at the pathlets and endpoints.

Security. Though out of our scope, security remains an open problem for, not just MTP, but INC in general, because INC involves examining and tampering with packet payload by the network, which is exactly what today's transport-layer security (TLS) strives to prevent. In an effort to resolve this, mcTLS [62] extends data privacy and integrity to middleboxes and enables explicitly controlling each middlebox's read/write access to the data being transferred. It may be complementary to MTP and provide similar security features: MTP's handshake can be amended with a similar key exchange, and a TLS record naturally fits in an MTP message. We leave further investigation into security issues to future work.

10 Other Related Work

There is a significant amount of recent research on developing new CCAs, including pHost [25], ExpressPass [15], NDP [30], Homa [61], dcPIM [13], HPCC [50], PINT [11] and BFC [27]. MTP is intentionally designed to be orthogonal and complementary to these new algorithms as datacenter operators can switch between CCAs. For example, MTP's pathlet feedback may be leveraged to communicate the "clearance to send" in receiver-driven CC [13, 15, 25, 30, 61], or it can be seen as a form of in-band network telemetry which has enabled more precise CC [11, 50]. We leave the study on applying more CCAs to MTP to future work.

The idea of using pathlets for LB and CC in MTP is inspired by previous work that uses pathlets for inter-domain routing in the Internet [26]. MTP itself can be thought of as a practical realization of the classical application-level framing idea [16] applied to transport for INC-enabled DCNs.

11 Conclusions

This paper presents MTP, a new message transport protocol designed to work with INC. Through end-to-end testbed experiments, we found that MTP improves the performance of an existing INC application. We also showed that MTP provides effective reliable delivery and congestion control for INC through microbenchmarks and simulations. As INC adoption increases in datacenters, we believe that message-based transport protocols such as MTP that treat offloads as first-class citizens will become invaluable for scaling application performance.

Acknowledgements

We thank our shepherd, Kate Lin, and the anonymous NSDI reviewers for their feedback that significantly improved the paper. This research was supported by NSF grants CNS-2214015, CNS-2207317 and CNS-2008273. Akella was also supported by a gift from Cisco Research.

References

[1] CSP-7551: Hyper Network Appliance. https://www.accton.com/product-csp-7551/.

[2] Intel Extensible Switch Architecture. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-09/intel-tofino-expandable-architecture-paper.pdf.

[3] Intel Tofino Switches. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[4] Performance Best Practices — grpc.io. https://grpc.io/docs/guides/performance/. [Accessed 01-09-2023].

[5] Service discovery - Amazon Elastic Container Service — docs.aws.amazon.com. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-discovery.html. [Accessed 26-08-2023].

[6] Service discovery and DNS | Google Kubernetes Engine (GKE) | Google Cloud — cloud.google.com. https://cloud.google.com/kubernetes-engine/docs/concepts/service-discovery. [Accessed 26-08-2023].

[7] The ns-3 discrete-event network simulator. http://www.nsnam.org.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery.

[10] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020.

[11] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[13] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. DcPIM: Near-optimal proactive datacenter transport. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '22. Association for Computing Machinery, 2022.

[14] Prashant Chandra, Nandita Dukkipati, Yuliang Li, Naveen Kumar Sharma, Arjun Singhvi, and Hassan Wassel. Falcon Transport Protocol Specification. https://www.opencompute.org/documents/falcon-spec-v1-1-pdf-1, April 2024. [Accessed 03-10-2024].

[15] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17. Association for Computing Machinery, 2017.

[16] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In Phil Karn, editor, *Proceedings of the ACM Symposium on Communications Architectures & Protocols, SIGCOMM 1990, Philadelpia, PA, USA, September 24-27, 1990*, pages 200–208. ACM, 1990.

[17] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. Offloading load balancers onto SmartNICs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 56–62, New York, NY, USA, 2021. Association for Computing Machinery.

[18] William J Dally et al. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed systems*, 3(2):194–205, 1992.

[19] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Advait Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*, pages 2130–2138, 2013.

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

[22] Clarence Filsfils, Darren Dukes, Stefano Previdi, John Leddy, Satoru Matsushima, and Daniel Voyer. IPv6 Segment Routing Header (SRH). RFC 8754, March 2020.

[23] Linux Foundation. Data plane development kit (DPDK), 2015.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[25] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[26] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 111–122, New York, NY, USA, 2009. Association for Computing Machinery.

[27] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[28] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[29] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[30] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 422–434, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. P4DB - the case for in-network OLTP. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1375–1389. ACM, 2022.

[33] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[34] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. *Proc. VLDB Endow.*, 14(9):1626–1639, may 2021.

[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion

Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2017.

[36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[38] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Nikolay Kavaldjiev, Gerard JM Smit, and Pierre G Jansen. A virtual channel router for on-chip networks. In *IEEE International SOC Conference, 2004. Proceedings.*, pages 289–293. IEEE, 2004.

[40] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21. Association for Computing Machinery, 2021.

[41] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 863–879, USA, 2019. USENIX Association.

[43] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[45] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2021.

[46] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *SOSP*, 2017.

[47] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[48] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2020.

[49] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2016.

[50] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming

Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent Stephens, Hassan Wassel, and Aditya Akella. Ringleader: Efficiently offloading intra-server orchestration to NICs. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation*, NSDI '23. USENIX Association, 2023.

[52] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. Towards accelerating data intensive application's shuffle process using SmartNICs. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(2), may 2023.

[53] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, November 2020.

[54] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[55] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray C. C. Cheung, and Jianfei He. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 376–391, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[57] Mellanox Technologies. RDMA aware networks programming user manual, 2015.

[58] Mao Miao, Fengyuan Ren, Xiaohui Luo, Jing Xie, Qingkai Meng, and Wenxue Cheng. SoftRDMA: Rekindling high performance software RDMA over commodity Ethernet. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet '17, page 43–49, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM, 2021.

[60] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: RTT-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[62] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in tls. *SIGCOMM Comput. Commun. Rev.*, 45(4):199–212, aug 2015.

[63] John Ousterhout. A Linux kernel implementation of the Homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 99–115. USENIX Association, July 2021.

[64] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-accelerated network applications. In *OSDI*. USENIX Association, 2018.

[65] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19. Association for Computing Machinery, 2019.

[66] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale

networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[67] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. PLB: Congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 207–218, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

[69] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2021.

[70] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 740–755. ACM, 2021.

[71] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

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

[73] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20. Association for Computing Machinery, 2020.

[75] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. Skv: A smartnic-offloaded distributed key-value store. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2022.

[76] Web Team. Service Discovery in a Microservices Architecture - NGINX — nginx.com. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/. [Accessed 26-08-2023].

[77] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. *SIGCOMM Comput. Commun. Rev.*, 35(4):37–48, aug 2005.

[78] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using Trio: Juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[79] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[80] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps intrusion prevention on a single server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2020.

[81] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[82] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2015.

A MTP Packet Header

Listing 1 shows the MTP packet header formats.

```
struct mtp_hdr_t {      // MTP header
  uint16_t src_port;    // source port
  uint16_t dst_port;    // destination port
  uint16_t len;         // length of this header and payload
  uint16_t checksum;    // packet checksum
  uint32_t msg_no;      // message number
  uint16_t seg_no;      // segment number within the message
  struct {
    uint8_t msg_ack : 1;   // E2E ACK
    uint8_t prx_ack : 1;   // PRX ACK
    uint8_t ptx_ack : 1;   // PTX ACK
    uint8_t feedback : 1;  // pathlet feedback follows
    uint8_t reserved : 4;
  } flags;
  uint32_t msg_len;     // total byte length of message
  uint8_t vc;           // virtual channel
}; // packed struct

struct mtp_feedback_t {  // MTP pathlet feedback header
  uint32_t pathlet_id;   // pathlet instance identifier
  uint8_t feedback;      // feedback value
}; // packed struct
```

Listing 1: MTP header formats.

B MTP Connection State

Table 2 shows the state maintained per connection.

Transmission (TX) Connection State	
`tx_msgs[]`	Message descriptors for TX, each containing a message number, message length, pointer to payload, next-to-send segment number.
`tx_vcs[]`	TX virtual channels, each containing a last message number and pointer to the occupying message descriptor.
`cc_state[]`	CC state for link and each pathlet, each piece containing a cwnd, cwnd occupancy and CCA-specific statistics (§C)
`lb_state[]`	LB state for each type of multi-instance pathlets, containing statistics for PPS (§6.2)
Receive (RX) Connection State	
`rx_msgs[]`	Message descriptors for RX, each containing a message number, message length, pointer to payload, and bitmap that marks received segments.
`rx_vcs[]`	RX virtual channels, each containing a last message number and pointer to the occupying message descriptor.

Table 2: **MTP Connection State.**

C MTP Congestion Control Algorithm

Algorithm 1 shows the implementation details of our Swift-inspired CCA.

D Additional Congestion Control Evaluation

D.1 Convergence

Figure 11 shows the convergence of per-flow goodput with legacy and MTP CC under bimodal pathlet processing time distribution. This time distribution does not cause severe fluctuation in goodput with many flows, even with legacy CC. Despite this, MTP CC still mitigates the divergence of legacy CC with 4 flows (1-2 and 4-5 seconds).

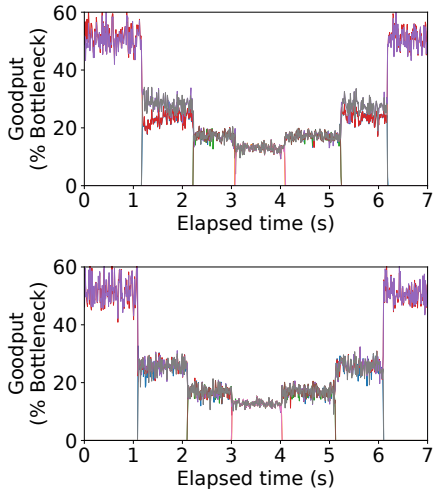


Figure 11: **Convergence under bimodal pathlet processing time distriburion with legacy (upper) and MTP (lower) CC.**

D.2 Multiple Pathlets

There can be more than one pathlet on a path, and it is important that MTP handles this. We evaluate MTP's CC with pathlet chains with different bottleneck locations. We use the topology in Figure 12, which represents a scenario where multiple paths share a bottleneck pathlet after some

Algorithm 1: MTP CC and PPS Algorithms

Data: $active_pathlet_{type} \forall type \in path$

Procedure on_rx_pathlet_feedback(*pathlet, feedback, msg_seg_num*)

/* Update cwnd with Swift */

if $feedback > 127$ **then**

  $cwnd_{pathlet} \leftarrow (0.4 + 0.6 \frac{128}{feedback}) \cdot cwnd_{pathlet}$

else

  $cwnd_{pathlet} \leftarrow (1 + \frac{msg_seg_num}{2 \cdot cwnd_{pathlet}}) \cdot cwnd_{pathlet}$

end

/* Decide whether to switch pathlet */

$type \leftarrow typeof(pathlet)$

if $pathlet = active_pathlet_{type} \wedge feedback > 160$ **then**

  $high_congestion_count \leftarrow high_congestion_count + 1$

else

  $high_congestion_count \leftarrow 0$

end

if $high_congestion_count = 3$ **then**

  $active_pathlet_{type} \leftarrow random(type)$

  $high_congestion_count \leftarrow 0$

end

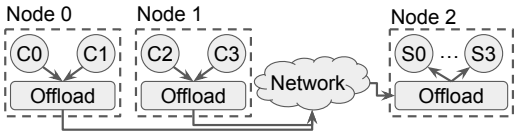


Figure 12: **Multi-pathlet topology. Two client processes on each of Node 0 and 1 send messages to a server process running on Node 2. Messages go through both client- and server-side offloads (pathlets). The bottleneck is the pathlet on Node 2.**

	Exponential (12 μs)		Bimodal (95% 6 μs, 5% 120 μs)	
CC	Avg	Stdev	Avg	Stdev
MTP	5.24	0.179	5.18	0.363
Legacy	5.30	0.360	5.04	0.544

Table 3: **Average and standard deviation of per-10 ms aggregate goodput in Gbps in chained pathlet topology.**

other pathlets. The offloads' processing times are drawn from distributions described in Table 3. We use the message generator as the client to generate 8KB messages (2 segments) in closed loops, set the offload buffer to 96KB, and configure the MTP stack to use MTP or legacy CC.

Table 3 shows the statistics on the total goodput across all flows for every 10 ms. We find that MTP and legacy CC achieve similar average goodput but the legacy CC yields 100% more standard deviation for exponential distribution and 50% more for bimodal distribution. This is due to the drawbacks of legacy end-to-end single-bit feedbacks, similar to the convergence problem discussed in Section 8.2.2.

E Simulation Results

We use a dumbbell topology to study MTP's operation with message mutation and intercept. The topology consists of 8 nodes on each of two sides of the bottleneck. All links are 10 Gbps and 10 μs.

Operation with Message Mutation. We make eight clients from one side send 4KB (4-segment) messages in a closed-loop manner to eight servers on the other side. We simulate

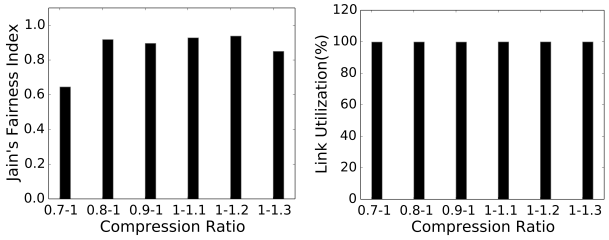


Figure 13: **MTP operation with mutation.**

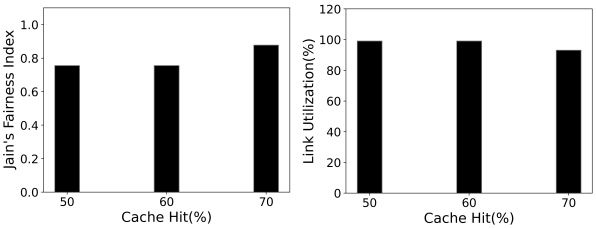


Figure 14: **MTP operation with intercept: fairness (left) & link utilization (right)**

an offload on the switch that (de)compresses data before the bottleneck link with an average processing delay of 2 μs. This represents not only compression engines in the real world but also other offloads such as HTTP load balancer that can change the message size [17]. The offload randomly chooses a compression ratio within a given range. A ratio > 1 means a decrease in message size.

Figure 13 shows the Jain's fairness index across all clients (left) and the utilization of the bottleneck link (right). We observe that MTP achieves good fairness as well as 100% utilization even as the offload decreases or increases the payload size by as much as 30%. However, fairness begins to deteriorate somewhat as the range expands beyond 20% (increase or decrease). We believe that this is due to DCTCP, which is not optimal for handling sudden data size changes in the network, and an optimized CCA for mutation is warranted, which we leave for future work. Note that TCP would fail to operate correctly in such cases.

Operation with Message Intercept. Using the above topology and workload we simulate an offload, which opportunistically intercepts a message. We vary the intercept rate from 50% to 70% (i.e., with 70% intercept rate, only 30% of messages will reach the dumbbell bottleneck). MTP remains unaffected by message intercept. We observe in Figure 14 that MTP achieves good fairness and full utilization of the bottleneck link even as the offload intercepts 50%–70% of messages. Both TCP and RDMA would fail to operate correctly in such cases.

ONCache: A Cache-Based Low-Overhead Container Overlay Network

Shengkai Lin[1], Shizhen Zhao[1], Peirui Cao[1], Xinchi Han[1],
Quan Tian[2], Wenfeng Liu[2], Qi Wu[2], Donghai Han[2], and Xinbing Wang[1]
[1]*Shanghai Jiao Tong University,* [2]*Broadcom*

Abstract

Recent years have witnessed a widespread adoption of containers. While containers simplify and accelerate application development, existing container network technologies either incur significant overhead, which hurts performance for distributed applications, or lose flexibility or compatibility, which hinders the widespread deployment in production.

We carefully analyze the kernel data path of an overlay network, quantifying the time consumed by each segment of the data path and identifying the *extra overhead* in an overlay network compared to bare metal. We observe that this extra overhead generates repetitive results among packets, which inspires us to introduce caches within an overlay network.

We design and implement ONCache (**O**verlay **N**etwork **Cache**), a cache-based container overlay network, to eliminate the extra overhead while maintaining flexibility and compatibility. We implement ONCache using the extended Berkeley Packet Filter (eBPF) with only 524 lines of code, and integrate it as a plugin of Antrea. With ONCache, containers attain networking performance akin to that of bare metal. Compared to the standard overlay networks, ONCache improves throughput and request-response transaction rate by 12% and 36% for TCP (20% and 34% for UDP), respectively, while significantly reducing per-packet CPU overhead. Popular distributed applications also benefit from ONCache.

1 Introduction

Containers [9, 21] are becoming increasingly popular for distributed application deployment due to their flexibility and lightweight nature. By bundling an application along with all its dependencies and configuration files into one container image, containers bring forth the ability to *build once and run anywhere*. Furthermore, containers leverage the host OS kernel instead of emulating an entire OS, and thus are more lightweight than virtual machines. Container orchestration tools such as Kubernetes [35], Docker Swarm [22], etc., further reduce container management complexity by facilitating features like auto-deployment, auto-scaling, and auto-high-availability [8].

Technology	Performance	Flexibility	Compatibility
Host	✓	✗	✓
Bridge	✓	✗	✓
Macvlan	✓	✗	✓
IPvlan	✓	✗	✓
SR-IOV [64]	✓	✗	✓
Overlay	✗	✓	✓
Falcon [37]	✗	✓	✓
Slim [74]	✓	✓	✗
ONCache	✓	✓	✓

Table 1: *Compare container networking technologies.*

Container networks play a key role in enabling communication between containers and supporting distributed applications. Ideally, a container network should offer low-overhead, high-performance networking while preserving flexibility and compatibility. However, none of the existing container networks fully satisfy these expectations, as outlined in Table 1.

Hurt flexibility. Containers in the host network share the host IP address and must coordinate ports on the host, constraining deployment flexibility. Bridge networks or device virtualization techniques (such as Macvlan/IPvlan/SR-IOV [64]) allow containers to have their own IP addresses and directly forward container packets to the underlay network. However, these networks require either all hosts to be in the same L2 network or the underlay network to be capable of routing container packets, imposing restrictions on container placement, migration, or network configuration.

Poor performance and high overhead. Container overlay networks utilizing tunneling techniques (e.g., VXLAN [41]) completely decouple containers and the underlay network, facilitating the configuration of container IP addresses regardless of the underlay network. Nonetheless, it incurs significant overhead [37, 38, 65, 74]. Our experiments show that compared to bare metal, the throughput of a single TCP flow in a tunnel-based overlay network is about 11% lower, with

the TCP request-response transaction rate being around 29% lower. Moreover, CPU utilization is much higher in both experiments. While certain techniques [36, 37, 59] endeavor to enhance overlay networks by distributing ingress packet processing across multiple CPU cores, the processing overhead of overlay networks remains significant.

Hurt compatibility. Slim [74] and its follow-up works [12, 39, 40] implement overlay networks via socket replacement, enabling containerized applications to utilize the host's sockets. While they maintain low overhead and some degree of flexibility in overlay networks, they do not inherently support non-connection-based protocols (e.g., UDP, ICMP) or container live migration. Additionally, they may conflict with tunneling-headers-based policies in the underlay network.

To attain high performance, flexibility, and compatibility at the same time, we design and implement ONCache (**O**verlay **N**etwork **Cache**), a container overlay network featuring a cache-based and low-overhead fast path. ONCache can be seamlessly integrated into existing standard overlay networks and has been tested with Antrea [1] and Flannel [28].

Our detailed contributions are summarized as follows:

1. We carefully analyze the kernel data path of popular container overlay networks (Antrea and Cilium) alongside bare metal, quantifying the time consumption of each segment of the overhead. Our analysis reveals the *extra overhead* incurred by container overlay networks. However, the overhead disperses across the kernel data path, presenting a significant challenge in mitigating it while preserving essential functionalities. (§2.2)
2. We identify the *invariance property* within the extra overhead, indicating that the processing results of the extra overhead are repetitive among packets. This property inspires us to use caches in overlay networks. (§2.4)
3. We propose a cache-based fast path for container overlay networks. The fast path bypasses the extra overhead without losing flexibility and compatibility. (§3)
4. We evaluate ONCache with microbenchmarks and popular distributed applications. Results demonstrate significant improvement of ONCache in throughput (12% higher with iperf3), latency (36% higher transaction rate), and CPU utilization (26% lower per-transaction) compared to the standard overlay networks, approaching performance akin to that of bare metal. Applications such as Memcached, PostgreSQL, and Nginx all derive benefits from ONCache. (§4)

ONCache's source code is publicly available at `https://github.com/shengkai16/ONCache`.

This work does not raise any ethical issues.

2 Background and Motivation

We analyze inter-host container networks, where overlay networks stand out for the superior flexibility and compatibility. After quantifying the detailed overhead of overlay networks and exploring recent efforts aimed at mitigating this overhead, we provide the motivation of ONCache.

2.1 Inter-host Container Networks

Container networks play a key role in facilitating communication among containers and with the external world. Below, we focus on inter-host container networks, which are classified based on the IP addresses utilized by *applications* and the *physical network*.

Both use host IP addresses. Host networks fall into this category. Containers within host networks share the host's network namespace and utilize the host's IP address. Host networks offer performance comparable to that of bare metal (where applications directly execute on the host) [65]. Nonetheless, host networks demand coordination of ports among containers residing on the same host, thereby substantially limiting application flexibility. This limitation renders host networks seldom used in production [66].

Both use container IP addresses. Bridge networks and device virtualization techniques, such as Macvlan, IPvlan, SR-IOV, fall into this category. In these networks, each container is assigned a unique IP address and is isolated from the host namespace. In bridge networks, all the containers are connected to Open vSwitch (OVS) [1], bridge [28], or similar entity, relying on them to forward packets to the physical network. Networks employing device virtualization techniques emulate multiple virtual interfaces with different IP addresses on each physical interface, with each virtual interface being attached to a container. These networks deliver performance akin to that of bare metal due to their simple data paths.

However, owing to their utilization of container IP addresses in the physical network, the above approaches exhibit poor flexibility. To route container IP addresses, one must either (1) place all hosts into one L2 network or (2) install and manage routing rules for container IP addresses within the physical network. These requirements impose constraints on container placement, migration or network configuration.

App uses container IP addresses and the physical network uses host IP addresses. Overlay networks fall within this category. In overlay networks, each container has its own IP address, and container packets are encapsulated using a tunneling protocol (e.g., VXLAN [41] or GENEVE [29]) before being transmitted to the physical network. Overlay networks completely decouple containers from the physical network, thereby ensuring flexibility for both. However, tunneling incurs extra overhead and hurts performance (see §2.2).

***Takeaway* #1** Overlay networks decouple applications from the physical network, allowing for application deployment without concern for physical network configuration, and vice versa. Given the superior flexibility of overlay networks, we focus on analyzing and improving overlay networks.

	Egress					Ingress				
Data path	Overhead type	Antrea	Cilium	BM	**Ours**	Overhead type	Antrea	Cilium	BM	**Ours**
Application network stack	skb allocation	1505	1566	1461	**1509**	skb releasing	715	818	780	**714**
	Conntrack	778	0	788	**763**	Conntrack	616	0	600	**592**
	Netfilter	0	0	305	**0**	Netfilter	0	0	173	**0**
	Others	423	560	547	**519**	Others	838	1016	979	**982**
Veth pair*	NS traversing	562	594		**489**	NS traversing	400			
eBPF*	eBPF		1513		**511**	eBPF		1429		**289**
Open vSwitch*	Conntrack	872				Conntrack	758			
	Flow matching	354				Flow matching	308			
	Action execution	92				Action execution	66			
VXLAN network stack*	Conntrack	0	471			Conntrack	0	271		
	Netfilter	667	421			Netfilter	466	303		
	Routing	50	468			Routing	294	554		
	Others	319	127			Others	619	444		
Link layer	Link layer	1858	1763	1799	**1700**	Link layer	2790	2848	2800	**2737**
Sum		7479	7483	4900	**5491**		7869	7683	5332	**5315**
Latency (μs)		22.97	23.15	16.57	**17.49**		22.97	23.15	16.57	**17.49**

Table 2: *Overhead breakdown of different networks. All values, except for the end-to-end latency in the last row, are in nanoseconds. BM stands for bare metal. Ours, i.e., ONCache, is proposed in §3 and evaluated in §4. "*" denotes the extra overhead compared to bare metal. Due to the limitations of the measurement tool, there is an error of about 200 ns.*

2.2 Analysis of Overlay Network Overhead

Prior studies [37,38,65,74] have shown the poor performance of overlay networks. We further quantify the overhead of two common overlay networks: Antrea [1] (v1.9.0) and Cilium [13] (v1.12.4), in comparison with bare metal.

Our analysis of the networks follows these steps. First, we qualitatively deconstruct the egress and ingress data paths using flame graphs [27], which provide comprehensive function call stacks on the data path. Next, we identify the exact functions (alongside their call stacks) that represent each segment of the overhead from the flame graphs. Finally, we measure the execution time for each function using the extended Berkeley Packet Filter (eBPF) and subsequently calculate the execution time for each segment of the overhead. The detailed analysis method is provided in Appendix A.

Table 2 presents the deconstructed overhead and the execution time for each segment of the overhead. Each listed execution time is the average of all the timing samples within a 1-second 1-byte TCP request-response (RR) test. We primarily analyze the overhead within bare metal and Antrea ("*" denotes the extra overhead compared to bare metal), while the distinct overhead within Cilium is also discussed. ONCache (ours) is analyzed in §4.

Application network stack. The application network stack directly interacts with applications. On the egress path, the application writes data to the socket. The application network stack then allocates a socket buffer (i.e., skb) for the data and encapsulates the packet layer by layer. Conversely, on the ingress path, the network stack decapsulates the headers and releases the socket buffer. Additionally, netfilter [47] filters packets on both the ingress and egress paths. Conntrack [18], as a component of netfilter, tracks connection (not TCP connections, defined by, for example, 5-tuple: source IP address, source port, destination IP address, destination port, and transport protocol) states and facilitates stateful filters to address probes and denial-of-service attacks. The execution of netfilter and conntrack depends on the system configuration.

Veth pair*. Containers run applications within individual network namespaces and employ veth pairs [69] to connect with the host namespace. Traversing veth pairs incurs overhead, which includes transmit queuing on the sender veth and software interrupt scheduling on the receiver veth. In Cilium, although the overhead of traversing veth pairs can be mitigated on the ingress path through bpf (Berkeley Packet Filter) redirect [71], it still exists on the egress path [17].

OVS*. OVS filters packets and intra-host routes packets between containers and the host interface. These filtering and routing actions are described by flows. The overhead in OVS can be categorized into connection tracking, flow matching, and action execution. Despite OVS employing a cache to expedite flow matching [53], connection tracking still consumes a substantial amount of CPU time. Cilium implements similar functionalities as OVS in eBPF [16]. However, the overall eBPF execution time is similar to that of OVS.

VXLAN[1] network stack*. The VXLAN network stack performs egress and ingress routing for VXLAN packets and encapsulates or decapsulates the outer headers (including VXLAN header, UDP header, IP header, and MAC header) to or from the packets. In Antrea, VXLAN routing is accelerated by OVS, resulting in notably low routing overhead. Additionally, conntrack tracks connections, and netfilter filters packets at this stage.

Link layer. On the egress path, the link layer queues and transmits packets to the network; on the ingress path, it allocates socket buffers and receives packets from the network.

***Takeaway* #2** The *extra overhead* in overlay networks, in-

[1]We take VXLAN as an example in following discussion. The analysis is similar for other tunneling protocols.

cluding veth pair, OVS (or similar entity), and the VXLAN network stack, incurs significant overhead and degrades network performance. Given its dispersion across various parts of the data path, mitigating the overhead while preserving essential functionalities presents a significant challenge.

2.3 Related Works on Overlay Networks

There are some recent efforts that aim at improving the performance of overlay networks.

Socket replacement. Slim [74] introduces a socket replacement mechanism that improves container overlay network performance. It establishes TCP connections in the host namespace and replaces TCP sockets in containers by those in the host namespace. While improving TCP performance, Slim exhibits notable limitations. Firstly, it does not support non-connection-based protocols such as UDP and ICMP, or container live migration, thereby constraining its compatibility. Secondly, Slim packets are not tunneling packets, which impacts underlay network policies that match on tunneling headers. Furthermore, Slim must establish an overlay connection for service discovery during connection setup, resulting in prolonged setup time. Some subsequent works [12, 39, 40] follow Slim's idea and improve connection setup time. However, they still suffer from the former two shortcomings.

CPU load balancing. Linux scaling techniques, such as Receive Packet Steering (RPS) and Receive Flow Steering (RFS) (and their hardware counterparts Receive Side Scaling (RSS) and Accelerated RFS (aRFS)) [59], improve performance by distributing ingress packet processing across multiple CPU cores and trying to reduce cache miss rate. Falcon [37] and mFlow [36] explore different strategies for distributing packet processing across multiple cores. However, they improve network performance at the cost of higher CPU utilization. Additionally, they only take effects on the ingress path and cannot further improve performance if the bottleneck lies in the egress path.

***Takeaway* #3** Despite many efforts to improve overlay networks, none of them simultaneously maintains low overhead, high performance, high flexibility and high compatibility.

2.4 Motivation: the Invariance Property

Based on the analysis in §2.2, we further classify the processing of the extra overhead into following types: connection tracking, packet filtering, intra-host routing, namespaces traversing and outer-header processing [2]. Among these overhead types, namespaces traversing does not yield any processing result, while the other four exhibit an *invariance property*.

Invariance in connection tracking. To the best of our knowledge, all common connection trackers (e.g., those in netfilter [2], OVS [51], Cilium [15]) have an established state, which signifies that the connection tracker has observed two-way communication of a connection. Once in the established state, the connection does not switch to another state until its completion.

Invariance in packet filtering. Packet filters can be classified into two types: stateless and stateful. Stateless filters only match on packet fields, typically the 5-tuple. Stateful filters also take into account the connection state, which is provided by the connection tracker. While OVS and netfilter contain numerous filters, the final decision to allow or deny packets with the same 5-tuple remains unchanged once the flow enters the established state.

Invariance in intra-host routing. If the container IP addresses are within the same subnet, layer-2 routing is enough, and the packet can be directly forwarded to the target interface. Otherwise, layer-3 routing is necessary and the MAC addresses should be modified before forwarding the packet. In both scenarios, the intra-host routing decisions are invariant for packets destined to the same container IP address.

Invariance in outer-header processing. On the ingress path, the outer headers are simply removed after checking fields such as destination and Time-to-Live (TTL). The correctness of the payload is ensured by the checksum of the inner headers. The egress path encapsulates the outer headers, whose most fields are related to routing and network configuration. On the same host, these fields are invariant for packets destined to the same container IP address: (1) The outer MAC header stores the MAC addresses, VLAN, and Ethernet type, which are invariant. (2) The IP addresses, protocol, TTL, differentiated services code point (DSCP), etc. in the outer IP header are invariant. Length, identification (ID), checksum fields vary among packets, but they can be easily updated with little computation. (3) In the outer UDP header, the destination port (VXLAN set to 4789), and the checksum (VXLAN set to 0)[3] are invariant [41]. The source port is calculated by simply hashing the 5-tuple of the inner headers. The length field can also be easily updated. (4) The VXLAN header stores the VXLAN network identifier (VNI), which does not change in an overlay network.

***Takeaway* #4** The extra overhead of overlay networks exhibits an invariance property. This observation motivates us to introduce *caches* to eliminate the extra overhead.

3 ONCache

ONCache is a container overlay network that offers a cache-based fast path with low overhead, high performance, flexibility, and compatibility. It is essentially a plug-in of standard overlay networks such as Antrea, Flannel. ONCache adopts a fail-safe design: when the fast path is unavailable, the traffic seamlessly falls back to the standard overlay network.

[2] If Kubernetes ClusterIP Service [62] is enabled, inter-host container traffic also needs Network Address Translation (NAT), as discussed in §3.5.

[3] While UDP checksum is necessary for certain tunneling protocols such as GENEVE, the computational overhead is low.

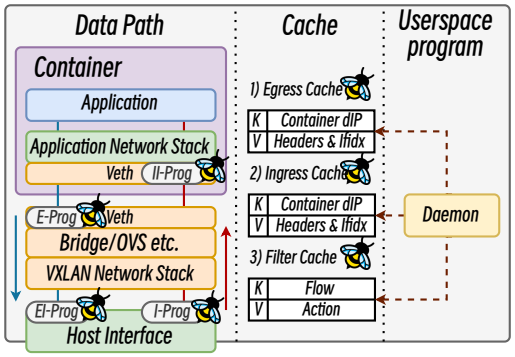


Figure 1: *Architecture of ONCache. ONCache consists of 4 eBPF programs in the data path, 3 eBPF maps, and 1 user space program. The components with the eBPF logo ("bee") represent eBPF programs or eBPF maps. dIP is short for destination IP address.*

eBPF Program	Abbr.	Hook Point
Egress-Prog	E-Prog	TC ingress of the veth (host-side)
Ingress-Prog	I-Prog	TC ingress of the host interface
Egress-Init-Prog	EI-Prog	TC egress of the host interface
Ingress-Init-Prog	II-Prog	TC ingress of the veth (container-side)

Table 3: *Hook points of the four eBPF programs.*

Define cache: Inspired by the invariance property discussed in §2.4, ONCache defines three caches—the egress cache, the ingress cache, and the filter cache—on each host to store the invariant results of the extra overhead. The detailed design of these caches is introduced in §3.1.

Initialize cache: ONCache relies on a standard overlay network to initialize the cache. In the event of a cache miss, ONCache passes the packet to the standard overlay network for processing and stores the processing results in caches if the packet's flow reaches the established state. See §3.2 for the detailed design.

Utilize cache: Upon a cache hit, ONCache filters the packet using the cached filtering results, constructs a tunneling packet using the cached outer headers, and then redirects the packet to the target interface by the cached routing decision. The cache-based fast path eliminates the *expensive* repetitive processing of *multiple layers*, as analyzed in §2.4, by several lightweight cache lookups. The detailed design is described in §3.3.

Maintain cache: ONCache ensures that the outdated cache entries are appropriately evicted when network-changing events (including container deletion, migration, and filter modification) happen. This is discussed in §3.4.

Cache compatibility: ONCache focuses on inter-host container networking and is also compatible with various traffic and features in the network. This is discussed in §3.5.

Optional improvements: We provide two optional improvements to further enhance performance. As they require kernel or tunneling protocol modifications, they are not applied as default designs. This is described in §3.6.

For ease of implementation and deployment, we implement ONCache with eBPF [24, 70]. eBPF is a technology that runs sandboxed programs within a privileged context like the Linux kernel, thereby safely and efficiently extending the capabilities of the kernel. eBPF programs are attached to kernel hook points (e.g., TC (traffic control) or XDP (eXpress Data Path) [31]) and are triggered each time the kernel executes through the hook points. eBPF stores data in a map data structure residing in kernel space, called eBPF map. Numerous prior works leverage eBPF for networking [4, 44, 56, 67, 72].

ONCache includes the following components, as shown in Figure 1: (1) four eBPF programs for packet processing and forwarding, with their hook points listed in Table 3; (2) three eBPF maps per host serving as the egress cache, the ingress cache, and the filter cache; (3) one user space daemon. The implementation of ONCache are much simpler compared to Slim and Falcon. The core functionalities can be realized with only 524 lines of eBPF C code, whereas Slim requires 2380 lines of code. Falcon, implemented with 606 lines of kernel code, is harder to implement and debug than eBPF. The core source code is shown in Appendix B.

Our idea diverges fundamentally from other cache-based network techniques. Prior works, such as OVS [53, 58] and Andromeda [20], leverage caches to accelerate flow/rule matching on their data paths. However, as analyzed in §2.2, the container overlay network remains suboptimal even with the utilization of the OVS cache, revealing the insufficiency of caching results of a individual layer. In contrast, ONCache is the first *cross-layer* cache within the container overlay network, which considers the extra overhead as a whole and caches the results once. Consequently, ONCache effectively eliminates the extra overhead compared to prior works.

3.1 The Cache Definition

ONCache contains three caches on each host, as shown in Figure 1: the egress cache, the ingress cache, and the filter cache. These caches are implemented using eBPF Least Recently Used (LRU) hash map, which evicts the least recently used elements to store new elements when the map is full.

The egress cache stores the outer headers and egress intra-host routing results, which are invariant for packets destined to the same container IP address. The routing results include the new inner MAC header (for L3 routing) and the egress host interface. To reduce the memory usage of the egress cache, we divide it into two levels: <*container destination IP address (dIP)* → *host dIP*> and <*host dIP* → *outer headers, inner MAC header, host interface index*>.

The ingress cache stores the ingress intra-host routing results, which are invariant for packets destined to the same container IP address. The structure of the ingress cache is <*container dIP* → *inner MAC header, veth (host-side) index*>.

The filter cache stores the filtering decision of each flow in the established state. Essentially, it functions as a flow whitelist, recording the allowed flows. By default, a flow is

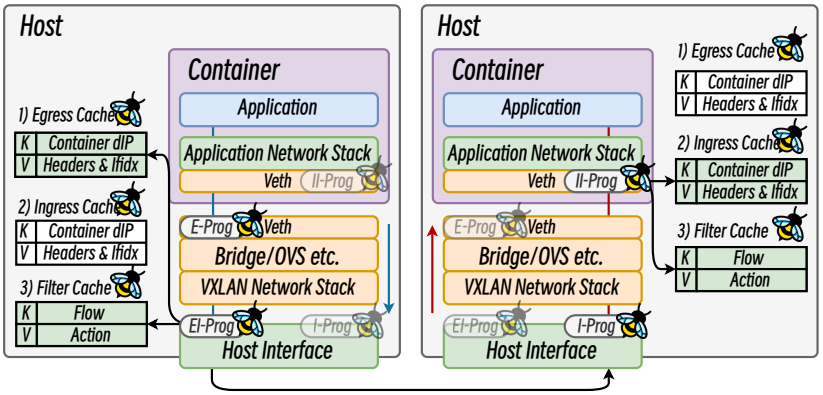


Figure 2: *The cache initialization process of ONCache. EI-Prog/II-Prog update the entries (green shaded) when the initialization requirements are met.*

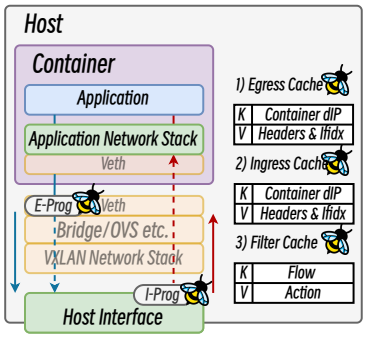


Figure 3: *Journey of an overlay packet in ONCache. EI-Prog/II-Prog are skiped over. The dash lines denotes the redirect path.*

defined by the 5-tuple (source IP address, source port, destination IP address, destination port, and transport protocol). One may also adjust the flow definition as required, e.g., adding a DSCP field to support DSCP filters.

In order to minimize cache misses, the capacity of the caches should be adjusted according to the cluster scale and concurrent flow number. Consider the largest cluster in Kubernetes [60], which has a maximum of 110 containers per host, 5k hosts, and 150k total containers. With up to 1M concurrent flows per host, to minimize cache misses, the egress/ingress/-filter cache will take up at most 1.56 MB/2.2 KB/20 MB of memory space per host, respectively (See Appendix C for details). This memory usage is negligible in modern servers.

3.2 Cache Initialization

If ONCache encounters a cache miss while trying to forward a packet, it will pass the packet to the fallback overlay network and try to initialize the cache meanwhile. The cache initialization process is depicted in Figure 2.

Initialize the Egress Path. When an egress container packet reaches the veth (host-side), Egress-Prog, which is hooked here, tries to forward the packet. If either the egress cache or the filter cache misses for the packet, Egress-Prog adds a *miss* mark to the packet. We reserve one bit within DSCP field of the inner IP header as the *miss* mark.[4] Egress-Prog then passes the packet to the fallback overlay network.

The fallback overlay network proceeds to forward, filter, and encapsulate the packet. Besides, we configure the overlay network to add an *est* mark to the packet if its flow reaches the established state. We use another bit in the DSCP field as the *est* mark. The aforementioned action only requires either minor adjustments to two OVS flow rules or the addition of one rule in netfilter, as detailed in Appendix B.2.

After being processed by the fallback overlay network, the packet reaches the host interface, where Egress-Init-Prog checks if the following requirements are met to initialize the caches: (1) the packet is a tunneling packet (e.g., a VXLAN packet); (2) the packet has the *miss* mark and the *est* mark.

If all requirements are met, Egress-Init-Prog initializes the egress cache by storing <*container dIP* → *outer headers, inner MAC header, host interface index*> into the cache. ONCache derives the host interface index from `sk_buff` struct, and the other fields from the tunneling packet. Additionally, the flow is whitelisted in the filter cache. Finally, the packet is sent out from the host interface.

Initialize the Ingress Path. Upon receiving a tunneling packet at the host interface, Ingress-Prog, which is hooked here, queries the cache and tries to forward the packet. If either the ingress cache or the filter cache misses for the packet, Ingress-Prog adds the *miss* mark to the packet and passes it to the fallback overlay network.

The fallback overlay network proceeds to process the packet, and add the *est* mark to the packet if its flow reaches the established state.

Then the packet reaches the veth (container-side), where Ingress-Init-Prog checks if the packet has both the *miss* and the *est* mark. If so, the program initializes the cache by storing <*container dIP* → *inner MAC header*> to the ingress cache and whitelisting the flow in the filter cache. <*container dIP* → *veth (host-side) index*> in the ingress cache is maintained by ONCache daemon upon container provisioning.

3.3 Cache-based Fast Path

We present the cache-based fast path of ONCache (Figure 3). The fast path is designed to be fail-safe and transparent to both applications and the underlay network. For packets that cannot be forwarded by the fast path, ONCache passes them to the fallback overlay network instead of dropping them.

3.3.1 The Egress Fast Path

The egress fast path is implemented by Egress-Prog, which handles an egress container packet in the following steps:

[4] We reserve 2 bits in total within the inner IP headers' DSCP field and require the network not to use these 2 bits for differentiated services.

Step #1: Cache retrieving. Egress-Prog first checks if the packet's flow is whitelisted in the filter cache and retrieves <*container dIP* → *outer headers, inner MAC header, host interface index*> from the egress cache. Any cache miss triggers a cache initialization process, as described in §3.2.

Additionally, Egress-Prog conducts a *reverse check* by verifying whether the container source IP address exists in the ingress cache and if the reverse flow is also whitelisted in the filter cache. If the reverse check is not satisfied, Egress-Prog passes the packet to the fallback overlay network. This is necessary because for a flow, the eviction of caches for two directions and the expiration of its conntrack entry are asynchronous. Utilizing the fast path only when caches for both directions are ready ensures that if any cache initialization is needed, conntrack can observe traffic in both directions and add *est* mark to packets, thereby ensuring the success of the initialization. See a counterexample in Appendix D.

Step #2: Encapsulating and intra-host routing. Having retrieved the egress cache, Egress-Prog proceeds to rewrite the inner MAC header and add the outer headers to the packet. Two modifications on the outer headers are needed: (1) Updating the length, ID, and checksum fields in the outer IP header, as well as the length field in the outer UDP header; (2) Calculating the outer UDP source port using the same hash function employed by the kernel. Finally, Egress-Prog invokes eBPF helper function `bpf_redirect` to redirect the packet to the cached host interface index.

3.3.2 The Ingress Fast Path

The ingress fast path is implemented by Ingress-Prog, which handles an ingress tunneling packet in the following steps:

Step #1: Destination check. Ingress-Prog first verifies whether destination MAC and IP addresses of the packet match with those of current host interface. TTL is also checked. If the destination check fails, the packet is then passed to the fallback overlay network for final decision.

Step #2: Cache retrieving. Then, Ingress-Prog checks if the packet's flow is whitelisted in the filter cache, and retrieves <*container dIP* → *inner MAC header, veth (host-side) index*> from the ingress cache. Any cache miss triggers a cache initialization process. Besides, the reverse check (described in §3.3.1) is also conducted on the ingress path.

Step #3: Decapsulating and intra-host routing. Ingress-Prog proceeds to strip the outer headers of the packet, rewrite the inner MAC header, and redirect the packet to the destination veth (container-side). The redirection is done by invoking the eBPF helper function `bpf_redirect_peer`.

Although the decapsulation of the outer headers is simplified in ONCache, no functionality is lost. Firstly, the payload is protected by checksums of the inner headers, which are verified by the container network stack. Secondly, the reassembly of fragmented packets is conducted by Generic Receive Offload (GRO) before reaching Ingress-Prog. Moreover, no other functionalities rely on the outer headers.

3.4 Cache Coherency

An overlay network is subject to change, necessitating ONCache to ensure cache coherency when changes occur. While container provisioning is inherently handled by cache initialization, ONCache handles other changes by the user space program, ONCache daemon.

Upon container deletion or unexpected container failures, ONCache daemon deletes the related caches. This prevents a new container with an old IP address from mistakenly utilizing outdated cache entries.

Upon other changes, including container migration or filter updates, ONCache employs a *delete-and-reinitialize* mechanism with four steps to ensure that the changes take effect immediately and properly in the fast path: (1) Pausing cache initialization by disabling OVS or netfilter from adding the *est* mark. (2) Removing the cache entries affected by the network change (the affected packets start using the fallback overlay network). (3) Applying the network change in the fallback overlay network (the network change takes effect immediately). (4) Resuming cache initialization (the cache entries are reinitialized and the affected packets start using the fast path again).

3.5 Cache Compatibility

Work with various traffic. ONCache is designed to accelerate inter-host container traffic and is not responsible for other types of traffic, such as intra-host container traffic, container-to-host-IP traffic, or container-to-external-IP traffic. These types of traffic are handled by the fallback overlay network.

The ClusterIP Service [62] in Kubernetes facilitates container access to a group of containers providing the same functionality via a single ClusterIP. Typically, such services rely on netfilter or IPVS (IP Virtual Server) [33] on the sender host for load balancing and Destination Network Address Translation (DNAT). However, ONCache's fast path bypasses netfilter and IPVS. Nevertheless, ONCache can support ClusterIP akin to Cilium's approach: implementing load balancing and DNAT by eBPF programs and maps [14]. This functionality can be integrated in Egress/Ingress-Prog and be compatible with the cache-based fast path.

Work with data-plane policies. Data-plane policies, such as rate limiting and quality of service (QoS) seamlessly integrate with ONCache. In Linux, these policies are implemented by queuing disciplines (qdiscs). ONCache's fast path does not bypass the qdiscs of the host interface.

Filters not supported by ONCache. Packet-based filters that match on the hash of the overlay packet [74] do not exhibit any invariance property. Besides, while all stateful filters in netfilter [2], OVS [51], Cilium [15] have an established state, there may exist stateful filters without an established state.

ONCache does not natively support such filters, because there is no invariant filter decision that can be cached. Nonetheless, it is feasible to implement such filters in eBPF case by case.

Work with service meshes. Service meshes are increasingly popular in container clusters such as Kubernetes, primarily deployed for functionalities like security, observability, and traffic management [63]. The most common implementations of service meshes employ software proxies called sidecars [73]. A sidecar is a separate process co-located with applications within the application network namespace (e.g., within a Kubernetes Pod [34]), and still relies on the overlay network for communication. Hence, ONCache benefits the communication of sidecar service meshes.

Compatibility with Container Network Interface (CNI). The current implementation of ONCache is based on TC eBPF. For CNIs that do not use TC eBPF in their data path, such as Antrea, Flannel, etc., ONCache can be seamlessly integrated without requiring any code modifications. Conversely, to use ONCache with CNIs that already utilize TC eBPF, such as Cilium, Calico, we need to reimplement ONCache within either the CNI's eBPF programs or the kernel source code.

Container live migration. ONCache is compatible with container live migration [19, 45]. With proper control plane operations (see §3.4), the container connections can be well-maintained. In contrast, Slim only supports cold migration [74], as its container connections are established on the host and become invalid after migration, disrupting the applications inside the containers.

Network debugging. ONCache supports ICMP, thereby enabling common debugging tools such as ping and traceroute. Users can also utilize tools like bpftool [6] to debug ONCache's eBPF programs and maps. Debugging with ONCache is easy and convenient. In contrast, debugging with Slim is much harder. Slim lacks support for ICMP, and there also lacks debug tools for its socket replacement mechanism.

Moreover, ONCache is compatible with Linux techniques such as segmentation offload [61], checksum offload [10], and scaling techniques [59]. See Appendix E for details.

3.6 Optional Improvements

We introduce two optional improvements for the fast path. Unlike the default design of ONCache, these optional improvements require either kernel or protocol modifications.

Optimizing the redirect data path. ONCache uses `bpf_redirect` and `bpf_redirect_peer` to redirect packets on the egress and ingress path, respectively. As shown in Figure 4 (a), the two redirect functions are asymmetric, and the egress redirect path fails to eliminate the namespaces traversing overhead. Our speculation is that a symmetric egress redirect (Figure 4 (b)) requires hooking networking eBPF programs in container namespaces, which may conflict with user applications. Nevertheless, we are still curious about the potential benefit of optimizing the egress redirect data path.

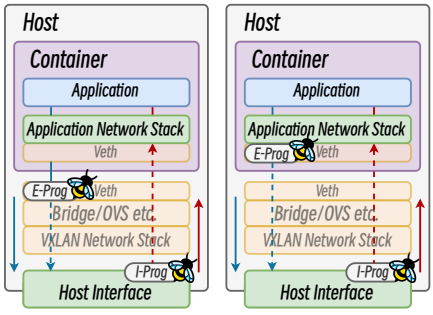


Figure 4: *The fast path in (a) ONCache without rpeer; (b) ONCache with rpeer.*

We design and implement a new eBPF redirect function in the Linux kernel, called `bpf_redirect_rpeer`. It provides a reversed path compared to `bpf_redirect_peer`, redirecting packets from the egress port of the veth (container-side) to the egress port of the host interface, as depicted in Figure 4 (b). By leveraging `bpf_redirect_rpeer`, the hook point of Egress-Prog changes to TC egress of the veth (container-side), enabling ONCache to further mitigate the namespaces traversing overhead on the egress path.

Optimizing the tunneling protocol. Traditional tunneling protocols like VXLAN introduce transmission overhead due to the outer headers, typically tens of bytes (e.g., 50 bytes for VXLAN). To eliminate this transmission overhead, we propose and implement a rewriting-based tunneling protocol.

With the new tunneling protocol, the fast path of ONCache works as below: (1) Egress-Prog modifies container source/destination IP/MAC addresses of an egress container packet to the host ones and writes a *restore key* to an idle field (e.g., DSCP) of the packet. Then the packet is redirected to the underlay network. (2) Ingress-Prog restores all the addresses to their original values according to the restore key and redirect the packet to the destination container. The mapping between the restore key and the container source/destination IP/MAC addresses is cached during cache initialization. See Appendix F for more detail.

4 Evaluation

We evaluate ONCache (without optional improvements by default) with microbenchmarks in §4.1 and popular distributed applications in §4.2. In both sections, ONCache's performance can be further improved by the optional improvements, at the cost of kernel/protocol modification. ONCache with optional improvements is evaluated in §4.3.

Our testbed is built on top of a base Kubernetes [35] cluster (v1.23.6), including API server, placement engine, etcd, etc. We run the experiments on Cloudlab [23] with three c6525-100g nodes, each equipped with an AMD EPYC 7402P 24-core processor@2.80 GHz. The processor uses hyperthreading. Each machine has 128 GB ECC memory and a dual-port Mellanox ConnectX-5 Ex 100 Gb interface. We do not

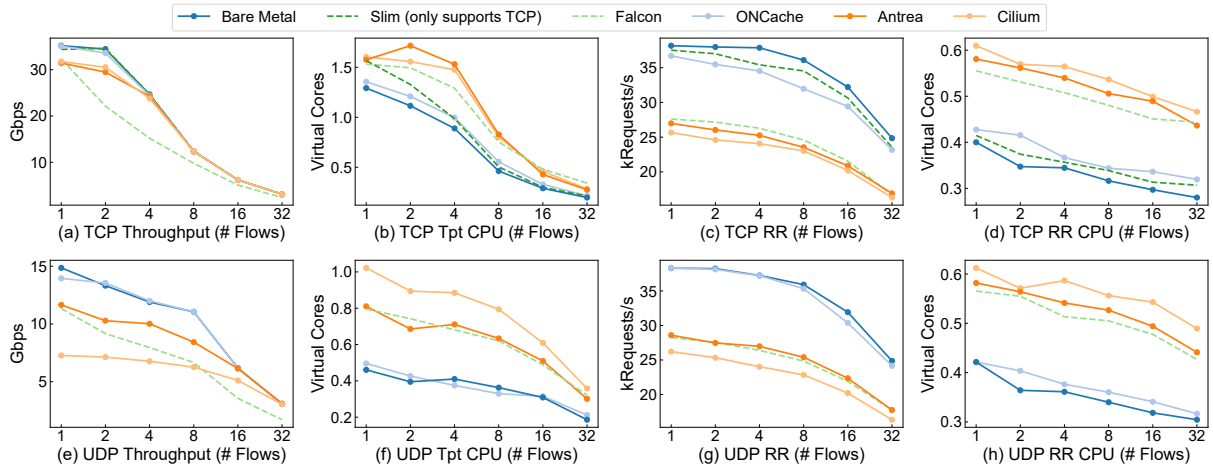


Figure 5: *TCP and UDP microbenchmark results of bare metal, Slim (only supports TCP), Falcon (Linux kernel v5.4), ONCache, Antrea and Cilium. Both Cilium and Antrea provide standard overlay networks. All data is the average of a single flow. CPU utilization is measured on the receiver host, normalized by throughput or RR, and scaled to Antrea's throughput or RR.*

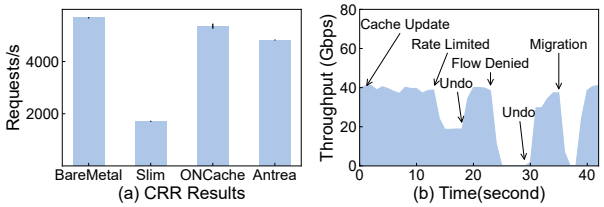


Figure 6: *(a) The Connect-Request-Response (CRR) rate of different networks. The higher the better. Error bars denote standard deviations. (b) iperf3 throughput for the functional completeness experiments.*

change Cloudlab's default configurations of segmentation offload [61], checksum offload [10], and scaling techniques [59]. We use Ubuntu 20.04 with Linux kernel version 5.14. ONCache is deployed as a plugin of the Antrea (encap mode).

4.1 Microbenchmarks

4.1.1 Throughput and Latency

We utilize iperf3 [32] to measure throughput and netperf [48] to measure Request-Response (RR) transaction rate. The RR test measures the rate of one-byte round-trips performed sequentially over a connection, wherein a higher transaction rate indicates lower end-to-end latency. We measure CPU utilization on the receiver host in all experiments using mpstat [46]. Both TCP and UDP tests are executed.

In all settings, we conduct parallel tests. Multiple container pairs are deployed on a pair of hosts, with all server containers residing on one host and all client containers on the other. Each container pair starts testing simultaneously, enabling the evaluation of ONCache's parallel performance.

To represent performance of standard overlay networks, which serves as baseline, we select the most widely used CNIs, Antrea [1] and Cilium [13]. We take bare metal as upper bound for overlay networks. To compare ONCache with prior works discussed in §2.3, we conduct microbenchmarks on Slim[5] and Falcon, which represent the ideas of socket replacement and CPU load balancing, respectively.

TCP microbenchmarks. ONCache significantly improves both throughput and RR performance while consuming less CPU resource. The per-flow throughput, RR, and CPU utilization results are shown in Figure 5 (a) – (d).

Compared to Antrea, ONCache improves TCP throughput by 11.53% and 13.96% in 1 and 2-parallel tests. In 4, 8, 16, and 32-parallel tests, all container networks saturate the 100 Gb physical network. ONCache reduces per-byte CPU utilization by 13.94% - 34.87% in different parallel tests. The throughput and CPU utilization of ONCache are very close to those of Slim and bare metal. The throughput of Falcon (only provides implementation in the kernel v5.4) is significantly lower. The reason is that the kernel v5.4 inherently exhibits lower bandwidth compared to the kernel v5.14 in our testbed.

RR performance of ONCache in all parallel tests outperforms Antrea by 35.81% to 40.91%, with a decrease in per-RR CPU utilization ranging from 26.02% to 32.03%. The RR results of ONCache exhibit a slight gap to those of Slim. Overhead, including egress namespace traversing, eBPF execution, and outer headers MTU overhead, contributes to this gap and can be mitigated by the optional improvements proposed in §3.6. As the RR test does not overwhelm any CPU cores, Falcon only slightly improves the RR results[6].

UDP microbenchmarks. In contrast to Slim, ONCache also benefits UDP traffic, as shown in Figure 5 (e) – (h). UDP throughput of ONCache is 19.68% - 31.76% higher than Antrea in 1, 2, 4, 8-parallel tests. In 16 and 32-parallel tests, the throughput remains the same due to bandwidth bottleneck. Moreover, per-byte CPU utilization is lower by 29.73% - 47.98% in all tests. The throughput gap between ONCache

[5] Slim does not support UDP, only the TCP microbenchmark is performed.
[6] In our testbed, the kernel version does not affect RR results.

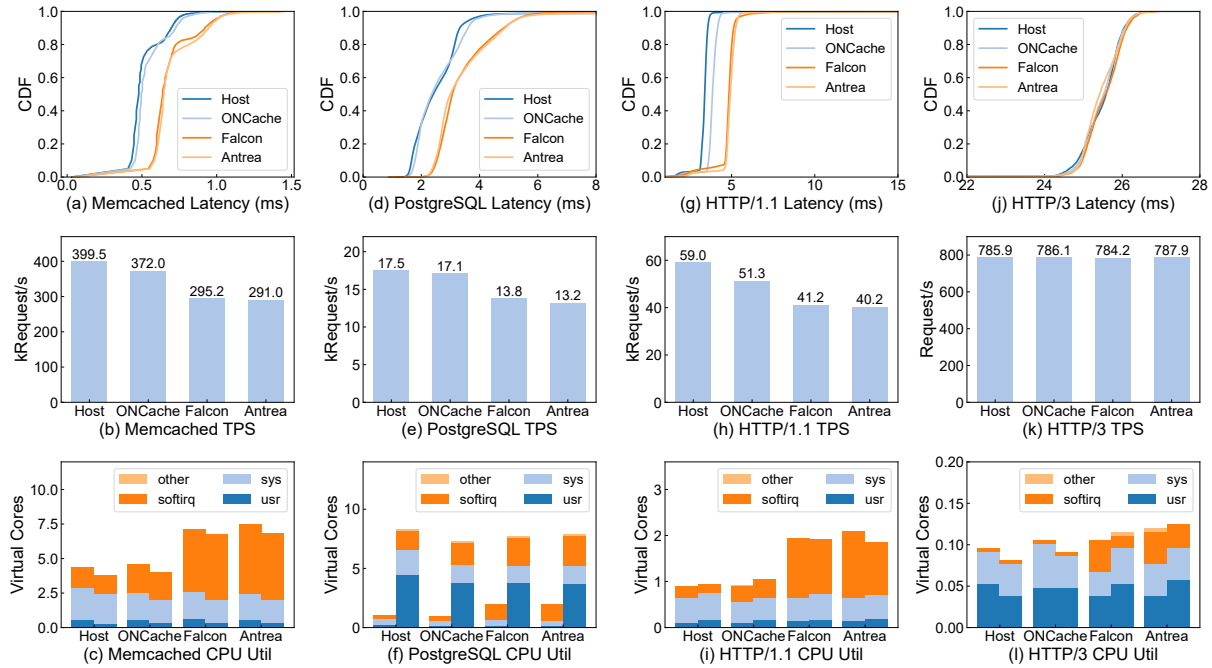


Figure 7: *The evaluation results of applications. Figures in each row present: the latency, the transaction rate (TPS) of all clients, and the CPU utilization of the client (left bar) & the server (right bar). To ensure a fair comparison, we normalize the CPU utilization by TPS and scale it to Antrea's TPS. Softirq stands for software interrupt request.*

and bare metal is less than 6%.

ONCache improves UDP RR performance by 34.13% - 39.12% in all tests, with per-RR CPU utilization reduced by 27.54% - 31.59%. There is also a small gap to Slim caused by the same reasons as in the TCP RR experiments.

Overhead quantifying. We quantify ONCache's overhead by the method proposed in §2.2 and present the results in Table 2. As expected, ONCache eliminates all the extra overhead, except for egress namespaces traversing overhead, which is addressed by `bpf_redirect_rpeer` proposed in §3.6. Although incurring eBPF execution overhead, ONCache still contributes to a significant reduction in overall network stack latency. The quantitative analysis aligns consistently with the microbenchmark results previously presented.

4.1.2 Cache Overhead

Cache initialization. We use netperf's Connect-Request-Response (CRR) test to measure TCP connection setup time and show the cache initialization overhead. In CRR, each RR transaction needs a new TCP connection, which requires additional cache initialization.

The CRR results are shown in Figure 6 (a). ONCache is better than Antrea but worse than bare metal. This is because ONCache relies on Antrea to handle the first 3 packets before caches are initialized. ONCache performs the same as Antrea in this part. Then the round trip of 1 byte (Request-Response part) goes through the fast path, which performs better than Antrea. Falcon does not benefit CRR compared to Antrea due to the same reason as in TCP RR. Slim performs significantly worse because it needs to first establish an overlay connection for service discovery, which incurs several extra RTTs.

Cache interference. We conduct an iperf3 test concurrently with continuous cache entry updates to show potential cache interference overhead. In this experiment, all cache capacities are set to 512, and the cache replacement policy is LRU. We use a script to continually insert 1000 redundant cache entries to the egress cache and subsequently delete them for 2 rounds. The test consumes about 8 seconds. As shown in the first 8 seconds of Figure 6 (b), the iperf3 flow exhibits no significant throughput fluctuation.

Cache scalability. We conduct a TCP RR test with a full egress cache containing 150,000 entries (for the largest Kubernetes cluster as mentioned in §3.1). As expected, the RR performance remains unaffected, showcasing the inherent scalability of hash maps.

4.1.3 Functional Completeness

Data-plane policies. We test whether data-plane policies, such as rate limiting, function correctly with ONCache. As shown in Figure 6 (b), in the absence of rate limiting, the throughput reaches about 39 Gbps. Then we limit the rate to 20 Gbps on the host interface and the throughput drops to about 18.5 Gbps. Upon removal of the rate limit, the throughput returns to its original level.

Packet filters. We test whether packet filters work properly with ONCache. During an iperf3 test with ONCache, we apply a simple packet filter which denies this iperf3 flow using the delete-and-reinitialize mechanism proposed in §3.4. As

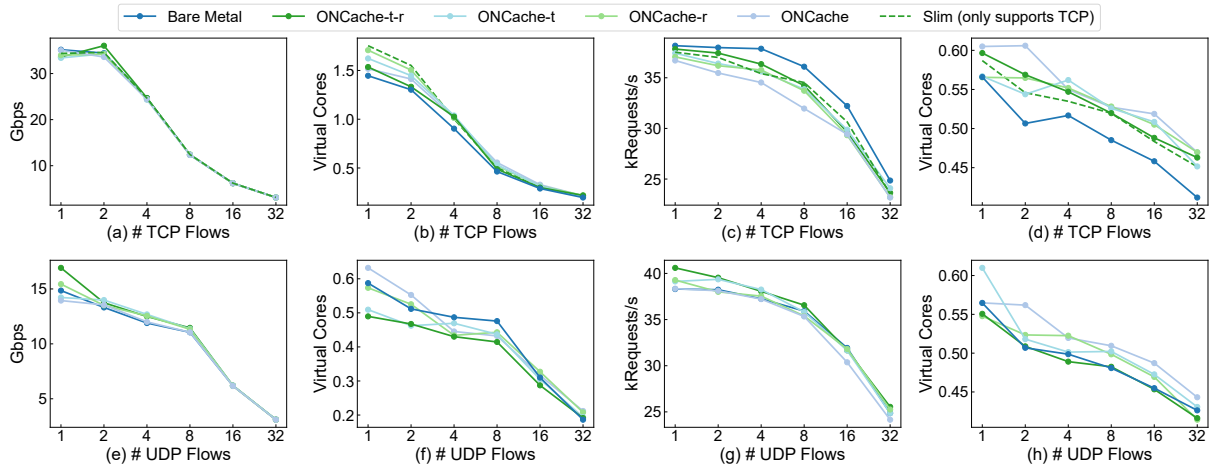


Figure 8: *TCP and UDP microbenchmark results of bare metal, ONCache with redirect rpeer (ONCache-r), ONCache with rewriting-based tunneling protocol (ONCache-t), ONCache with both (ONCache-t-r), and ONCache with neither (ONCache). The CPU utilization is normalized by throughput or RR and scaled to bare metal's throughput or RR.*

expected, the throughput drops to 0, as shown in Figure 6 (b). The throughput recovers upon removing the filter.

Container live migration. We evaluate whether ONCache works properly with container live migration. Since Kubernetes does not natively support live migration, we imitate it by modifying the host IP address and VXLAN tunnels while the container remains alive. As shown in Figure 6 (b), the throughput drops to 0 when migration starts (the host IP address is changed) and recovers once the migration finishes (VXLAN tunnels are updated) after about 2 seconds.

Both the packet filter and container live migration experiments induce network changes and require using the delete-and-reinitialize mechanism (proposed in §3.4) to apply the changes. Thus these experiments also demonstrate the effectiveness of the delete-and-reinitialize mechanism.

4.2 Applications

We evaluate ONCache on three real-world applications: in-memory key-value store Memcached [42], database PostgreSQL [55], and web server Nginx [50]. We take Antrea as performance baseline and Docker host network [21] as upper bound. Falcon is also evaluated. As Slim's performance is akin to that of the host network, it is omitted from this section. Notably, Slim only supports TCP, limiting its compatibility.

Memcached. We deploy a pair of containers on two hosts. One container runs a Memcached [42] server (v1.6), while the other runs a standard Memcached benchmark tool memtier [43] (v2.0.0) as a client. Upon the start of the experiment, the client spawns 4 threads, with each thread starting 50 connections. The ratio of SET and GET requests is 1:10. Memtier runs as fast as possible. We record CPU utilization by mpstat on both hosts during the experiment.

ONCache achieves much lower latency and higher transaction rate than Antrea. Figure 7 (a) is the CDF graph of the latency of GET requests. The average latency is reduced by 22.71%, while the 99.9 percentile latency is reduced by 27.69%. The latency gap relative to the host network is within 6%. The transaction rates are shown in Figure 7 (b). ONCache outperforms Antrea by 27.83%, while has a 7% gap compared to the host network.

ONCache also significantly reduces per-transaction CPU utilization, as shown in Figure 7 (c). The per-transaction CPU utilization of ONCache is 38.91% and 40.98% lower on the client and the server, respectively. The utilization remains 4.09% and 6.62% higher than that in the host network.

PostgreSQL. We deploy a PostgreSQL database [55] (v15.3) and its benchmark tool, pgbench [54], on a pair of containers on different hosts. Pgbench implements the TPC-B benchmark, creating a database with 5 million banking accounts and executing 50 clients concurrently. Pgbench runs as fast as possible. We measure CPU utilization by mpstat.

ONCache benefits PostgreSQL's latency and transaction rate, as shown in Figure 7 (d)(e). Compare to Antrea, ONCache reduces the average latency by 22.34% and the transaction rate by 29.40%, while having a gap of 2.30% and 2.54% to the host network, respectively. ONCache also reduces CPU utilization, as shown in Figure 7 (f). The per-transaction CPU utilization is reduced by 51.84% and 7.26% for the client and the server, respectively. The reduction of the server is smaller due to relative high user CPU utilization. ONCache reduces soft interrupt utilization on the server by 27.36%.

Nginx. To evaluate performance of HTTP servers, we conduct load tests on HTTP/1.1 and HTTP/3 [57], representing TCP-based and UDP-based HTTP, respectively. SSL is disabled for HTTP/1.1 and enabled for HTTP/3. We employ Nginx [50] (v1.25.1) and h2load [30] (v1.55.1) in the experiment. H2load runs 100 clients for HTTP/1.1 and 10 clients for HTTP/3. Each client issues 2 concurrent streams and requests a 1 KB file from the server. H2load runs as fast as possible.

ONCache improves latency and transaction rate for HTTP/1.1, as shown in Figure 7 (g) and (h). ONCache re-

duces HTTP/1.1 request latency by 21.53% and improves the transaction rate by 27.43%. There remains a performance gap between ONCache and the host network (15.04% for latency and 13.12% for transaction rate). The gap is caused by the same reason as analyzed in TCP RR, as the transaction rate is high enough to expose overhead. For HTTP/3, the performance is notably poorer and remains consistent across different networks, as shown in Figure 7 (j) and (k). This could potentially be attributed to the experimental QUIC support of Nginx [26].

ONCache reduces per-transaction CPU utilization for both HTTP/1.1 and HTTP/3, as shown in Figure 7 (i) and (l). For HTTP/1.1, CPU utilization is reduced by 56.26% for the client and 43.52% for the server. For HTTP/3, the reduction is 11.80% and 26.76% for the client and the server, respectively.

4.3 Evaluation for Optional Improvements

We evaluate ONCache with redirect rpeer (ONCache-r), rewriting-based tunneling protocol (ONCache-t), and both of them (ONCache-t-r) by the methods described in §4.1. The results are shown in Figure 8. The baseline in this section is ONCache without the optional improvements.

The optional improvements benefit RR transaction rate, as shown in Figure 8 (c)(g). ONCache-t, ONCache-r, and ONCache-t-r improve 1-parallel TCP RR performance by 1.96%, 0.97%, and 3.08%, respectively (2.04%, 2.43%, and 5.87% for UDP). For multi-parallel tests, the optional improvements also benefit RR performance. ONCache-t-r provides the most improvement in RR performance, nearly equaling the sum of those of ONCache-t and ONCache-r, and achieves nearly the same RR performance as Slim.

The optional improvements generally reduce CPU utilization, as shown in Figure 8 (b)(d)(f)(h). The CPU utilization reduction varies across different experiments, due to noise from other processes in the system.

We also evaluate the optional improvements with the applications used in §4.2. The results are presented in Appendix G.

5 Discussion

Why using TC hook? There are two main types of eBPF hooks for networking: XDP and TC [5]. Compared to XDP, TC eBPF programs do not require driver support, can redirect packets with lower overhead, are compatible with Linux traffic control (tc) module, and can be hooked on both ingress and egress ports. Therefore, TC is a better choice for ONCache.

Security of eBPF-based ONCache. Only processes that run by privileged user (root) or have the capability `CAP_BPF` can load eBPF programs and read/write eBPF maps, unless the privileged user enables unprivileged eBPF [25][7]. Thus, the eBPF components of ONCache are protected by the eBPF permission control. Additionally, ONCache's fast path resides in kernel mode and is inaccessible from applications. In contrast, Slim has serious security issues because it exposes host namespace file descriptors to containers, thereby breaking resource isolation provided by Linux namespaces. Slim's kernel module could mitigate the security issues, but resource isolation is still undermined.

[7] Although there are userspace eBPF implementations [68] that do not necessitate the privilege at all, they execute eBPF programs in userspace and differ from the eBPF adopted by ONCache.

6 Related Works

Cache in network virtualization. OVS [53, 58] employs a cache to accelerate flow matching. However, as analyzed in §2.2, there is still significant extra overhead in overlay networks. Andromeda [20], a VM cluster network virtualization stack, maintains a cache of routing and filtering decisions, sharing a similar idea with OVS. Nevertheless, none of the prior works quantify container overlay networks and employ a cross-layer cache to bridge the performance gap between container overlay networks and bare metal, as ONCache does.

eBPF-based container networks. ONCache differs significantly from existing eBPF-based container networks like Cilium [13] and Calico [7]. For example, in Cilium, the eBPF-based data path aims to replace netfilter in the host network stack with eBPF programs and is effective where container packets are directly forwarded to underlay network [17]. However, as analyzed in §2.2, Cilium fails to improve overlay networks and the extra overhead still exists. In contrast, ONCache aims to eliminate the extra overhead with the proposed cache, as shown in Table 2.

Service mesh sidecar optimization. While a sidecar simplifies traffic management, policy enforcement, network security, etc., it incurs significant overhead. Zhu et al. [73] proposed MeshInsight to quantify the overhead of service mesh sidecars, and works such as SPRIGHT [56], mRPC [11] optimize sidecars by different mechanisms. However, these works focus on optimizing the overall intra-host service mesh architecture, rather than the inter-host container communication.

7 Conclusion

We design and implement ONCache, a cache-based container overlay network that effectively narrows the performance gap between bare metal and overlay networks without losing flexibility and compatibility. In both microbenchmark and application experiments, ONCache significantly outperforms standard overlay networks in terms of throughput and latency while reducing CPU utilization. ONCache can be seamlessly integrated with existing CNIs such as Antrea, Flannel, etc.

Acknowledgements. We sincerely thank the anonymous reviewers for their helpful comments. This work was supported by the NSF China (No. 62272292).

References

[1] Antrea. https://antrea.io/.

[2] P. Ayuso. Netfilter's connection tracking system. *Login: The Usenix Magazine*, 31:34–39, 2006.

[3] Bcc. https://github.com/iovisor/bcc.

[4] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[5] Bpf and xdp reference. https://docs.cilium.io/en/stable/bpf/.

[6] bpftool. https://github.com/libbpf/bpftool.

[7] Calico. https://www.tigera.io/project-calico/.

[8] Emiliano Casalicchio. Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*, pages 221–235, 2019.

[9] Emiliano Casalicchio and Stefano Iannucci. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*, 32(17):e5668, 2020.

[10] Checksum offloads. https://docs.kernel.org/networking/checksum-offloads.html.

[11] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 141–159, 2023.

[12] Sunyanan Choochotkaew, Tatsuhiro Chiba, Scott Trent, and Marcelo Amaral. Bypass container overlay networks with transparent bpf-driven socket replacement. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 134–143. IEEE, 2022.

[13] Cilium. https://cilium.io/.

[14] Cilium service load-balancing. https://docs.cilium.io/en/stable/network/kubernetes/intro/#service-load-balancing.

[15] cilium/bpf/lib/conntrack.h. https://github.com/cilium/cilium/blob/main/bpf/lib/conntrack.h.

[16] Cilium - ebpf datapath introduction. https://docs.cilium.io/en/stable/network/ebpf/intro/.

[17] Cni benchmark: Understanding cilium network performance. https://cilium.io/blog/2021/05/11/cni-benchmark.

[18] The state machine. https://rlworkman.net/howtos/iptables/chunkyhtml/c1276.html.

[19] Criu. https://criu.org/Main_Page.

[20] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 373–387, 2018.

[21] Docker. https://www.docker.com/.

[22] Docker swarm. https://docs.docker.com/get-started/orchestration/.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[24] ebpf. https://ebpf.io/.

[25] ebpf safety. https://ebpf.io/what-is-ebpf/#ebpf-safety.

[26] Module ngx_http_v3_module. https://nginx.org/en/docs/http/ngx_http_v3_module.html.

[27] brendangregg/flamegraph. https://github.com/brendangregg/FlameGraph.

[28] Flannel. https://github.com/flannel-io/flannel.

[29] Jesse Gross, Ilango Ganga, and T. Sridhar. Geneve: Generic network virtualization encapsulation. RFC 8926, November 2020.

[30] h2load. https://nghttp2.org/documentation/h2load-howto.html.

[31] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[32] iperf. https://github.com/esnet/iperf.

[33] Ipvs. http://www.linuxvirtualserver.org/software/ipvs.html.

[34] Sidecar containers. https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/.

[35] Kubernetes. https://kubernetes.io/.

[36] Jiaxin Lei, Manish Munikar, Hui Lu, and Rao Jia. Accelerating packet processing in container overlay networks via packet-level parallelism. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 79–89. IEEE, 2023.

[37] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. Parallelizing packet processing in container overlay networks. *EuroSys 2021*, 2021.

[38] Jiaxin Lei, Kun Suo, Hui Lu, and Jia Rao. Tackling parallelization challenges of kernel network stack for container overlay networks. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[39] Fusheng Lin, Xin Zhang, Guo Chen, Li Chen, Kenli Li, and Hongbo Jiang. Slim and fast: Low-overhead container overlay network with fast connection setup. *IEEE Transactions on Cloud Computing*, 2023.

[40] Hao Liu, Wenxin Li, Yiren Pang, Renjie Pei, Yitao Hu, Yuan Liu, Lide Suo, and Keqiu Li. Accelerating data delivery of latency-sensitive applications in container overlay network. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[41] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Larry Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. Virtual eXtensible Local Area Network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, August 2014.

[42] memcached. https://memcached.org/.

[43] memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.

[44] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for ebpf-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021.

[45] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[46] Mpstat. https://linux.die.net/man/1/mpstat.

[47] netfilter. https://www.netfilter.org/.

[48] Netperf. https://hewlettpackard.github.io/netperf/.

[49] netpipe(1) - linux man page. https://linux.die.net/man/1/netpipe.

[50] Nginx. https://nginx.org/.

[51] Ovs conntrack tutorial. https://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/.

[52] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[53] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vSwitch. In *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, pages 117–130, 2015.

[54] pgbench. https://www.postgresql.org/docs/current/pgbench.html.

[55] Postgresql. https://www.postgresql.org/.

[56] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 780–794, 2022.

[57] Quic. https://www.chromium.org/quic/.

[58] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. Scaling open vSwitch with a computational cache. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1359–1374, 2022.

[59] Scaling in the linux networking stack. https://docs.kernel.org/networking/scaling.html.

[60] Considerations for large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/.

[61] Segmentation offloads. https://docs.kernel.org/networking/segmentation-offloads.html.

[62] Service. https://kubernetes.io/docs/concepts/services-networking/service/.

[63] Service meshes are on the rise — but greater understanding and experience are required. https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf.

[64] Single root i/o virtualization. https://pcisig.com/specifications/iov/single_root/.

[65] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 189–197. IEEE, 2018.

[66] The kubernetes network model. https://kubernetes.io/docs/concepts/services-networking/.

[67] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the ACM SIGCOMM 2021 Conference*, pages 245–257, 2021.

[68] iovisor/ubpf. https://github.com/iovisor/ubpf.

[69] veth. https://man7.org/linux/man-pages/man4/veth.4.html.

[70] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[71] Virtual ethernet device optimization with ebpf. https://cilium.io/blog/2020/11/10/cilium-19/#veth.

[72] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, 2023.

[73] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, et al. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 142–157, 2023.

[74] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS kernel support for a Low-Overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 331–344, 2019.

A Quantitatively Analyzing Network Stacks

In this section, we describe how we quantitatively analyze network stacks. We analyze the overhead incurred during the execution of a 1-byte TCP RR test conducted using NPtcp [49].

Firstly, we perform a qualitative analysis of overhead in different data paths with two steps: (1) Utilizing the perf tool [52], we profile network stack functions for a duration of 10 seconds and subsequently employ a flame graph generator [27] to generate flame graphs. (2) According to the flame graphs, we classify network stack functions into different segments of the overhead, e.g., the application network stack, OVS, etc.

Next, we time the network stack functions using eBPF one by one. As the network stack functions may execute in both the application network stack and the VXLAN network stack, we need to distinguish them by their call stacks. We implement two eBPF programs leveraging the BPF Compiler Collection (BCC) [3] tool. These programs are hooked to `kprobes` and `kretprobes` hook points, executing at the entry and return points of each specific function, respectively. They capture the current call stack along with the timestamp, and store the function (with the call stack) and its execution time in an eBPF map. For each function, we record all the execution time samples within one second and compute the average of these samples, which serves as the final result, as shown in Table 2.

B The Core Code of ONCache

In this section, we present core code of ONCache's eBPF programs to help to understand the design of ONCache.

B.1 The Local Cache Definition

The code in this section implements functionalities that are described in §3.1.

The egress cache:

```
struct egressinfo {
    unsigned char outer_header[64];
    __u32 ifidx;
};
struct bpf_elf_map SEC("maps") egressip_cache = {
    .type = BPF_MAP_TYPE_LRU_HASH,
    .size_key = sizeof(__be32),
    .size_value = sizeof(__be32),
    .pinning    = PIN_GLOBAL_NS,
    .max_elem = 4096
};
struct bpf_elf_map SEC("maps") egress_cache = {
    .type = BPF_MAP_TYPE_LRU_HASH,
    .size_key = sizeof(__be32),
    .size_value = sizeof(struct egressinfo),
    .pinning    = PIN_GLOBAL_NS,
    .max_elem = 1024,
};
```

The ingress cache.

```
struct ingressinfo {
    __u32 ifidx;
    unsigned char dmac[ETH_ALEN];
    unsigned char smac[ETH_ALEN];
};
struct bpf_elf_map SEC("maps") ingress_cache = {
    .type = BPF_MAP_TYPE_LRU_HASH,
    .size_key = sizeof(__be32),
    .size_value = sizeof(struct ingressinfo),
    .pinning    = PIN_GLOBAL_NS,
    .max_elem = 1024,
};
```

The filter cache.

```
struct action {
    __u16 ingress;
    __u16 egress;
};
struct bpf_elf_map SEC("maps") filter_cache = {
    .type = BPF_MAP_TYPE_LRU_HASH,
    .size_key = sizeof(struct fivetuple),
    .size_value = sizeof(struct action),
    .pinning    = PIN_GLOBAL_NS,
    .max_elem = 4096,
};
```

B.2 Cache Initialization

The code in this section implements functionalities that are described in §3.2.

Initialize the Egress Path. `TC_ACT_OK` indicates the kernel to proceed with the usual packet processing. ONCache uses `TC_ACT_OK` to pass the packet to the fallback overlay network.

```
// Checks if miss and est marked.
if ((inner_iph->tos & 0xc) != 0xc) return TC_ACT_OK;
// Update filter cache
struct fivetuple tuple_;
if (parse_5tuple_e(inner_iph, data_end, &tuple_)) return
     TC_ACT_OK;
struct action eaction_ = {
    .egress = 1,
    .ingress = 0
};
if(bpf_map_update_elem(&filter_cache, &tuple_, &eaction_,
     BPF_NOEXIST)) {
    struct action* action_ =
     bpf_map_lookup_elem(&filter_cache, &tuple_);
    if (action_) action_->egress = 1;
}
// Update egress cache
struct egressinfo egressinfo_;
initegressinfo(&egressinfo_, skb);
if(bpf_map_update_elem(&egress_cache, &outer_iph->daddr,
     &egressinfo_, BPF_NOEXIST))
    return TC_ACT_OK;
if(bpf_map_update_elem(&egressip_cache, &inner_iph->daddr,
     &outer_iph->daddr, BPF_NOEXIST))
    return TC_ACT_OK;
// Erase the TOS mark.
set_ip_tos(skb, 50, 0);
```

```
// Drop packets reported as invalid by conntrack.
table=ConntrackState, priority=210,ct_state=+inv+trk,ip actions=drop
// Forwarded tracked Service packets
table=ConntrackState, priority=200,ct_state=-new+trk,ct_mark=0x10/0x10,ip\
    actions=load:0x1->NXM_OF_IP_TOS[3],load:0x1->NXM_NX_REG0[9],\
    resubmit(,AntreaPolicyEgressRule)
// Forwarded tracked non-Service packets
table=ConntrackState, priority=190,ct_state=-new+trk,ip \
    actions=load:0x1->NXM_OF_IP_TOS[3],resubmit(,AntreaPolicyEgressRule)
// Match the first packet of untracked connection and forward it
table=ConntrackState, priority=0 actions=resubmit(,PreRoutingClassifier)
```

Figure 9: *Parts of OVS flows in Antrea. The action that we introduce is highlighted in red font, through which we set a predefined DSCP bit to 1 if the flow reaches established state.*

Initialize the Ingress Path.

```
// Checks if miss and est marked.
if ((iphdr->tos & 0xc) != 0xc) return TC_ACT_OK;
// Update ingress cache
struct ingressinfo* ingressinfo_ =
     bpf_map_lookup_elem(&ingress_cache, &iphdr->daddr);
if (!ingressinfo_) {
    return TC_ACT_OK;
} else {
    __builtin_memcpy(ingressinfo_->dmac, eth->h_dest,
     ETH_ALEN);
    __builtin_memcpy(ingressinfo_->smac, eth->h_source,
     ETH_ALEN);
}
// Update filter cache
struct fivetuple tuple_;
if (parse_5tuple_in(iphdr, data_end, &tuple_)) return
     TC_ACT_OK;
struct action eaction_ = {
    .egress = 0,
    .ingress = 1
};
if(bpf_map_update_elem(&filter_cache, &tuple_, &eaction_,
     BPF_NOEXIST)) {
    struct action* action_ =
     bpf_map_lookup_elem(&filter_cache, &tuple_);
    if (action_) action_->ingress = 1;
}
// Erase the TOS mark.
set_ip_tos(skb, 0, 0);
```

Extra configuration on OVS or netfilter. The addition of the *est* mark can be realized through either OVS or netfilter.

For OVS, this action requires modifying two OVS flows, as shown in Figure 9. Originally, the two flows are intended for forwarding non-new-state tracked packets. We utilize them to add the *est* mark to packets meanwhile. The actions that we introduce are highlighted in red font, through which we set a predefined DSCP bit to 1 if the flow reaches established state.

Alternatively, this action can be independently realized by adding a new rule in netfilter as follows.

```
iptables -t mangle -A FORWARD -m conntrack --ctstate
     ESTABLISHED -m dscp --dscp 0x1 -j DSCP --set-dscp 0x3
```

B.3 Cache-based Fast Path

The code in this section implements functionalities that are described in §3.3.

B.3.1 The Egress Data Path

Step #1: Cache retrieving.

```
struct fivetuple tuple_;
if (parse_5tuple_e(iphdr, data_end, &tuple_)) return
     TC_ACT_OK;
struct action *action_ =
     bpf_map_lookup_elem(&filter_cache, &tuple_);
if (!action_ || !(action_->ingress & action_->egress)) {
    set_ip_tos(skb, 0, 0x4);
    return TC_ACT_OK;
}
__be32* nodeip_ = bpf_map_lookup_elem(&egressip_cache,
     &iphdr->daddr);
if (!nodeip_) {
    // TOS 0x4 is used as miss mark
    set_ip_tos(skb, 0, 0x4);
    return TC_ACT_OK;
}
struct egressinfo* egressinfo_ =
     bpf_map_lookup_elem(&egress_cache, nodeip_);
if (!egressinfo_) {
    set_ip_tos(skb, 0, 0x4);
    return TC_ACT_OK;
}
struct ingressinfo* ingressinfo_ =
     bpf_map_lookup_elem(&ingress_cache, &iphdr->saddr);
if (!ingressinfo_ || !ingressinfo_complete(ingressinfo_))
    return TC_ACT_OK;
```

Step #2: Encapsulating and intra-host routing.

```
// Prepend the outer headers
if (bpf_skb_adjust_room(skb, 50, BPF_ADJ_ROOM_MAC,
     BPF_F_ADJ_ROOM_FIXED_GSO |
     BPF_F_ADJ_ROOM_ENCAP_L3_IPV4 |
     BPF_F_ADJ_ROOM_ENCAP_L4_UDP|
     BPF_F_ADJ_ROOM_ENCAP_L2(14)|
     BPF_F_ADJ_ROOM_ENCAP_L2_ETH)))
    return TC_ACT_OK;
// Boundary check
...
// Headers includes 50B of outer hdr and 14B inner MAC hdr.
__builtin_memcpy(data, egressinfo_->headers, 64);
set_lengthandid(skb, skb->len);
// Set the UDP source port
__u32 hash = bpf_get_hash_recalc(skb);
__be16 sport = get_udpsport(hash);
bpf_skb_store_bytes(skb, UDP_PORT_OFF, &sport,
     sizeof(sport), 0);
// Packet redirecting
action = bpf_redirect(egressinfo_->ifidx, 0);
return action;
```

B.3.2 The Ingress Data Path

Step #1: Destination check.

```c
struct bpf_elf_map SEC("maps") devmap = {
    .type = BPF_MAP_TYPE_HASH,
    .size_key = sizeof(int),
    .size_value = sizeof(struct devinfo),
    .pinning    = PIN_GLOBAL_NS,
    .max_elem = 8,
};

struct ethhdr *outer_eth = data;
int ifindex = ctx->ifindex;
struct devinfo *devinfo_ = bpf_map_lookup_elem(&devmap,
    &ifindex);
if (!devinfo_ || maccmp(outer_eth->h_dest, devinfo_->mac,
    ETH_ALEN)) return TC_ACT_OK;
// Check if Ethernet frame has IP packet and set IP hdr ptr
if (outer_eth->h_proto != bpf_htons(ETH_P_IP)) return
    TC_ACT_OK;
struct iphdr *outer_iph = (struct iphdr *)(outer_eth + 1);
if (outer_iph->daddr != devinfo_->ip) return TC_ACT_OK;
```

Step #2: Cache retrieving.

```c
struct fivetuple tuple_;
if (parse_5tuple_in(inner_iph, data_end, &tuple_)) return
    TC_ACT_OK;
struct action *action_ =
    bpf_map_lookup_elem(&filter_cache, &tuple_);
if (!action_ || !(action_->ingress & action_->egress)) {
    set_ip_tos(skb, 50, 0x4);
    return TC_ACT_OK;
}
struct ingressinfo* ingressinfo_ =
    bpf_map_lookup_elem(&ingress_cache,
    &inner_iph->daddr);
if (!ingressinfo_ || !ingressinfo_complete(ingressinfo_)) {
    set_ip_tos(skb, 50, 0x4);
    return TC_ACT_OK;
}
if (!bpf_map_lookup_elem(&egressip_cache,
    &inner_iph->saddr)) return TC_ACT_OK;
```

Step #3: Decapsulating and intra-host routing.

```c
if (bpf_skb_adjust_room(ctx, -50, BPF_ADJ_ROOM_MAC, 0))
    return TC_ACT_OK;
// Boundary check
...
// Set new MAC
struct ethhdr *eth = data;
__builtin_memcpy(eth->h_dest, ingressinfo_->dmac,
    ETH_ALEN);
__builtin_memcpy(eth->h_source, ingressinfo_->smac,
    ETH_ALEN);
// Packet redirecting
action = bpf_redirect_peer(ingressinfo_->ifidx, 0);
return action;
```

C Calculation of the Maps Size

As per the cache definitions described in §3.1, the sizes of cache entries are specified as below: 8 bytes for the egress cache (first level), 72 bytes for the egress cache (second level), 20 bytes for the ingress cache, and 20 bytes for the filter cache.

To eliminate cache eviction by the LRU mechanism for a cluster with a maximum of 110 containers per host, 5k hosts, 150k total containers, and up to 1M concurrent flows per host, we need the egress cache (first level) to have 150k entries, the egress cache (second level) to have 5k entries, the ingress cache to have 110 entries, and the filter cache to have 1M entries.

Consequently, the sizes of the caches can be calculated as follows: the egress cache is 8 B * 150k + 72 B * 5k = 1.56 MB, the ingress cache is 20 * 110 = 2.2 KB, and the filter cache is 20 B * 1M = 20 MB.

D An Example to Understand Necessity of Reverse Check

We further use a counterexample to show the necessity of reverse check.

Let's consider a scenario where ONCache only checks the egress cache on the egress data path. Initially, flow f enters the established state in conntrack, and all caches for f are initialized. Consequently, all packets of flow f go through ONCache's fast path. However, since flow f bypasses the conntrack module in the fallback overlay network, its conntrack entry eventually expires after a predefined duration. Subsequently, if flow f is evicted from the ingress cache by LRU mechanism, it can still leverage the egress fast path but becomes ineligible for the ingress fast path. Ideally, we envision ONCache to reinitialize the ingress cache for flow f. But unfortunately, flow f cannot re-enter the established state in conntrack because conntrack records a flow as established only upon observing packets in both directions [2]. Consequently, the ingress cache for flow f can never be reinitialized and flow f can no longer utilize the ingress fast path.

E Compatibility Discussions

Segmentation offloads. Segmentation offloads, such as Generic Segmentation Offload (GSO) and Generic Receive Offload (GRO) [61], enable network interfaces to segment and reassemble packets. The hooking points of all ONCache eBPF programs are TC. On the egress path, GSO happens after TC [5]. Thus GSO and other hardware segmentation mechanisms of the host interface are not bypassed by the fast path. On the ingress path, GRO happens prior to TC [5], posing no conflicts with ONCache. Hence, ONCache is compatible with segmentation offload techniques.

Scaling techniques. Linux scaling techniques, including Receive Packet Steering (RPS) and Receive Flow Steering (RFS) (and their hardware counterparts Receive Side Scaling (RSS) and Accelerated RFS (aRFS)) [59], employ different CPU load balancing strategies on the ingress data path. They are all compatible with ONCache: RSS and aRFS are hardware mechanisms; RPS and RFS are in software but happen

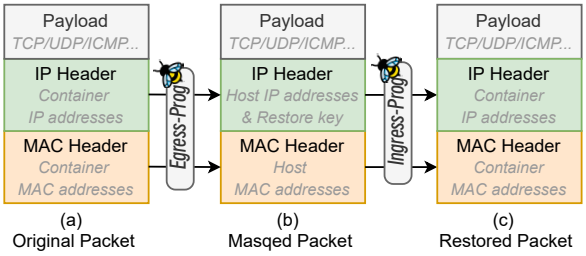


Figure 10: *The changes of a container packet forwarded by ONCache with the rewriting-based tunneling protocol. (a) The original packet from containers. (b) The masqueraded packet. (c) The restored packet.*

before Ingress-Prog on the ingress path.

Checksum offloads. Checksum offloads, including transmitting and receiving checksum offload [10], are often enabled to reduce checksum overhead. As they are hardware mechanisms, ONCache is compatible with them.

F Rewriting-based Tunneling Protocol

This section describes how the rewriting-based tunneling protocol works in ONCache.

The rewriting-based tunneling protocol operates by substituting the encapsulation of outer headers in traditional tunneling protocols, with the rewriting of inner headers. The modifications to packet headers along the data path are depicted in Figure 10.

New caches. The egress and ingress caches are redesigned to accommodate the rewriting-based tunneling protocol.

The egress cache stores data used on the egress path, structured as <*container sdIP → host interface index, host sdIP, host sdMAC, restore key*>[8].

The ingress cache stores data used on the ingress path, structured as <*host sIP & restore key → veth (host-side) index, container sdIP, container dMAC*>. To accommodate the new initialization process described later, the ingress cache is divided into two maps. The first is the ingressIP cache: <*host sIP & restore key → container sdIP*>, while the second is the ingress cache: <*container dIP → container dMAC, veth (host-side) index*>.

The new packet journey. On the sender host, Egress-Prog retrieves <*container sdIP → host interface index, host sdIP, host sdMAC, restore key*> from the egress cache. Subsequently, Egress-Prog conducts the *masquerading* process by modifying the container source/destination MAC/IP addresses to those of the host.

To ensure the masqueraded packet can be restored on the receiver host, ONCache additionally writes a restore key to the packet. Then the receiver host utilizes the restore key in conjunction with the host IP addresses to restore container source/destination MAC/IP addresses. The restore key is allocated during cache initialization and is stored in the egress cache. The user has the flexibility to designate a specific field to accommodate the restore key within the packet. This field can be any available field in the IP header, such as ID, DSCP, or Option. The capacity of the restore key depends on the width of the chosen field. After writing the restore key to the packet, referred to as the masqueraded packet, Egress-Prog redirects it to the host interface using either `bpf_redirect` or `bpf_redirect_rpeer`.

On the receiver host, Ingress-Prog restores the masqueraded packet. Ingress-Prog retrieves <*host sIP & restore key → veth (host-side) index, container sdIP, container dMAC*> from the ingress cache, and writes the container IP and MAC addresses back to the headers. After restoring, the packet is redirected to the destination veth using `bpf_redirect_peer`.

The new cache initialization. The initialization for MAC and IP addresses, as well as interface indexes, remains akin to the original design (as described in §3.2), and the allocation of the restore key is newly added. The restore key is allocated on the receiver host and subsequently delivered to the sender host. This ensures that the masquerading operation utilizes the restore key only after the receiver host recognizes it. The initialization of the filter cache is the same as that described in §3.2 and is therefore omitted in this section. ONCache utilizes a round-trip of tunneling packets to initialize the cache. The process is shown in Figure 11 and encompasses four steps:

Step #1 (tagged in Figure 11 ①): Upon encountering a cache miss, Egress-Prog passes the packet to the fallback overlay network. Then the packet reaches Egress-Init-Prog after undergoing processing within the fallback overlay network. Egress-Init-Prog parses the headers and stores <*container sdIP → host interface index, host sdIP, host sdMAC*> to the egress cache. It also allocates a restore key for the reverse flow, chosen randomly or sequentially, and writes <*host sIP & restore key → container sdIP*> to the ingressIP cache. As a hash map, the ingressIP cache naturely ensures the uniqueness of the restore key. To deliver the restore key to the peer host, Egress-Init-Prog writes the restore key into the user-defined field within the inner header. Finally, the packet is transmitted to the underlay network.

Step #2 (tagged in Figure 11 ②): Upon receiving a tunneling packet on the receiver host, Ingress-Init-Prog continues the initialization. It parses the restore key contained within the packet, and stores <*container sdIP → restore key*> to the egress cache. Next, it stores <*container dIP → container dMAC*> to the ingress cache (<*container dIP → veth (host-side) index*> is maintained by the user space daemon). Finally, the packet is forwarded to the destination application.

The above two steps fill up half of fields within the egress cache and ingress cache on both hosts. Subsequently, the reply of this tunneling packet goes through a similar yet reverse

[8] sIP = source IP address; dIP = destination IP address. Similarly, for sMAC, dMAC, sdIP, and sdMAC.

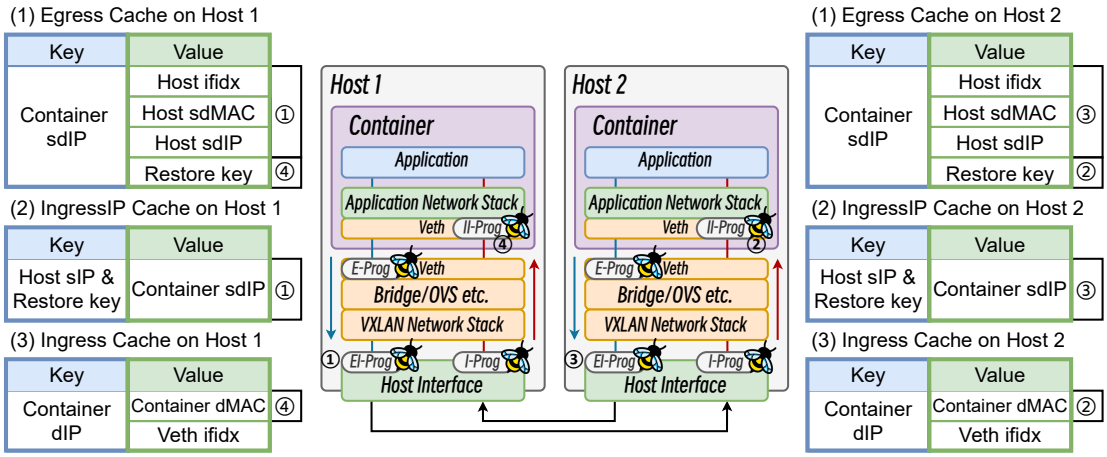


Figure 11: *The cache initialization of ONCache with the rewriting-based tunneling protocol.* ① - ④ *mark four steps where parts of the caches are filled up. ifidx is short for interface index.*

Results		ONCache-t	ONCache-r	ONCache-t-r	Host	ONCache
Memcached	Latency	-2.94%	-7.01%	-6.04%	-5.38%	0%
	TPS	+2.97%	+1.74%	+4.32%	+7.42%	0%
	CPU	+1.92%	+2.14%	-1.10%	-6.21%	0%
PostgreSQL	Latency	+0.04%	-1.28%	-5.77%	-2.25%	0%
	TPS	-0.27%	+1.23%	+6.21%	+2.61%	0%
	CPU	-1.86%	-4.61%	-10.36%	+13.17%	0%
HTTP/1.1	Latency	-2.82%	-8.46%	-10.00%	-13.08%	0%
	TPS	+2.78%	+9.09%	+10.90%	+15.10%	0%
	CPU	-5.48%	-11.93%	-14.02%	-11.26%	0%
HTTP/3	Latency	0.00%	+0.16%	+0.16%	+0.04%	0%
	TPS	-0.03%	-0.03%	-0.01%	-0.03%	0%
	CPU	-26.30%	+15.82%	-10.51%	-10.50%	0%

Table 4: *The application performance and server CPU utilization (normalized by TPS) of ONCache-t, ONCache-r, ONCache-t-r, and the host network compared to ONCache.*

process (tagged in Figure 11 ③④), thereby completing the entire initialization process.

G Application Evaluation for Optional Improvements

This section, we show the application evaluation results for the optional improvements. The evaluation results of ONCache-r, ONCache-t, ONCache-t-r, and the host network are listed in Table 4. The listed values are relative to ONCache.

The application evaluation reveals the following key findings: (1) Both the rewriting-based tunneling protocol (ONCache-t) and `bpf_redirect_rpeer` (ONCache-r) improve latency and TPS performance across all applications, except for HTTP/3. They also reduce per-transaction CPU utilization for PostgreSQL and HTTP/1.1. (2) With both optional improvements, ONCache-t-r generally achieves the best latency, TPS, and CPU utilization across all applications, except for HTTP/3. Notably, the performance of ONCache-t-r closely rivals that of the host network.

Due to the constraints of the experimental QUIC support of Nginx, the absolute result values of HTTP/3 across different implementations exhibit slight variations. Consequently, the evaluation results of HTTP/3 are inconclusive.

GREEN: Carbon-efficient Resource Scheduling for Machine Learning Clusters

Kaiqiang Xu[1], Decang Sun[1], Han Tian[2], Junxue Zhang[1], Kai Chen[1]
[1]*iSING Lab, Hong Kong University of Science and Technology* [2]*USTC*

Abstract

This paper explores the problem of scheduling machine Learning (ML) jobs while also taking into account the reduction of carbon emissions in the cluster. Traditional cluster schedulers for ML jobs mainly focus on optimizing job completion time (JCT), but do not consider the environmental impact of their decisions, resulting in a suboptimal carbon footprint. To address this issue, we propose GREEN, an ML cluster scheduler that is both time-efficient and carbon-efficient. At its core, GREEN uses a unique carbon-aware scheduling algorithm that reduces carbon footprint with minimized impact on JCT. Additionally, it leverages the temporal flexibility of ML jobs to reduce carbon emissions by shifting workloads to less carbon-intensive times, while still maintaining overall daily capacity. Our experiments using real ML jobs workload demonstrate that GREEN can achieve up to 41.2% reduction in cluster-wide carbon footprint and 12% reduction in peak power consumption, while incurring 3.6%-5.9% time efficiency tradeoff compared to existing methods.

1 Introduction

Machine Learning (ML) workloads currently constitute 8% of the 54 GW global data center demand and are projected to increase to 15-20% by 2028 [3]. Current production clusters handle a large volume of ML jobs, supported by fast parallel computing infrastructures using GPU devices. These setups, termed ML-as-a-Service (MLaaS) clusters, enhance utilization and reduce costs by allowing multiple users to share resources [33]. Given the rising concern about AI's environmental impact, there is a pressing need for a new cluster resource management strategy to mitigate the environmental footprint of ML processing, especially in the pursuit of global carbon neutrality [1, 26].

Existing cluster resource schedulers for ML jobs primarily focus on optimizing metrics related to job completion time (JCT) for time efficiency. However, their optimization practices may result in inefficient energy usage [39]. For example, schedulers like Pollux [20], Optimus [19], and Themis [14] optimize throughput by dynamically adjusting GPU allocation and job hyperparameters but neglect variations in jobs' power consumption during scaling and reconfiguration. Zeus [39] balances the tradeoff between energy consumption and time efficiency, but it lacks consideration for resource allocation among multiple jobs with different energy characteristics.

More importantly, energy usage is *not* the only factor that contributes to the carbon footprint. Carbon intensity, representing carbon emissions per unit of energy consumed, can vary significantly over time [18] and considering this temporal variation is crucial in reducing the carbon footprint of ML clusters [34]. ML training jobs often span days or weeks, during which they are intermittently paused and resumed based on their priority [33]. Therefore, these jobs have a level of temporal flexibility – the timing of job execution can be shifted as long as the total allocated resources are preserved within a specified time frame (e.g., 24 hours). By exploiting this flexibility, schedulers can align job power consumption with carbon intensity fluctuations, effectively reducing the overall carbon footprint. However, existing energy-aware optimizations [23, 39] overlooked carbon intensity and job flexibility's combined effect on cluster-wide environmental impact.

This paper presents GREEN, a novel carbon-efficient cluster scheduler that aims to reduce the overall carbon footprint of the cluster while achieving comparable time efficiency performance to prior work. GREEN utilizes a carbon-aware scheduling algorithm that balances the cluster-wide JCT and carbon footprint by considering various carbon-related factors, including the *energy efficiency* of each job, which derives from how effectively energy usage contributes to a job's training progress. Additionally, the algorithms exploit the temporal flexibility of ML jobs to reduce grid carbon emissions by automatically shifting workloads to greener hours with lower carbon intensity while still maintaining overall daily capacity.

GREEN makes carbon-aware decisions on scheduling by taking into account carbon-related factors. Specifically, GREEN employs a carbon tracker (§4) to monitor the energy consumption and other runtime metrics of ML jobs throughout their execution, and uses a factor model to compute the *energy efficiency* (the change in energy usage as the job progresses) for each job and *carbon footprint* (the accumulated carbon emissions based on energy usage).

GREEN proposes two optimizers (§5) for optimizing energy efficiency and reducing carbon footprint in ML clusters. The *Energy Efficiency Optimizer* scales resources allocated to jobs, taking into account the scalability demonstrated by the amount of energy consumption for a job when scaling, while the *Carbon Footprint Optimizer* dynamically rearranges job priorities for peak load shifting, minimizing cluster-wide carbon emissions while ensuring fairness in resource allocation.

To enable the co-optimization of these two optimizers, GREEN's scheduling algorithm (§6) incorporates a Multi-

level Feedback Queue (MLFQ) mechanism. The upper queue, guided by the *Energy Efficiency Optimizer*, emphasizes energy efficiency by scaling out jobs that achieve greater progress with lower energy consumption. The lower queue retains the progress and decisions made in the upper queue and utilizes the *Carbon Footprint Optimizer* to minimize the carbon footprint by relocating high-power-consuming jobs to "greener" hours with lower carbon intensity.

GREEN optimizes cluster-wide carbon footprint but has a non-goal: GREEN avoids altering job-level configurations (e.g. model hyperparameters) as in some existing ML schedulers [20, 39]. This design choice makes GREEN non-intrusive to the user's ML implementation and becomes orthogonal to optimization techniques that are provided in ML frameworks at the single-job level, such as adaptive hyperparameters and learning rate decay, which would otherwise conflict with job-level optimizations in cluster schedulers.

We evaluate GREEN using two types of workloads: (1) a production ML cluster used by a research institution with over 500 active academic users, and (2) a real-world machine learning workload trace from Alibaba [33]. We compare the performance of GREEN to state-of-the-art cluster schedulers under these workloads, and report on time metrics including JCT and makespan, as well as energy metrics including carbon footprint and peak power usage. Our results show that GREEN achieves significant reductions in the cluster's carbon footprint (up to 41.2%) and peak power draw (up to 12%), while maintaining comparable time efficiency to prior work, with a tradeoff of 3.6%-5.9% in average JCT.

We summarize our contributions as follows:

- We present a factor model to calculate carbon-related metrics and factors, including Carbon Intensity Curve (CIC), carbon footprint, and energy efficiency, providing important signals to the scheduler.
- We propose a carbon-efficient scheduler that scales out jobs while ensuring energy efficiency, and moves high-power jobs to greener hours to reduce carbon footprint without compromising fairness in resource allocation.
- We implement and evaluate GREEN with the real workload in a 32-node production cluster and the ML job trace from a large-scale enterprise cloud, and results show that GREEN can achieve up to 41.2% carbon footprint reduction, by trading <5.9% in average JCT.

2 Background and Motivations

This section discusses related work on ML cluster schedulers and energy-aware systems under a wider context.

2.1 ML Cluster Scheduling

A machine learning (ML) cluster is a shared environment where multiple users can submit training jobs that compete for limited resources. The cluster scheduler is responsible for managing job queues and allocating resources, with the goal of optimizing performance metrics such as job completion time (JCT) and resource utilization. This setup is commonly referred to as Machine Learning-as-a-Service (MLaaS) [33].

Schedulers	Scale-Adapt.	Energy-Aware	Carbon-Aware	Model-Agnostic
Gandiva / AntMan	✗	✗	✗	✗
Tiresias	✗	✗	✗	✓
Optimus	✓	✗	✗	✓
Pollux	✓	✗	✗	✗
Zeus	✗	✓	✗	✗
GREEN	✓	✓	✓	✓

Table 1: Characteristics of representative cluster schedulers.

Studies in ML job clusters explore optimization techniques in different directions. We classify these works based on key characteristics and present representative ones in Table 1.

① **Scale-Adaptive (or Elastic)**. Scale-adaptive schedulers dynamically adjust resource allocation (i.e., number of GPUs) for jobs based on their efficiency in speeding up tasks. Pollux [20] assesses statistical efficiency in ML training, measuring progress per unit of data processed. Optimus [19] builds predictive models for job-specific system throughput to determine scalability.

In the energy-aware context, jobs have different energy characteristics when scaling out, therefore, varying scheduling decisions can lead to different outcomes in energy consumption, affecting overall power consumption. This renders some scheduling decisions less favorable than others in the energy context. For instance, Pollux may prioritize scaling out jobs with lower speedup potential, preventing stragglers and enhancing average JCT. However, this approach directs more resources to jobs with lower energy utilization.

Non-scale-adaptive schedulers overlook the scalability of ML job performance under allocated resources. Tiresias [7] and Gandiva [35], for instance, require users to set a fixed number of GPUs throughout execution, limiting cluster-side performance optimization possibilities.

② **Model-Agnostic**. Some schedulers control model hyperparameters or require prior knowledge of a job (e.g., a specific pattern). Pollux [20] dynamically adjusts DNN hyperparameters (e.g., learning rate, per-GPU batch size) for improved statistical efficiency. Gandiva [35] uses a time-slicing approach for a group of jobs running the same DNN model. AntMan [36] uses a specialized framework for fine-grained GPU-sharing. A recent work, GreenFlow [6], dynamically allocates GPUs and optimizes job-level configurations to reduce average JCT under a carbon emission budget.

However, modifying job configurations introduces compatibility and usability challenges. Key hyperparameters, such

as per-GPU batch size and learning rate, are highly sensitive in ML workloads and exhibits intricate behaviors, especially in modern model architecture such as transformer-based models. Enforcing control over these hyperparameters not only requires additional engineering overhead on users to integrate with external schedulers but also conflicts with model-level optimizations in ML frameworks. Similar concerns have been highlighted in recent scheduler research [11].

③ **Energy-Aware.** GPU power consumption, which varies with utilization, is a key driver of ML models' overall energy usage. For instance, an NVIDIA H100 GPU consumes between 150W at low utilization and over 700W at peak load. Optimizing job energy efficiency thus requires accounting for GPU utilization. Zeus [39] optimizes the trade-off between energy consumption and time efficiency for individual jobs but does not address cluster-wide resource allocation across jobs with varying energy profiles. Meanwhile, prior research on energy-aware optimizations primarily targets generic workloads without leveraging ML-specific characteristics and requirements, as discussed in §2.2.

Design Space of GREEN. GREEN embodies all three attributes discussed above: scale-adaptive, model-agnostic, and energy-aware. Specifically, GREEN achieves energy and time efficiency comparable to prior work, without modifying job-level settings such as model hyperparameters. This approach preserves the non-intrusiveness of GREEN within the user's ML implementation and ensures GREEN's compatibility with both existing and future job-level optimization techniques in ML frameworks (as discussed in §9).

2.2 Carbon-aware Resource Scheduling

When addressing carbon-aware scheduling, it is important to recognize that reducing energy consumption may not be the sole goal. In optimizing for a lower carbon footprint, the focus shifts to carbon intensity, which measures emissions per kWh of power consumption.

Carbon Intensity ($kgCO_2/kWh$) signifies carbon emission per kilowatt-hour of electricity. It can be publicly obtained from public sources as a grid characteristic. Figure 1 illustrates hourly carbon intensity averages using data from UK [18], revealing daily variation.

The conceptual model below calculates carbon footprint.

$$\texttt{FOOTPRINT}_{\texttt{job}} = \int \mathit{Power}_{\mathit{host}} \times \mathit{Carbon\ Intensity}\ dTime$$

The model tracks device power and then integrates the product of power and carbon intensity over time. This integral is essential as both power and carbon intensity are time-varying.

An intuition is that during low carbon intensity time, more optimal scheduling decisions might be possible by moving high-power jobs to these lower periods. Certain ML training jobs exhibit flexibility for this time-shifting. This is because ML model training can span over weeks, or even months for large models, during which they are intermittently paused and resumed based on their scheduling priority [33]. The timing of job execution can be shifted as long as the total allocated resources are preserved within a specified time frame (e.g., 24 hours). By exploiting this flexibility, schedulers can align job power consumption with carbon intensity fluctuations, effectively reducing the overall carbon footprint.

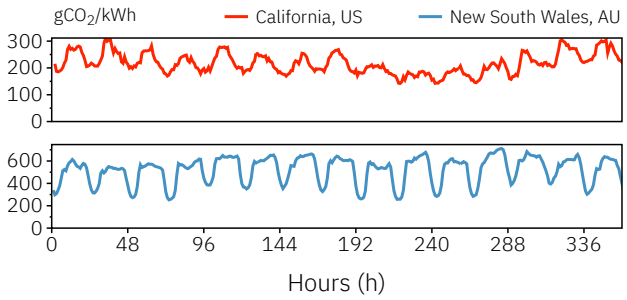


Figure 1: Daily pattern of historical carbon intensity, using data from August 2023 [16]. The carbon intensity of energy supplied by the electric grid depends on the energy sources.

Next, we briefly discuss existing work in carbon-aware scheduling, focusing on the limitations of these approaches in the ML cluster context.

Power and Resource Throttling. Google's CICS [23] imposes a limit on cluster capacity during periods of high carbon intensity. Some approaches [6, 39] utilize Dynamic Voltage and Frequency Scaling (DVFS) to enforce GPU power limits, which directly caps energy usage. However, these methods reduce energy consumption essentially by *scaling back the work performed*. In the MLaaS context, where ML clusters often operate at high occupancy, imposing power or resource limits reduce cluster capacity, leading to a negative impact on overall job completion time (JCT).

Lack of Capacity Constraints. EcoVisor [28] and WaitAWhile [34] use scheduling strategies that do not account for resource competition between jobs. CarbonScaler [8] requires a user-specified job deadline and can scale jobs using unlimited, on-demand cloud resources to meet the deadline. GAIA [9] focuses on cloud environments with on-demand resources, aiming to balance carbon footprint, time performance, and resource cost. In contrast, the MLaaS setting involves static cluster resources, leading to competition between jobs. In this context, the scheduler must make dynamic decisions on resource allocation while adhering to the cluster's resource constraints.

In summary, we observe that prior approaches to carbon-aware scheduling often overlook resource constraints, either throttling jobs during periods of high carbon intensity—leaving resources underutilized—or elastically scaling jobs with unconstrained on-demand capacity, disregarding resource limitations. Meanwhile, some systems require additional user input, such as predefined job deadlines [8] or

carbon budgets [6], which are often difficult for ML workloads due to their unpredictable execution characteristics.

To bridge this gap, we propose GREEN, a resource scheduler for ML jobs that operates strictly within available resource capacity. GREEN exploits characteristics of ML workloads to monitoring and reduce the cluster-wide carbon footprint while preserving time efficiency.

2.3 Motivation and Challenges

We consider the opportunities missed by existing job schedulers to reduce cluster-wide carbon footprint.

Motivation 1: Available resources are not allocated in a way that optimizes energy utilization. Prior ML cluster schedulers focuses on JCT optimization without addressing cluster-wide energy efficiency. As discussed in §2.2, these approaches are either energy-unaware [7, 14, 20] or emphasize at the single-job level [39]. Consequently, they do not effectively optimize for clusters where jobs' energy characteristics vary and resource allocation impacts cluster-wide energy consumption.

Opportunity: The cluster scheduler should determine the appropriate level of resources to allocate to a job, taking into account its scalability in both speed and energy consumption.

Motivation 2: Temporal flexibility of workload is not exploited to reduce carbon footprint. As ML models may take days or weeks to train, ML jobs are generally not sensitive to minor variations in progress during the process. This indicates that they have a certain degree of temporal flexibility. While we also observe the fact that carbon intensity (described in §2.2) varies substantially over time of day, we can exploit the temporal elasticity to consume energy when the electricity grid is less carbon-intensive.

Opportunity: The cluster scheduler should apply temporal shifting to flexible workloads, moving up or delaying them to "greener" hours with lower carbon intensity.

Challenges: The key challenge lies in making carbon-aware scheduling decisions without introducing significant performance degradation. To achieve this, the scheduler must:

- Preserve cluster efficiency: New scheduling and scaling mechanisms must maintain overall cluster efficiency, ensuring job execution speed remains comparable to state-of-the-art schedulers.
- Ensure resource fairness in temporal shifting: When shifting flexible workloads temporally, the total resource allocation over a given period (e.g., 24 hours) must be preserved to prevent job starvation.

3 GREEN Overview

GREEN is a GPU cluster scheduler designed to minimize the carbon footprint of ML job execution, while preserving time efficiency on par with existing ML job schedulers.

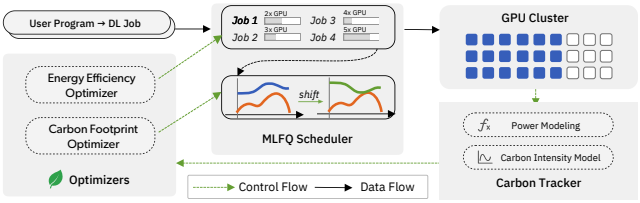


Figure 2: An overview of GREEN's system components and workflow

Workflow. As depicted in Figure 2, GREEN processes user-submitted jobs and allocates resources across them. Being model-agnostic, GREEN makes no assumptions of user implementation under specific frameworks. Similar to existing ML cluster schedulers [7, 20, 29] in the MLaaS setting [33], GREEN optimizes cluster-wide metrics without providing direct SLO or deadline guarantees for individual jobs.

Carbon Tracker (§4). Carbon tracker is the essential component for profiling. It makes GREEN carbon-aware by monitoring factors including per-job energy consumption throughout ML job execution as well as estimating the carbon footprint using the carbon-intensity model. These factors serve as important inputs for GREEN's carbon-related optimizers and scheduling algorithm described below.

Optimizers (§5). We propose two optimizers employed in GREEN: (1) the Energy Efficiency Optimizer, which allocates resources to jobs in a way that maximizes energy efficiency in the cluster, taking into account per-GPU energy-to-progress efficiency, and (2) the Carbon Efficiency Optimizer, which minimizes the cluster-wide carbon footprint by dynamically adjusting job priorities for peak load shifting.

MLFQ Scheduler (§6). The scheduling algorithm utilized by GREEN resembles a Multilevel Feedback Queue (MLFQ), allowing for different scheduling strategies based on varying job states. Two scheduling queues are managed by the aforementioned optimizers, in a synergistic manner: the upper queue emphasizes energy efficiency by scaling out jobs that achieve greater progress with lower energy consumption, while the lower queue maintains the scaling decision from the upper queue and minimizes the carbon footprint by shifting high-power-consuming jobs to greener hours with lower carbon intensity.

4 Carbon Tracker

GREEN is carbon-aware as it monitors per-job energy consumption throughout the execution of an ML job, and estimates carbon footprint using a carbon-intensity model.

4.1 Modeling Power Draw

The job's power draw P_{job} at time t can be estimated with:

$$P_{\texttt{job}}(t) = P_{gpu}(t) + P_{cpu_model}(t) + P_{static}$$

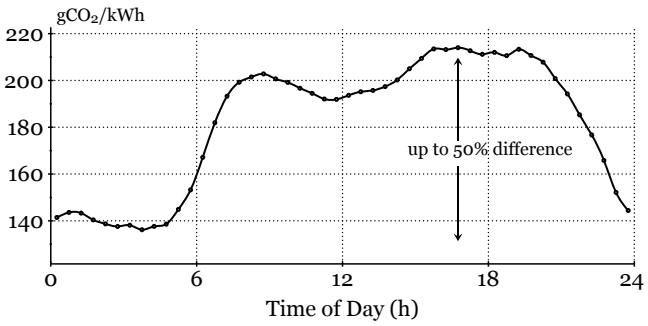


Figure 3: Carbon Intensity Curve (CIC): plotted with average carbon intensity values for each hour of the day in a 3-month period [18]. Y-axis is not zero-based.

We use NVML (NVIDIA Management Library) to read the power draw of GPUs $P_{\texttt{gpu}}$ in real-time. For CPU power consumption, we employ the model [22] to estimate power consumption $P_{\texttt{cpu}}$ in kW from CPU usage. Additionally, to factor in other devices like DRAM and motherboard, which are less influenced by their load, we use a static value $P_{\texttt{static}}$ to represent their power draw.

For the actual calculation, we discretized the equation into a step function with 30-second intervals to minimize measurement overhead. Notably, NVML power readings may not always be accurate [38], but our method remains adaptable to any measurement source, including external power meters for direct hardware power assessment.

4.2 Estimating Carbon Footprint

Carbon Footprint refers to the cumulative amount of carbon emissions generated since the start of a job. These emissions are measured in units of kilograms of carbon dioxide ($kgCO_2$).

Carbon Intensity. We introduce the Carbon Intensity Curve (CIC) as our carbon intensity model (Figure 3).

$$\texttt{FOOTPRINT}_{\texttt{job}}(T) = \int_0^T P_{\texttt{job}}(t) \cdot CIC(t)dt$$

The equation above defines the carbon footprint. By calculating the integral of $P_{job} \cdot CIC$ over time t, the result reflects both power consumption and carbon intensity over time.

Carbon intensity data is publicly accessible online in many regions [16, 31]. In our implementation, we make use of historical carbon intensity data, as it usually exhibits consistent daily pattern over a period of time. GREEN's carbon footprint optimizer can adapt to any pattern of CIC (§5.2), and recent work [28] proposes a software-based energy system to retrieve energy usage and carbon intensity in real-time.

5 GREEN Optimizers

This section introduces two optimizers in GREEN's scheduling algorithm: Energy Efficiency Optimizer (§5.1) allocates resources to jobs to maximize energy efficiency, taking into account the scalability of per-GPU energy-to-progress efficiency. Carbon Footprint Optimizer (§5.2) reduces the cluster-wide carbon footprint by dynamically adjusting job priorities to shift peak loads to greener hours.

5.1 Energy Efficiency Optimizer

The objective of GREEN's Energy Efficiency Optimizer is to determine the appropriate level of resources to allocate to a given job, maximizing energy efficiency in the cluster. This objective is motivated by the observation that a job's scalability measured in energy efficiency may differ from its performance scalability (e.g., training throughput), leading to suboptimal cluster-wide energy efficiency.

Definition. To quantify energy efficiency, we formulate it as the derivative of job progress $Progress_{\texttt{job}}$ with respect to total accumulated energy usage $\texttt{W}_{\texttt{job}}$, as shown in Equation 1.

$$\texttt{EFFICIENCY}_{\texttt{job}}(t) = \frac{\mathrm{d}Progress_{\texttt{job}}(t)}{\mathrm{d}\texttt{W}_{\texttt{job}}(t)} \tag{1}$$

This formulation enables the scheduling algorithm to leverage two key pieces of information:

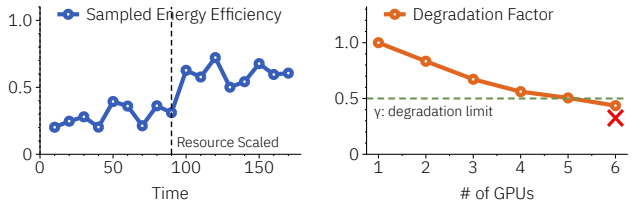
- Jobs with higher energy efficiency are able to make more progress with the same power consumption when compared to jobs with lower energy efficiency.
- By allocating more computing resources to a job, GREEN can observe changes in energy efficiency and learn about the job's scalability. Ideally, there would be no decrease in energy efficiency when scaling out a job.

The unit of energy efficiency is *not* important, as long as it can be considered as a unit of work done over a unit of power consumption. This is because our focus is on observing its relative change ratio during the job's execution, and the unit cancels out in the calculation.

Calculation of Energy Efficiency. In Equation 1, energy efficiency is defined as a function of time, and to obtain this function for an actual job run, GREEN needs to approximate it using runtime data. There are two built-in methods for GREEN to measure the progress of a job, depending on whether the progress of a job can be directly obtained during its runtime.

If job progress is available, GREEN calculates the efficiency function by sampling progress for a duration of time as defined in Equation 2.

$$\texttt{EFFICIENCY}_{\texttt{job}}(t) = \frac{Progress_{\texttt{job}}(t) - Progress_{\texttt{job}}(t-\Delta t)}{\texttt{W}_{\texttt{job}}(t) - \texttt{W}_{\texttt{job}}(t-\Delta t)} \tag{2}$$



(a) Energy Efficiency **(b)** Degradation Factor

Figure 4: An example of energy efficiency calculation. *Left*: Energy efficiency data sampled before and after resource scaling, with the unit being % of progress per kWh. *Right*: Degradation factor over the number of GPUs, obtained by averaging efficiency data per GPU.

The variable Δt represents the time elapsed since the most recent job scaling event. GREEN tracks job progress by recording the number of completed epochs and finding the ratio of completed epochs to the total epochs. This method works well because ML training repeats in rounds and each training epoch takes roughly the same amount of time.

If job progress cannot be directly obtained, such as when training large models with long epoch time, GREEN can approximate progress using the job's training throughput $Throughput_{\texttt{job}}$, i.e., the number of data samples trained over a duration of time, as shown in Equation 3.

$$\texttt{EFFICIENCY}_{\texttt{job}}(t) = \frac{\Sigma_{t'=t}^{t-\Delta t} Throughput_{\texttt{job}}(t')}{\texttt{W}_{\texttt{job}}(t) - \texttt{W}_{\texttt{job}}(t-\Delta t)} \quad (3)$$

When progress measurements are not available, the formula represents an absolute value of the progress made (i.e., throughput) during that duration of time.

Figure 4a shows an example of the energy efficiency collected before and after one scaling event.

We make note that retrieving job throughput is a common practice in today's cluster schedulers [20, 39], in practice, it often just requires a simple log print by the job.

Job Scaling with Energy Efficiency. Dynamic job scaling plays a crucial role in enhancing the performance of ML cluster JCT [20]. GREEN employs the energy efficiency metric, as defined earlier, to make scaling decisions.

For each job that is prepared to run, it commences with one GPU (or the minimum number of GPUs necessary for the job). We introduce an efficiency degradation factor, which represents the proportion of per-GPU energy efficiency decline relative to the initial job energy efficiency:

$$\texttt{DEGRADATION}_{\texttt{job}}(\texttt{t}) = \frac{\texttt{EFFICIENCY}_{\texttt{job}}(\texttt{t})}{\texttt{EFFICIENCY}_{\texttt{job}}(\texttt{t}_0)} \quad (4)$$

where $\texttt{EFFICIENCY_job}(\texttt{t}_0)$ represents the job's energy efficiency with the minimum GPU required, which is profiled at the beginning time (t_0) of the job's execution.

In subsequent scheduling rounds, the job will receive an additional GPU if its efficiency degradation factor remains above a threshold γ. For instance, with γ at 0.9, it means that the job's energy efficiency must not decrease by more than 10% to be eligible for scaling out. The value of γ is dynamically adjusted according to both the cluster-wide occupancy rate and job characteristics to prevent over- or under-allocation of resources. Additionally, A maximum resource cap prevents well-scaled jobs from taking up all GPU resources. More details about GREEN's adaptive strategy can be found in §6.4.

Figure 4b demonstrates an example of the change in energy efficiency during scaling. The unit of energy efficiency cancels out, as we only use its relative change in percentage as a scheduling factor during job execution. This scaling approach reduces the cluster's carbon footprint by allocating more resources to jobs that efficiently utilize energy for progress.

Relation with Single-job Optimizations. The design of the Energy Efficiency Optimizer is orthogonal to previous optimizations on the job-level. GREEN optimizes the energy efficiency with resource allocation strategies, while earlier work such as [39] takes resource allocation as an input, and investigates job-level parallelization and configurations to optimize efficiency. These optimizations can be combined and utilized together to enhance overall energy efficiency.

5.2 Carbon Footprint Optimizer

The Carbon Footprint Optimizer in GREEN aims to reduce the cluster-wide carbon footprint through peak load shifting. This is achieved by dynamically adjusting job priorities in scheduling.

The job priority is calculated as a combination of two factors, namely the Carbon Factor and the Shifting Factor, with higher priority values indicating lower scheduling priorities.

$$\texttt{PRIORITY}_{\texttt{job}} = \left(\frac{\texttt{FOOTPRINT}_{\texttt{job}}}{\texttt{DEGRADATION}_{\texttt{job}}}\right) \cdot \texttt{SHIFTING}_{\texttt{job}} \quad (5)$$

An input t, denoting the system time, exists for all factors defined above and is omitted for readability.

Carbon Factor. The carbon factor is calculated by dividing a job's accumulated carbon footprint, denoted as $\texttt{FOOTPRINT}_{job}$ (§4), by its energy efficiency degradation factor represented by $\texttt{DEGRADATION}_{job}$ (Equation 4).

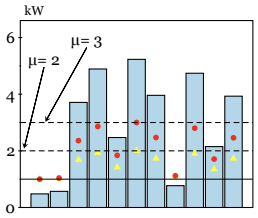
The Carbon Factor serves as the foundation for job priority and exhibits two essential properties:

(1) It tracks the carbon emissions produced during a job's execution, similar to the Least-Attained-Service (LAS) algorithm, while taking environmental impact into account. This factor helps achieve fair resource allocation among all jobs.

(2) It penalizes jobs with poor energy efficiency by dividing their carbon footprint by the degradation factor. This method enables energy-efficient jobs that are scaled out by the Energy Efficiency optimizer to complete earlier, thereby enhancing overall JCT. It also helps to avoid the unnecessary occupation of the cluster's computing resources.

$\texttt{SHIFTING}_{\texttt{job}}$	if $P_{job} \leq \widetilde{P}$	if $P_{job} > \widetilde{P}$
if $CIC_t \leq \bar{I}$	$P^*_{\texttt{job}}$	$1/P^*_{\texttt{job}}$
if $CIC_t > \bar{I}$	$1/P^*_{\texttt{job}}$	$P^*_{\texttt{job}}$

(a) Function for Shifting Factor



(b) $P^*_{\texttt{job}}$ Example

Figure 5: Left: Piecewise function for calculating Shifting Factor. Right: $P_{\texttt{job}}$ (bar) and $P^*_{\texttt{job}}$ when $\mu = 2$ (yellow dots) or $\mu = 3$ (red dots).

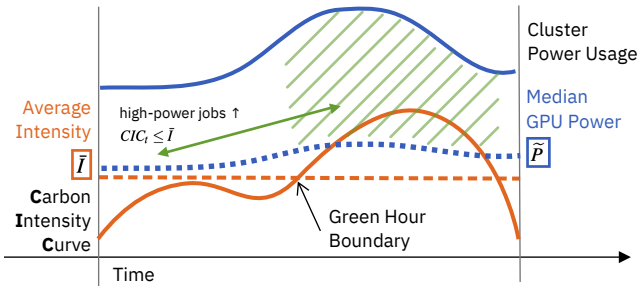


Figure 6: High-power jobs have higher priority when carbon intensity is below $\bar{I}$, and are assigned a small shift factor according to Table 5a to be scheduled earlier. The curve shown in figure is pre-shifting.

Shifting Factor. Temporal job shifting follows a heuristic that favors delaying high-power consumption jobs to "greener" hours with low carbon intensity and prioritizing low-power-consumption jobs during high-carbon-intensity periods.

The core idea behind the shift factor calculation is to adjust job priorities based on the current carbon intensity level relative to the daily values and the job's power consumption level relative to all unfinished jobs.

A piecewise function is employed, as outlined in Figure 5a, to achieve this using $\widetilde{P}$ and $\bar{I}$ as boundary values, dividing the shifting factor calculation into four quadrants:

1) *Average carbon intensity*, denoted by $\bar{I}$, is calculated using historical carbon intensity data, the hours with carbon intensity lower than $\bar{I}$ are considered to be *greener hours*.

2) *Median GPU power consumption*, denoted by $\widetilde{P}$, is obtained by finding the median GPU power draw over all profiled or running jobs. The jobs with GPU power draw higher than $\widetilde{P}$ are considered to be *high-power jobs*.

The piecewise function for calculating shifting factors rescales the job's power consumption $P_{\texttt{job}}$ to $[1,\mu]$, denoted as $P^*_{\texttt{job}}$, as defined below:

$$P^*_{\texttt{job}} = \frac{(P_{\texttt{job}} - P_{min})}{(P_{max} - P_{min})} \times (\mu - 1) + 1 \qquad (6)$$

The parameter μ (≥ 1) controls the scale of $P^*_{\texttt{job}}$ (example shown in Figure 5b) and hence the degree of temporal shifting, with larger values causing more job shifting. When $\mu = 1$, temporal shifting is essentially disabled, as the shifting factor is always scaled to 1. Evaluations show that $\mu = 2$ effectively balances the two factors to achieve both time and carbon efficiency (see sensitivity experiments in §A.2).

Figure 6 depicts an example usage of the piecewise function: when the current carbon intensity is lower than $\bar{I}$, the high-power jobs are assigned with a small shifting factor according to Figure 5a, resulting in higher job priority.

The optimizer can adapt to any pattern of carbon intensity changes, as it dynamically calculates the boundary values $\bar{I}$ and $\widetilde{P}$ based on given CIC and job power consumption.

Impact on Cluster-wide and Individual JCT. Deferring high-power tasks does not necessarily lead to a delay in the cluster-wide job completion time (JCT). This is because the scheduler swaps the daily computation resources assigned to high-power and low-power jobs in the temporal dimension but preserves daily capacity for a job. Analysis in §6.4 detailed the discussion about how GREEN avoids starvation for both long and short jobs. Meanwhile, our evaluation in §8.2.2 also provides empirical evidence to support the statement.

6 Putting It Together: MLFQ Scheduling

GREEN's scheduling algorithm resembles a Multilevel Feedback Queue (MLFQ), which employs different scheduling strategies based on job states (Figure 7).

6.1 Upper Queue: Online Profiling and Scaling

The upper queue utilizes the Energy Efficiency (EE) Optimizer (§5.1) to determine the scale of computing resources allocated to each job, ensuring that jobs with higher energy efficiency receive a greater share of energy. The logic is shown in Algorithm 1 lines 3-11.

Upon submission to the cluster, new jobs are placed at the end of the upper queue and profiled for potential scaling.

When a job is initially submitted for execution, it starts with a single GPU (or the minimum number of GPUs necessary for the job). Repeatedly for each scheduling round, GREEN iterates through the queue in the first-come-first-serve (FCFS) order, and evaluates the energy efficiency degradation factor to determine a job's eligibility for further scaling (lines 4-7). When there are multiple jobs in this queue, the additional unit of computing resources is allocated to them in FCFS order until the cluster reaches its maximum capacity (lines 7-8).

Queue Transition. The job will not leave the upper queue as long as it is still eligible for scaling. Jobs transition to the lower queue once the EE optimizer has completed the scaling process. Transitioning between queues does not necessarily interrupt the job's execution, as it may remain eligible to run based on the priority assigned in the lower queue.

6.2 Lower Queue: Peak Load Shifting

The lower queue employs the Carbon Footprint (CF) Optimizer (§5.2) to conduct peak load shifting by relocating

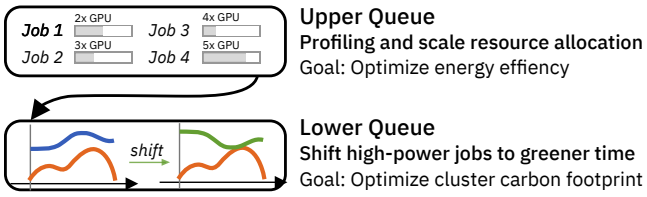


Figure 7: MLFQ enables two optimizers to work together. The upper queue determines the job's spatial placement (number of GPUs), while the lower queue decides the temporal aspect (time to run) to achieve different optimization goals.

energy-intensive jobs to greener hours with lower carbon intensity, thus reducing the cluster-wide carbon footprint. The core scheduling logic is shown in Algorithm 1 lines 12-17.

The lower queue determines job priorities using the CF optimizer, while inheriting resource scaling decisions from the upper queue. Partial execution from online profiling, is considered part of the running time, and the work completed during this phase is retained as part of the job progress.

In the lower queue, GREEN calculates each job's priority based on two factors (§5.2): the carbon factor, which considers job energy consumption while ensuring fairness in resource allocation, and the shifting factor, which moves high-energy-consuming jobs to hours with lower carbon intensity. GREEN sorts jobs by their priorities and schedules them for execution in descending order.

6.3 Scheduler Workflow

The scheduling workflow consists of both active and passive components.

Active scheduling, shown in Algorithm 1, operates in rounds to assign job priorities and preempt lower-priority jobs. It ranks jobs for immediate execution at the top (triggering preemption), followed by jobs in the FCFS upper queue, and then those in the lower queue ordered by the priorities calculated (line 12).

To limit excessive preemption, the interval between active scheduling, known as the time quantum, is set to 30 minutes. Our findings in §A.2 show that this interval length does not significantly influence overall outcomes, as long as it is not excessively short.

Passive scheduling comes into effect in the intervals between active scheduling rounds. During these periods, when a job finishes and releases resources, the scheduler deploys the passive strategy to avoid idling. This passive strategy starts the next job based on its rank in the latest active scheduling without recalculating priorities or triggering preemptions.

Job SLO and Preemptions. Similar to existing cluster schedulers [7, 20, 29] used in MLaaS environments [33], GREEN prioritizes global cluster efficiency rather than providing strict Service Level Objectives (SLOs) or deadline guarantees for individual jobs. GREEN assumes all jobs are preemptible, which may not align with workloads requiring strict

Algorithm 1: Carbon-efficient MLFQ Scheduling

Input:
- Jobs in Upper and Lower Queue: $\mathcal{J}_\mathcal{U}, \mathcal{J}_\mathcal{L} \in \mathcal{J}$
- Cluster Capacity and Upper Queue Cap: $\mathcal{C}$ and θ
- Current Timestamp: t

1 **begin**
2 Allocation $\mathcal{A} = \{\}$
3 **for** *Job* $J \in \mathcal{J}_\mathcal{U}$ *ordered by* $\texttt{DEGRADATION}_J$ **do**
4 **if** $\texttt{DEGRADATION}_J = \Delta\texttt{EFFICIENCY}_J \geq \gamma$ **then**
5 Scale out J with one more GPU
6 $\mathcal{A} = \mathcal{A} \cup J$
7 **if** $Capacity(\mathcal{A}) > \theta \times \mathcal{C}$ **then**
8 break
9 **else**
10 Remove J from $\mathcal{J}_\mathcal{U}$ and add to $\mathcal{J}_\mathcal{L}$
11 **end**
12 $\texttt{PRIO}_\texttt{J} = (\frac{\texttt{FOOTPRINT}_\texttt{J}}{\texttt{DEGRADATION}_\texttt{J}}) \cdot \texttt{SHIFTING}_\texttt{J}\ \forall J \in \mathcal{J}_\mathcal{L}$
13 **for** *Job* $J \in \mathcal{J}_\mathcal{L}$ *ordered by* $\texttt{PRIORITY}_J$ **do**
14 $\mathcal{A} = \mathcal{A} \cup J$
15 **if** $Capacity(\mathcal{A}) > \mathcal{C}$ **then**
16 break
17 **end**
18 Signal jobs $\forall J \in \mathcal{J}/\mathcal{A}$ for preemption
19 Allocate GPUs for $\forall J \in \mathcal{A}$
20 **end**

execution guarantees. In real-world deployments, GREEN's scheduling policy may be overridden to prevent preemptions for jobs with specific requirements.

6.4 Starvation Avoidance

Starvation Avoidance for Short Jobs. In the upper queue, where job scaling occurs, as a job progresses and could potentially be complete, it implicitly favors jobs with lower GPU consumption. This approximates the short-job-first (SJF) strategy, improving average JCT and avoiding starvation of shorter jobs. The scaling process in the upper queue also eliminates the need for a separate profiling stage. In the lower queue, the carbon footprint-based factor has an LAS-like behavior, which deprioritizes and preempts longer-running jobs over time, making room for shorter jobs. In §8.2.2, we quantitively evaluate the impact on cluster-wide and individual JCTs.

Starvation Avoidance for Long Jobs. To prevent starvation of longer jobs during peaks of new job submissions (when new jobs enter the upper queue and gain priority over existing long jobs in the lower queue.), a knob for the capacity cap is enforced in the upper queue. Default of 30%, this cap limits the total cluster capacity available to jobs in the upper queue. Upon reaching this cap, the scheduler prioritizes the jobs by their degradation factors, skips the remaining upper queue and considers the lower queue (Line 3-11 in Algorithm 1).

7 Implementation

We implement GREEN with Slurm [25]. Slurm is an open-source job cluster management system widely used to manage computing clusters. See Appendix B for our open-source plan.

Scheduler Implementation. GREEN's scheduler workflow is implemented as a Slurm scheduler plugin that supports custom policies to set job priorities in Slurm's job queue. The scheduler plugin communicates with an external daemon process, which stores CIC data, collects job energy statistics from profiler agents running on each compute node, and returns scheduling decisions to the scheduler plugin.

Energy-efficient Job Termination. We adopt the popular checkpoint-restart methods [20] for job scaling and preemption. However, the method may lose intermediate results since the last checkpoint, wasting computational resources. GREEN sends a `SIGTERM` signal with a 120-second grace period to ML jobs upon termination, allowing jobs to save a checkpoint after completing the current iteration and exit gracefully. The time for checkpointing and subsequent restart is recorded as part of the job's execution time.

8 Evaluation

We evaluated GREEN using time and energy metrics, comparing them against five baseline schedulers under a real-world ML workload and a trace-driven simulation workload.

8.1 Experimental Setup

Testbed. Our testbed cluster consists of 32 compute nodes. Each node is equipped with 2 Nvidia RTX 3090 GPUs, 20 CPU cores, and 64GB RAM, interconnected with the Mellanox ConnectX5 NIC. The nodes are managed by Docker, and all run on 64-bit Ubuntu 18.04 with CUDA v11.1.

Baseline Schedulers. We compare GREEN to 5 schedulers, representing the state-of-the-art in their respective categories, as introduced in §2. For schedulers that are not scale-adaptive, we preset each job's GPU allocation to the point where GREEN achieves the best energy efficiency.

1) *Tiresias* [7]. Tiresias adopts Least-Attained-Service (LAS) scheduling for ML clusters.

2) *GAIA's Carbon-Time* [9]. GAIA's Carbon-Time policy schedules jobs to start at times that minimize carbon emissions, allowing a maximum delay specified by parameter W. We set W to 12 hours, as suggested by the original paper.

3) *EcoVisor* [28]. EcoVisor enforces carbon-aware policies for jobs. We extend and implement its policy: jobs are queued in arrival order if the carbon intensity is higher than the threshold (set 10th percentile of the CIC in use).

4) *Pollux* [20]. Pollux is the state-of-the-art ML scheduler, jointly optimizing job hyperparameters and cluster resource allocation. Pollux is not model-agnostic and not energy-aware.

5) *Zeus* [39]. Zeus adjusts model hyperparameters and GPU power limits for individual jobs. We use Tiresias to allocate resources across jobs in our evaluation.

ML Workload. We collected 791 jobs from real users over a 24-hour period on a university-managed production cluster [32], containing self-contained job packages (code, data, configs). We replayed these jobs in our testbed following their original submission timeline. Our evaluation, with 791 jobs, is about 5 times larger than prior work (e.g., Pollux with 160 jobs [20]). The training duration of each job ranged from 1 minute to 37 hours (detailed in Table 2) and involved large ML models for tasks such as image classification (ResNet-50 [10], ShuffleNet [41]), speech recognition (DeepSpeech2 [2]), and natural language processing (BERT [5], GPT-2 [21]).

Pollux and Zeus are not model-agnostic and require job-specific configurations. Since user-submitted jobs may have incompatible implementations, we invested 100 man-hours to manually adjust them. Over 55% of the 791 jobs had unresolvable issues, such as unsupported learning rate scalers or gradient optimizers, while about 25% were excluded due to non-standard ML model implementations. Ultimately, we adjusted 150 jobs for non-model-agnostic evaluations (§8.2.3).

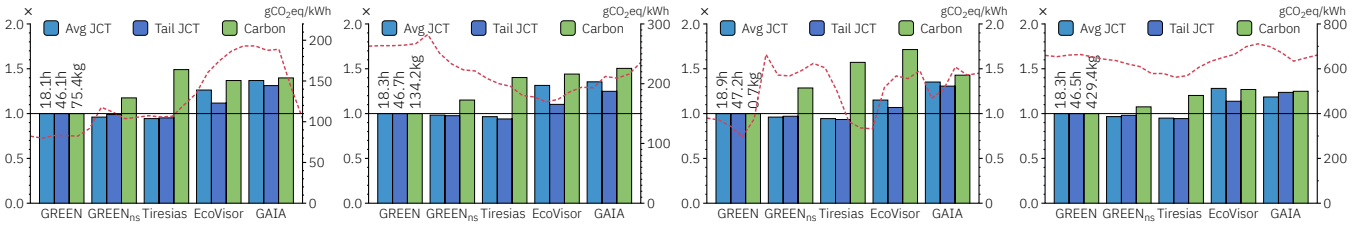
JCT Measurements. Typical ML cluster schedulers (as seen in baseline schedulers) operate in multi-tenant ML clusters where users submit jobs that run independently without deadline constraints. This workflow is also known as MLaaS [33]. The cluster scheduler allocates resources to optimize cluster-wide metrics like average JCT. In our testbed evaluation, we replayed jobs on their original timeline and calculated individual JCTs as the time from submission to completion — including queueing and preemption, with each job's running time determined by its user program.

Carbon Footprint Estimation. We tracked energy consumption to estimate emissions (details in §4), following a calculation method consistent with existing carbon-aware systems [8, 23, 28, 34]. In our evaluation, we used historical hourly carbon intensity data from Electricity Maps [16] for August 2023 across four regions: California, USA; the United Kingdom; Sweden; and Poland. These regions represent diverse combinations of carbon intensity and variability. Figure 8 presents the 24-hour carbon intensity patterns for each region in their respective subfigures with dotted red lines.

GREEN's scheduling algorithm (§5) dynamically computes boundary values for job shifting based on a given Carbon Intensity Curve (CIC). In our evaluation, we provide a 48-hour CIC to GREEN, representing typical carbon intensity patterns. For the year-long simulator experiments, a new 24-hour CIC is provided for every 24 hours to GREEN using consecutive historical data in 2023 to evaluate GREEN's adaptability to long-term, seasonal changes in carbon intensity.

8.2 Testbed Experiments

GREEN reduces the cluster-wide carbon footprint by up to 41.2%, with a trade-off of up to 3.6%-5.9% in average JCT and 5.1%-7.1% in tail JCT under a real ML workload when



(a) United Kingdom **(b)** California, USA **(c)** Sweden **(d)** Poland

Figure 8: Cluster-wide carbon footprint and JCT were evaluated across four regions with different carbon intensity profiles. The dotted red line shows the 24-hour carbon intensity pattern (right axis). The results are shown as ratios to GREEN (left axis).

compared to state-of-the-art cluster schedulers.

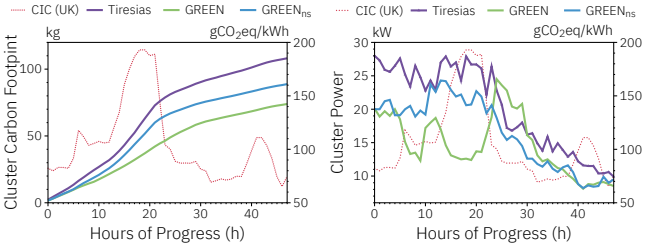
8.2.1 Cluster-wide Carbon Footprint

Figure 8 shows that GREEN reduces the cluster-wide carbon footprint by up to 41.2% (32.2% on average across 4 regions). When evaluating GREEN without job shifting over time (shown as GREEN_{ns}, meaning 'no shifting'), it achieves an average carbon reduction of 21% across the four regions.

Performance variations across regions. Regions with high variability in carbon intensity see more significant carbon footprint reductions due to job shifting during greener hours, such as the UK and Sweden (Figure 8a and 8c). As GREEN makes dynamic adjustments of the shifting threshold based on the carbon intensity profile, the average carbon intensity does not substantially affect the reduction percentage. However, we make a note that while the reduction percentage can be significant (over 30%), the absolute carbon footprint reduction may be minimal in areas like Sweden, where carbon intensity is comparatively low (less than 1% relative to other regions). In such cases, job shifting might introduce unnecessary overhead. These practical considerations are also noticed in recent literature [9, 30] and are further discussed in §9.

Carbon Footprint and Power Consumption. By monitoring energy efficiency during job scaling, GREEN avoids allocating additional resources to jobs with low energy efficiency. Figure 9a demonstrates the carbon footprint curve of GREEN, showing how the growth rate during high carbon intensity hours is lower than the other baselines. Figure 9b shows that GREEN reduces the cluster-wide power consumption. Under the UK CIC trace, GREEN achieves a 12% peak power reduction (from 28kW to 24.5kW) and a 25% decrease in total energy consumption (from 966kWh to 725kWh). This result shows another benefit of GREEN: physical clusters are typically designed and billed based on peak demand, and lower peak power usage reduces the required power capacity.

Comparison with Carbon-aware Baselines. GREEN outperforms GAIA and EcoVisor by 25.2% in average JCT across 4 regions. GAIA and EcoVisor delay job starts until the estimated carbon footprint is lower, often leaving resources idle during high carbon intensity periods. In contrast, GREEN dynamically calculates the job shifting threshold and prioritizes lower-power jobs during high-intensity periods, reducing



(a) Cluster-wide carbon footprint **(b)** Cluster-wide power draw

Figure 9: Cluster-wide carbon emission accumulation and power draw over the evaluation period (the same period shown in Figure 10). The dotted red line shows CIC on the right axis.

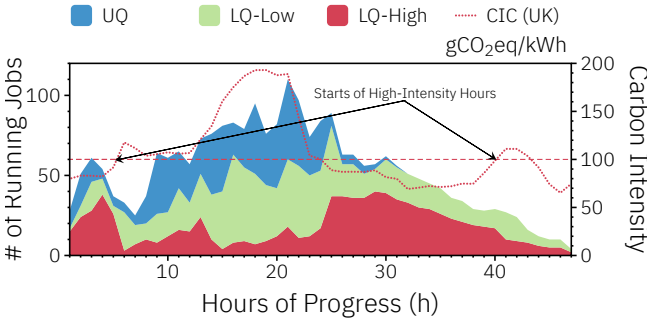


Figure 10: Job scheduling timeline showing the number of high- and lower-power jobs (left axis) responding to carbon intensity changes. UQ: Upper Queue; LQ-High/LQ-Low: Lower Queue's high/low-power jobs. The dotted red line represents the CIC and its boundary value (right axis).

power usage without leaving resources idle.

GAIA and EcoVisor exhibit higher carbon footprints because, in the multi-tenant ML cluster setting (targeted by GREEN, Tiresia, Pollux, and similar cluster schedulers), resource capacity is static: when jobs are delayed to a greener hour, idle resources still consume a baseline level of energy.

It is important to note that GAIA and EcoVisor are effective in their intended cloud environments, where resources are requested on demand by users, and idle resources do not incur additional costs or carbon emissions. Our evaluation extends their policies for multi-tenant ML cluster setting, where resource competition and constraints are factors.

Job Size (% of Total Jobs)	JCT Increase	
	Average	Tail
Extra Small Jobs (0-9 minutes, 22%)	−0.5%	−0.4%
Small Jobs (10-59 minutes, 30%)	1.7%	1.2%
Medium Jobs (1 - 10 hours, 30%)	4.4%	4.8%
Large Jobs (≥ 10 hours, 18%)	6.9%	5.8%

Table 2: GREEN's impact on individual JCTs compared to Tiresia across 4 job size categories.

8.2.2 JCT Performance

Figure 8 shows that GREEN maintains comparable JCT to baseline schedulers. Compared to the best-performing baseline, Tiresia, GREEN shows a 3.6%-5.9% increase in average JCT and a 5.1%-7.1% increase in tail JCT. With time-shifting disabled (GREEN_{ns}), the difference is smaller at 1.7%-1.8%, while still achieving a 21% reduction in carbon footprint.

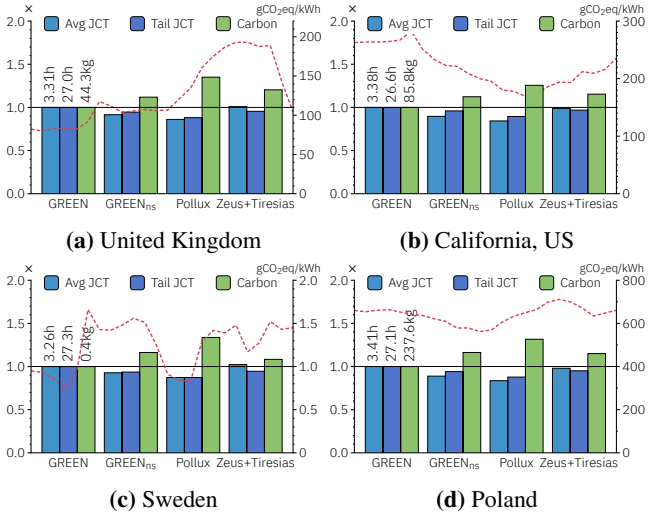
Cluster-wide Job Timeline. Figure 10 illustrates the job scheduling timeline in our testbed. During the transition between high and low carbon intensity hours, job priorities are swapped — lower-power jobs take precedence in high-intensity periods to reduce cluster power consumption. This adjustment is also reflected in Figure 9b, where power draws decrease during high carbon intensity hours. As shown in the figure, not all high-power jobs are stopped during high-intensity periods. They still run when (1) lower-power jobs do not fully utilize available resources or (2) high-power jobs have smaller carbon terms (§5) and gain higher priority.

Impact on Individual JCTs. Table 2 shows the evaluation of GREEN's impact on jobs' individual JCTs by comparing with Tiresia (the best JCT baseline) across 4 size categories based on total running time. GREEN had a maximum JCT increase of 6.9% for large jobs and a decrease of 0.5% for smaller jobs, demonstrating its effective use of MLFQ to prioritize shorter jobs while avoiding starvation of longer ones.

To understand the trade-off between overall JCT and carbon consumption, we conducted sensitivity analyses to evaluate how different extents of time shifting impact this trade-off. The results are provided in Appendix A for interested readers.

8.2.3 Non-Model-Agnostic Baselines

Figure 11 shows that GREEN reduces the carbon footprint by 23.9% on average across four regions compared to Pollux, with trade-offs of 17.2% in average JCT and 13.4% in tail JCT. Given that Pollux outperforms prior model-agnostic schedulers like Tiresia and Optimus by over 50% [20], this highlights GREEN 's ability to balance speed and carbon efficiency while remaining model-agnostic. GREEN also outperforms the combined Zeus (single-job energy optimization with GPU power limits) and Tiresia solution, achieving a 12.7% carbon footprint reduction with a similar average JCT.



(a) United Kingdom **(b)** California, US

(c) Sweden **(d)** Poland

Figure 11: Non-model-agnostic baselines evaluated with 150 compatible testbed jobs. Results are shown as ratios to GREEN (left axis), with CIC in dotted red lines (right axis).

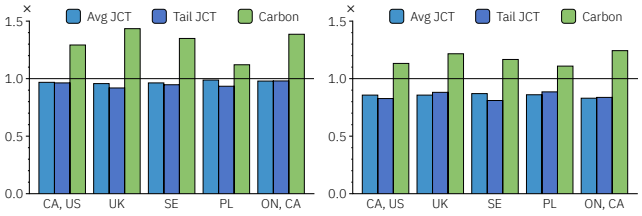
8.3 Trace-Driven Simulator Experiments

We conducted a year-long simulator experiment to evaluate GREEN's adaptability to seasonal changes in carbon intensity.

Simulator Construction. The simulator operates as a step-based discrete-time system with a 1-minute granularity, resulting in 518,400 steps for a full year. Each step performs three tasks: (1) calculate job progress and carbon footprint based on GPU time, progress factor (%/minute), current CIC, and power draw; (2) check the job trace for new job arrivals and adding them to the queue; (3) call the simulated scheduler to adjust jobs based on scheduling policies.

For workload simulation, we sample 400 job submissions from a 24-hour window in Alibaba's cluster workload trace [33] and repeat this process 365 times to construct a year-long trace with 146,000 jobs. The trace provides submission times and durations but lacks model-specific details. Following prior simulation methodologies in Pollux and Zeus, we categorize jobs by duration and map them to training jobs from our production workload (§8.1). To accurately model job execution, we pre-run jobs under varying configurations (e.g., GPU count, batch size) and record progress factors and power consumption. This enables the simulator to infer job progress and energy usage based on scheduling decisions. Additionally, we track checkpoint-resume overheads for each job category for preemptions to ensure simulation fidelity.

Performance Analysis. Figure 12 shows our simulation results. We evaluated GREEN across five regions, including Ontario, Canada, and compared it to the best-performing baselines, Tiresias and Pollux. The simulator showed improvement patterns across regions similar to our testbed results (showing better performance in high-variance locations). Compared to Tiresias, GREEN reduced the carbon footprint by 31.6%, with a 5.1% increase in average JCT and 7.5% in tail JCT.



(a) Tiresias **(b)** Pollux

Figure 12: Baselines evaluated in a year-long simulation across 5 regions. Results are shown as ratios to GREEN.

Job Size (% of Total Jobs)	JCT Increase	
	Average	Tail
Extra Small Jobs (0-9 minutes, 16%)	−0.1%	−0.3%
Small Jobs (10-59 minutes, 38%)	2.4%	4.7%
Medium Jobs (1 - 10 hours, 36%)	5.5%	8.0%
Large Jobs ($\geq$ 10 hours, 8%)	5.9%	6.1%

Table 3: GREEN's impact on individual JCTs compared to Tiresia in the simulator, breaking down by job sizes.

Compared to Pollux, GREEN achieved a 21.1% reduction in carbon footprint with a 15% increase in both average and tail JCT. We also report individual JCT impact in Table 3. Overall, the results show that GREEN adapts to seasonal carbon intensity variations to deliver carbon and time efficiency.

9 Applicability and Limitations

We discuss the applicability of GREEN's core designs.

Carbon Tracking and Forecasting. While GREEN effectively reduces carbon emissions through optimized GPU scheduling, several broader factors influence ML clusters' overall carbon footprint. Notably, *embodied carbon*—the emissions associated with hardware manufacturing, transportation, and lifecycle—is not considered in GREEN's model. Additionally, GREEN does not account for *data center Power Usage Effectiveness (PUE)*, which captures energy overheads from cooling, power distribution, and auxiliary infrastructure. We note that scheduling has a limited impact on carbon emissions from these factors. Recent work has shown that carbon-aware scheduling should not account for embodied carbon emissions, as they represent a sunk cost [4]. Similarly, PUE optimization is typically addressed at the data center design level and largely independent of workload scheduling.

GREEN dynamically adjusts workload execution based on carbon intensity forecasts, leveraging publicly available carbon data to estimate emissions. However, inaccuracies in carbon intensity predictions can lead to suboptimal scheduling decisions, and incorporating more accurate forecasting methods [15] could improve scheduling effectiveness. Furthermore, integrating real-time carbon intensity data directly from data centers presents an opportunity to enhance carbon-aware scheduling, which remains an open research challenge.

Scheduling Mechanism. GREEN's scheduling mechanism supports diverse ML task types and GPU heterogeneity, as its carbon tracking methods can generalize across various ML training regimes. However, GREEN is not well-suited for inference workloads, which are latency-sensitive as they respond to user requests in real time. The preemption-based scheduling mechanisms used by GREEN may introduce unexpected interruptions for inference tasks.

We also identified the following cases where GREEN's scheduling approach may have limited effectiveness:

- Special carbon intensity patterns: In regions with consistently low carbon intensity (e.g., Sweden), significant emission reductions are harder to achieve. Similarly, when carbon intensity remains stable, temporal job shifting may not be effective and could cause unnecessary scheduling overhead from frequent preemptions.
- Long job starvation: While GREEN includes starvation mitigation mechanisms, long-running jobs in highly congested clusters may still experience delays due to its LAS-based scheduling. Like Tiresias [7], a starvation threshold that resets job priority can be used to prevent starvation.

Integration with ML Frameworks. System research in ML is progressing quickly, with significant improvements in frameworks and optimization methods [12, 17, 27, 40]. Previous studies [33] show that GPU usage can vary widely across ML workloads depending on the optimization techniques used, which also affect power consumption. GREEN is designed to be *non-intrusive*, meaning it can schedule ML tasks without requiring users to change their code. Therefore, it is *orthogonal* to job-level optimizations, allowing it to work with both current and future ML system improvements.

10 Conclusion

GREEN utilizes a carbon-intelligent algorithm to schedule, scale, and shift workloads to optimize cluster-wide environmental and efficiency performance. We evaluate GREEN under real ML workloads in a production cluster and show that it can achieve significant carbon footprint reductions while maintaining time efficiency comparable to existing work.

Acknowledgments

We thank our shepherd, Noman Bashir, and the anonymous reviewers from NSDI 2025, as well as reviewers from our previous submissions, for their valuable feedback. GREEN was developed and evaluated on the TACC cluster [37] at the Hong Kong University of Science and Technology. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF 16213621, NSFC 62062005, 62402407. Han Tian and Kai Chen are the corresponding authors.

References

[1] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 118–132. ACM, 2023.

[2] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 173–182. JMLR.org, 2016.

[3] Victor Avelar, Patrick Donovan, Paul Lin, Wendy Torell, and Maria A Torres Arango. The AI disruption: Challenges and guidance for data center design. *Schneider Electric*, 2023.

[4] Noman Bashir, Varun Gohil, Anagha Belavadi Subramanya, Mohammad Shahrad, David E. Irwin, Elsa Olivetti, and Christina Delimitrou. The sunk carbon fallacy: Rethinking carbon footprint metrics for effective carbon-aware scheduling. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, November 20-22, 2024*, pages 542–551. ACM, 2024.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[6] Diandian Gu, Yihao Zhao, Peng Sun, Xin Jin, and Xuanzhe Liu. Greenflow: A carbon-efficient scheduler for deep learning workloads. *IEEE Trans. Parallel Distributed Syst.*, 36(2):168–184, 2025.

[7] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 485–500. USENIX Association, 2019.

[8] Walid A. Hanafy, Qianlin Liang, Noman Bashir, David E. Irwin, and Prashant J. Shenoy. Carbonscaler: Leveraging cloud workload elasticity for optimizing carbon-efficiency. In Michele Garetto, Andrea Marin, Florin Ciucu, Giulia Fanti, and Rhonda Righter, editors, *Abstracts of the 2024 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/PERFORMANCE 2024, Venice, Italy, June 10-14, 2024*, pages 49–50. ACM, 2024.

[9] Walid A. Hanafy, Qianlin Liang, Noman Bashir, Abel Souza, David E. Irwin, and Prashant J. Shenoy. Going green for less green: Optimizing the cost of reducing cloud carbon emissions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 479–496. ACM, 2024.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[11] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 457–472. ACM, 2023.

[12] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Sol: Fast distributed computation over slow networks. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 273–288. USENIX Association, 2020.

[13] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai

Chen. Understanding communication characteristics of distributed training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, page 1–8, New York, NY, USA, 2024. Association for Computing Machinery.

[14] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020.

[15] Diptyaroop Maji, Prashant J. Shenoy, and Ramesh K. Sitaraman. Carboncast: multi-day forecasting of grid carbon intensity. In *Proceedings of the 9th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, BuildSys 2022, Boston, Massachusetts, November 9-10, 2022*, pages 198–207. ACM, 2022.

[16] Electricity Map. https://www.electricitymap.org/map, Accessed August 2024.

[17] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 1–15. ACM, 2019.

[18] National Grid ESO, Environmental Defense Fund Europe, University of Oxford Department of Computer Science, and WWF. Carbon intensity API. https://carbonintensity.org.uk, Accessed August 2024.

[19] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 3:1–3:14. ACM, 2018.

[20] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021.

[21] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[22] Ana Radovanovic, Bokan Chen, Saurav Talukdar, Binz Roy, Alexandre Duarte, and Mahya Shahbazi. Power modeling for effective datacenter planning and compute management. *IEEE Transactions on Smart Grid*, 13(2):1611–1621, 2021.

[23] Ana Radovanovic, Ross Koningstein, Ian Schneider, Bokan Chen, Alexandre Duarte, Binz Roy, Diyue Xiao, Maya Haridasan, Patrick Hung, Nick Care, et al. Carbon-aware computing for datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, 2022.

[24] TACC Github Repo. https://github.com/turingaicloud/.

[25] SchedMD. https://slurm.schedmd.com/documentation.html, Accessed August 2024.

[26] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, 2020.

[27] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[28] Abel Souza, Noman Bashir, Jorge Murillo, Walid A. Hanafy, Qianlin Liang, David E. Irwin, and Prashant J. Shenoy. Ecovisor: A virtual energy system for carbon-efficient applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 252–265. ACM, 2023.

[29] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 642–657. ACM, 2023.

[30] Thanathorn Sukprasert, Abel Souza, Noman Bashir, David E. Irwin, and Prashant J. Shenoy. On the limitations of carbon-aware temporal and spatial workload shifting in the cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*, pages 924–941. ACM, 2024.

[31] WattTime. https://www.watttime.org/, Accessed August 2024.

[32] TACC Website. https://tacc.ust.hk/.

[33] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 945–960. USENIX Association, 2022.

[34] Philipp Wiesner, Ilja Behnke, Dominik Scheinert, Kain Kordian Gontarska, and Lauritz Thamsen. Let's wait awhile: how temporal workload shifting can reduce carbon emissions in the cloud. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, pages 260–272. ACM, 2021.

[35] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[36] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 533–548. USENIX Association, 2020.

[37] Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, Xinchen Wan, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. Design and operation of shared machine learning clusters on campus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 295–310. ACM, 2025.

[38] Zeyu Yang, Karel Adámek, and Wesley Armour. Part-time power measurements: nvidia-smi's lack of attention. *CoRR*, abs/2312.02741, 2023.

[39] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 119–139. USENIX Association, 2023.

[40] Chaoliang Zeng, Xudong Liao, Xiaodian Cheng, Han Tian, Xinchen Wan, Hao Wang, and Kai Chen. Accelerating neural recommendation training with embedding scheduling. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*. USENIX Association, 2024.

[41] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 6848–6856. Computer Vision Foundation / IEEE Computer Society, 2018.

A Deep Dive Experiments

In the deep dive experiments, we inspect the effectiveness of each GREEN's design component and also conduct a sensitivity analysis of the parameter used in the scheduling algorithm.

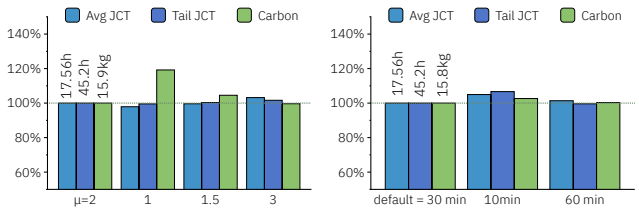
A.1 Understanding the Optimizers

Impact of Energy Efficiency Optimizer (§5.1). The optimizer allocates more resources to jobs with higher energy efficiency. We evaluate the impact of energy-efficiency-based scaling by fixing the number of GPUs to 1 and 2. Our results in Table 4 demonstrate that dynamic scaling decisions impact both energy consumption and execution speed, as evidenced by its worse performance in both carbon footprint and JCT.

Impact of Carbon Footprint Optimizer (§5.2). We replaced the carbon term with an LAS factor: the number of GPU hours consumed by the job. The results, as presented in Table 4, indicate that the cluster-wide carbon footprint increases by approximately 16%, as jobs with higher energy efficiency are no longer prioritized for scaling. We then disabled the temporal shifting by fixing the Shifting Factor to 1. While JCT performance improves slightly, the cluster-wide carbon footprint increases by 20%, as GREEN no longer moves high-power tasks to greener hours.

Scheduler	Job Completion Time		Carbon
	Average	Tail	
Energy Efficiency Optimizer			
GREEN	17.56h	45.20h	15.88kg
No scaling (1 GPU)	19.91h	48.89h	18.38kg
No scaling (2 GPU)	18.53h	46.57h	16.71kg
Carbon Footprint Optimizer			
GREEN	17.56h	45.20h	15.88kg
Carbon Factor = GPU hours	18.30h	47.15h	18.52kg
No Shifting	17.19h	44.99h	18.93kg

Table 4: Ablation experiments of GREEN optimizers.



(a) temporal shifting **(b)** scheduling interval

Figure 13: Sensitivity test of μ (§5.2). μ controls the level of temporal shifting. A larger μ value will shift more high-power jobs to greener hours. Sensitivity test of scheduling interval (§6.3). 30-minute or longer interval works well because carbon intensity changes slowly, allowing less frequent preemptions.

A.2 Sensitivity Analysis

Extent of Temporal Shifting (Figure 13a). In §5.2, we set $\mu\,(\geq 1)$ in controlling temporal shifting. Our tests indicate that setting $\mu = 2$ optimizes the balance between relevant factors. A too high μ skews scheduling, prioritizing high-power jobs unfairly due to an amplified shifting term.

Length of Scheduling Interval (Figure 13b). The interval in MLFQ scheduling influences preemption frequency (§6.3). Our default is 30 minutes, and sensitivity analysis reveals that its exact length has minimal impact on outcomes. This is because carbon intensity, despite significant absolute variations, changes slowly, allowing less frequent preemptions without compromising overall cluster optimization.

B Artifacts

GREEN is designed and evaluated as a scheduler policy on the TACC cluster [32] at the Hong Kong University of Science and Technology. We will release GREEN's source code in TACC's open-source repo [24], which includes the scheduler implementation as a Slurm plugin, a daemon program, and a profiler agent, along with scripts and traces for submitting jobs to the scheduler. More details on our cluster operation and analysis can be found in [13, 37].

The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling

Linsong Guo, Danial Zuberi, Tal Garfinkel, Amy Ousterhout
UC San Diego

Abstract

Preemptive scheduling promises to mitigate head-of-line blocking and enable flexible scheduling while retaining a simple programming model. Despite this, preemption is underutilized in server-side software today. Instead, high-performance datacenter systems and language runtimes often rely on cooperative concurrency, or else use preemption only at very coarse timescales, limiting its effectiveness. A key reason that preemption is underutilized today is that existing preemption mechanisms have high and unpredictable overheads.

Intel recently introduced support for user interrupts, a new feature that offers an opportunity to change this. By enabling interrupts to be sent and received entirely in user space, user interrupts can significantly lower the overhead of preemption. In this paper, we shed light on how user interrupts impact the landscape of preemption mechanisms. We build two user-level schedulers that leverage user interrupts for low-overhead preemption. We find that user interrupts are not a panacea. For example, they provide limited benefits when other software layers constrain the kinds of scheduling policies that can be used. Still, user interrupts can match or exceed the performance of existing mechanisms for all but the highest preemption rates, while achieving much more consistent overheads and retaining a user-friendly programming model.

1 Introduction

Datacenter applications today suffer from high tail latency. For example, a recent study of remote procedure calls (RPCs) at Google showed that 90% of RPC methods were capable of completing in hundreds of microseconds or less, and yet 90% of them had median latencies in the milliseconds, with even higher tail latencies [52]. A myriad of factors contribute to this high tail latency, one of which is queuing of requests at servers. When requests have variable service times, as is the case for Google's RPCs, short requests can queue behind long requests and suffer from *head-of-line blocking*.

Head-of-line blocking is common because many applications have tasks with heterogeneous service times. For example, in hybrid transactional and analytical processing (HTAP) databases [47], short transactional tasks coexist with long-running analytical tasks and may be delayed by them unless the database is provisioned with extra resources [28,35,36,41]. A well-known way to reduce head-of-line blocking without over-provisioning is preemption [39, 56]. With preemption, a scheduler periodically interrupts running tasks, giving shorter, more urgent tasks an opportunity to run and complete quickly. However, preemption is underutilized today.

Though kernel schedulers do implement preemption, they typically do so only at millisecond timescales, which is not sufficient to ensure sub-millisecond tail latencies. Furthermore, kernel-based scheduling adds microseconds of overhead to scheduling operations [23]. As a result, many users turn to userspace schedulers and runtimes instead, employing an *M:N* threading model that schedules *M* user-level threads across *N* kernel threads [6, 10, 33, 37, 45, 49, 57]. We focus on such user-level schedulers.

Today's user-level schedulers rarely implement preemption. Most runtime systems and programming languages eschew preemption in favor of cooperative concurrency, which runs tasks to completion and relies on the developer to manually yield cores as needed (e.g., async tasks in Rust and coroutines in C++). Runtimes like Go [6] that do implement preemption do so only at coarse granularities (every 10 ms) so applications cannot rely on them to provide sub-millisecond tail latency. Similarly, high-performance kernel-bypass systems in datacenters also commonly rely on cooperative concurrency [25, 33, 45, 48, 49, 57].

One reason that preemption is underutilized is the high and unpredictable overheads of the typical mechanisms for user-level preemption today: signals and compiler instrumentation. With *signals*, a dedicated timer thread determines when a user-level thread should be preempted and sends a signal to interrupt it [6, 53]. This involves transitioning to the kernel and back on both the timer core and the core of the preempted thread, adding significant overhead on each preemption (§2.1), resulting in practical limits on preemption frequency.

In contrast, with *compiler instrumentation*, the compiler inserts code throughout a program that checks if it should voluntarily yield the CPU [6,24,37,43], for example, by polling a variable in shared memory. Here, the overhead of each check is much lower than a signal, so it can sometimes enable more efficient fine-grained preemption. However, the checking overhead is hard to predict, as it is heavily dependent on program control flow. Thus this approach can also result in overheads that are high, variable, and workload dependent (§2.2).

Fortunately, a new hardware feature called *user interrupts* recently introduced in Intel's Sapphire Rapids CPU offers the potential for a better approach. User interrupts offer a new way to send and receive inter-processor interrupts entirely in userspace, eliminating the multiple kernel transitions required to send and receive signals (§2.3).

In this work, we seek to understand the potential benefits of user interrupts for preemptive scheduling. We ask: *can user interrupts improve our ability to achieve fine-grained preemptive scheduling with low and predictable overheads, compared to existing approaches?* We answer this question with a measurement study and analysis of two user-level schedulers. While we are not the first to utilize user interrupts for preemptive scheduling [32, 38, 42], we are the first to shed light on the tradeoffs between user interrupts and compiler-based approaches, and to analyze fine-grained preemptive scheduling in the context of a widely used runtime (the Go runtime).

We began by studying the performance impact of preemption with each mechanism on a collection of different workloads (§2). We found, unsurprisingly, that user interrupts consistently outperform signals, reducing the per-preemption overhead from 2.4 μs to 0.4 μs. Consequently, user interrupts enable more frequent preemption at the same cost. In contrast, with compiler instrumentation, overheads can be quite low, but are unpredictable and workload dependent. We investigated several mitigations from other work [37] such as loop unrolling, but we found that rather than solving this problem, they merely shift the costs and complexity to other parts of the system. We also found that the overhead of compiler instrumentation is impacted little by the preemption frequency. Thus, compiler instrumentation shines for ultra-high frequency preemption, e.g., every 5 μs or less, but with a larger preemption quantum, the relative performance depends on the application and user interrupts often provide lower overhead.

Next, we built two user-level schedulers that perform preemptive scheduling with user interrupts (§3). First, **Aspen-KB** extends Caladan [33]—a highly optimized **k**ernel-**b**ypass runtime—with support for user interrupts, allowing us to explore the limits of user-level preemption with user interrupts. Aspen-KB also has support for signals and compiler instrumentation, which makes apples-to-apples comparisons among all three mechanisms possible, without the confounding effects of different scheduling algorithms, programming languages, or runtimes. Second, **Aspen-Go** extends the popular Go runtime, enabling comparisons between preemption mechanisms in the context of a runtime that is used by millions of developers. Both Aspen-KB and Aspen-Go leverage known techniques but carefully apply them to optimize for fine-grained preemptive scheduling. They are available at `https://github.com/LinsongGuo/aspen.git`.

We analyzed the performance of Aspen-KB and Aspen-Go by running applications such as key-value stores (RocksDB [15] and BadgerDB [1]) and a data analysis application (DataFrame [2, 50]) (§4). We have two key findings.

First, in the context of the kernel-bypass runtime Aspen-KB, we found that user interrupts enable Aspen to achieve much better performance than signals. Compared to compiler instrumentation, user interrupts generally deliver better or similar performance when the preemption quantum is greater than 10 μs. With an extremely small quantum (10 μs or lower), compiler instrumentation may offer a small performance benefit over user interrupts. However, compiler instrumentation requires manual intervention to determine whether loop backedges, function calls, or both should be instrumented, and to determine the optimal depth for loop unrolling. Thus, *user interrupts are the best choice for most applications*, due to their improved usability and similar or better performance.

Second, in contrast, we found that user interrupts provided only modest benefits for preemptive scheduling in Aspen-Go. The reason for this is that significantly shrinking the preemption quantum (e.g., to less than 50 μs) proved counterproductive. The design of Go constrains the kinds of scheduling policies that are possible, making preemption less effective at avoiding head-of-line blocking, and every preemption comes at the cost of software overheads in the Go runtime. With preemption at the coarse granularity of 50 μs, replacing signals with lower-overhead user interrupts provides only a small benefit. Thus, *when a system is not fully designed for fine-grained preemptive scheduling, user interrupts provide limited benefits*.

2 Preemption Mechanisms

Three preemption mechanisms are available to userspace schedulers today: signals (§2.1), compiler instrumentation (§2.2), and user interrupts (§2.3). Here, we explore how these mechanisms work and why the first two can have high and unpredictable overheads. We then examine user interrupts (§2.3) and how they can offer lower and more consistent overheads.

2.1 Signals

Signal-based preemption [6, 53, 54] involves two key components: a timer thread and a signal handler. The timer thread typically runs on its own core and is responsible for determining when user-level threads should be preempted.

To preempt a user-level thread running on kernel thread k, the timer thread sends a signal to k. The signal triggers the invocation of k's signal handler function, on the core that k is currently running on. The signal handler then calls into the user-level scheduler, which selects the thread to run in the next quantum. The primary limitation of signal-based preemption is that the signal handler is expensive, incurring multiple costly OS context switches, which ultimately limits preemption frequency.

To explore the basic overhead of each of our three mechanisms, including signals, we ran the following experiment. A dedicated *timer core* sends a notification (signal, user interrupt, etc.) to a single application core at a regular interval (the *preemption quantum*). This notification triggers the application core to run an empty signal-handler function that immediately returns to the application thread. As there is no context switch to a different thread and the scheduler is not invoked, this benchmark illustrates the minimum possi-

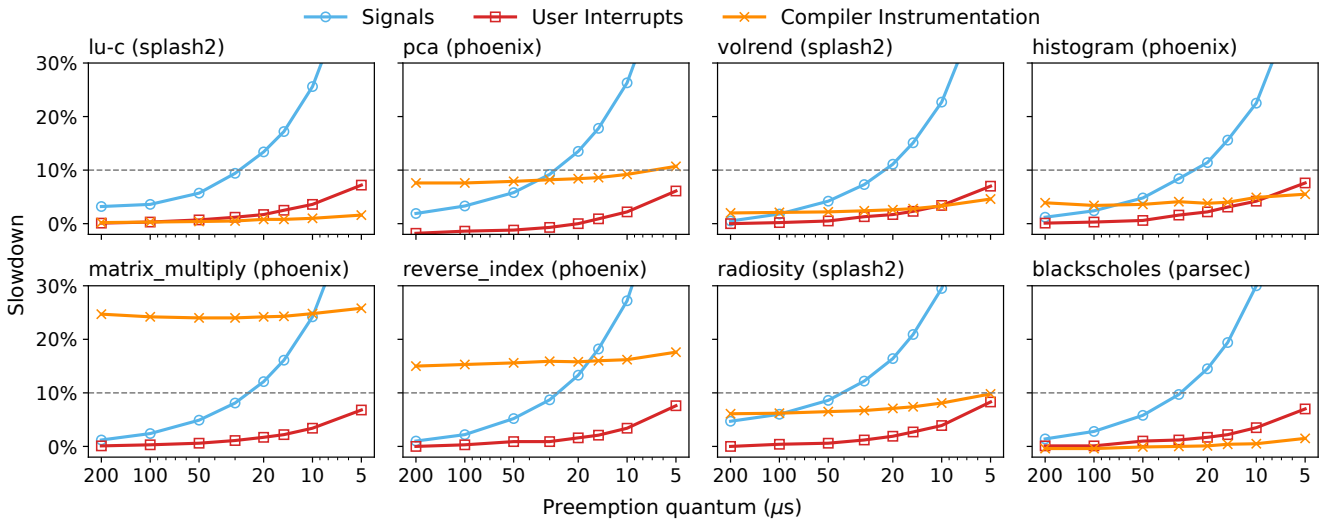


Figure 1: Slowdown of eight programs with different preemption mechanisms. The preemption quantum (x-axis) represents the time between consecutive preemptions and the slowdown (y-axis) indicates how much slower the program runs (as a percentage) compared to running without preemption. For a slowdown tolerance of 10% or less, user interrupts support a 5 μs preemption quantum, while signals support about 30 μs. Compiler Instrumentation is implemented using Concord [37].

ble overhead of each mechanism. We ran this experiment with 24 programs from three benchmark suites: Splash-2 [16], Phoenix [13], and Parsec [26]. In addition, we report the end-to-end cost of preemption (including context switching to a different thread) for latency-sensitive applications in §4.2.

The full results for all benchmark programs are provided in Appendix C. For a representative set of eight programs, Figure 1 shows how much slower each program runs (y-axis) due to receiving periodic signals (blue lines), as we vary the preemption quantum (x-axis). With a large quantum, the slowdown is negligible. For example, with a quantum of 100 μs, the slowdown for most programs is around 3%. However, as the frequency increases so does the cost, so that with a preemption quantum of 10 μs, the eight programs each experience a slowdown of roughly 25% or more, due to the overhead of signals.

Overall, handling a single signal on an application core slows down the program by about 2.4 μs. The dominant cost is the transitions between user and kernel space to receive and deliver the signal; these add up to about 1.4 μs. The remaining cost is due to factors such as additional branch mispredictions and cache misses that occur within the program, due to running kernel code and accessing kernel data structures while handling the signal. Leveraging virtualization features to use normal hardware interrupts rather than signals can reduce the per-signal overhead to 0.6 μs [39], but in practice it can be challenging to deploy systems that require these features.

2.2 Compiler Instrumentation

With compiler instrumentation, preemption is implemented by inserting code at specific points throughout an application that checks if the currently running user-level thread should yield the CPU. Typically, checks are added at function entry points and loop back edges, ensuring that the thread will yield regardless of control flow.

Similar to signals, compiler instrumentation typically relies on a timer thread. However, in this approach, the timer thread indicates that a user-level thread should yield by updating a shared variable in memory. This variable is periodically checked (polled) by the user-level thread to determine if it should yield. While each check is relatively inexpensive, the polling frequency depends on program control flow. Consequently, polling overhead varies across workloads and can be quite high and difficult to predict. This raises several issues. In code with tight loops or small recursive functions, these costs can become prohibitive [14]. Also, at higher preemption frequencies, polling may not take place often enough to ensure that a thread always yields in a timely manner.

Different approaches have been explored to mitigate the problem of overhead. For example, in its first decade Go only instrumented function entry points, to avoid the impact of polling in tight loops. Unfortunately, this required that programmers understand how Go's preemption mechanism worked and manually added explicit calls to the Go scheduler in loops when needed. Aside from the added cognitive burden this imposed, when overlooked, this could lead to high latencies and even program freezes that were difficult to debug [14]. When Go developers attempted to add checks to loop back edges they found that while this only increased the average runtime on a suite of benchmarks by around 7%, at worst it could increase runtime for an individual benchmark by as much as 96% [14]. Thus, they abandoned this approach in favor of signals.

Recent systems have attempted to address these overhead challenges through a collection of different approaches, such as loop unrolling [37, 43]. We use Concord [37] as a repre-

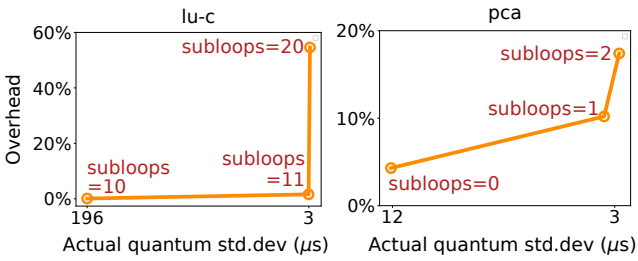


Figure 2: Small changes to the subloops parameter in Concord's compiler instrumentation can significantly impact preemption overhead and timeliness (standard deviation of the measured interval between preemptions). The x-axis uses a log scale.

sentative example that employs compiler instrumentation and reduces polling overhead by placing probes more strategically. We re-run the microbenchmark described above (§2.1), using Concord's LLVM passes to instrument each program. Figure 1 (orange lines) shows the results. While compiler instrumentation can achieve low overhead for some programs (such as *lu-c* and *blackscholes*) when they are well configured, other programs (*pca*, *matrix_multiply*, *reverse_index*) still suffer from significant overhead due to tight loops.

Furthermore, it is non-trivial to find a good configuration for each program, as there are multiple parameters that need to be tuned. For example, Concord features a parameter called `subloops` that limits the number of sub-loops in which probes can be inserted within each outermost loop. Slight variations in this parameter can result in large changes in preemption timeliness and overhead, as shown in Figure 2. For *lu-c* (left), setting `subloops` to 10 results in a high standard deviation of preemption intervals, indicating poor timeliness, while increasing the value to just 11 brings the standard deviation to an acceptable level. Further increasing `subloops` to 20 or more increases preemption overhead to greater than 50%. Similarly, in the benchmark *pca*, increasing `subloops` from 0 to 2 raises the overhead by more than 13%.

Another parameter than can be tuned with compiler instrumentation is the degree of loop unrolling; Concord uses loop unrolling to reduce preemption overhead in programs with tight loops. While loop unrolling can indeed reduce overhead for some programs, such as *blackscholes*, excessive unrolling can increase pressure on the instruction cache and microop cache [11], potentially causing some programs to run more slowly [3, 4]. For example, configuring *matrix_multiply* to unroll loops four times results in a preemption slowdown of 31%, compared to 25% without loop unrolling. Figure 1 illustrates the performance of compiler instrumentation under the best configuration of these parameters that we could find, but tuning these parameters required significant manual effort.

2.3 User Interrupts

Recently Intel's Sapphire Rapids CPUs [20] introduced support for sending inter-processor interrupts directly in user space, via *user interrupts*. Similar to signals, user interrupts enable asynchronous preemption that avoids the polling overheads imposed by compiler instrumentation. In contrast with signals, user interrupts avoid the costly overheads of transitions between kernel and user space.

Normally, inter-processor interrupts (IPIs) are sent from one core to another by programming the APIC from the kernel. This directs control flow to go through the interrupt vector table on the receiving core when an interrupt is delivered. Unfortunately, when communicating user-space events from one core to another, i.e., with signals, this requires multiple expensive protection boundary crossings.

User interrupts offer an alternative by providing a hardware fast path for sending and receiving interrupts. User space code cannot simply be allowed to send interrupts to other processes without access control, so, similar to other kernel-bypass mechanisms, user interrupts delegate access control to the operating system. For a *sender thread* to send user interrupts to a *receiver thread*, both must register with the kernel first. Once registration is complete, the kernel does not participate in the sending and receiving of user interrupts. The processor can directly consult a kernel-managed table about where to route interrupts, allowing a sender thread to ensure that user interrupts are delivered to a specific receiver thread.

To measure the overhead of receiving user interrupts, we again configure a timer core to periodically interrupt an application running on a separate core, this time by sending user interrupts. Figure 1 (red lines) shows that, as with signals, the program slowdown increases as the preemption frequency increases. However, the overhead of handling a single user interrupt is much lower than that of handling a signal—only 0.4 μs compared to 2.4 μs—so the slowdown remains tolerable down to much smaller quantum sizes. User interrupts have so much lower overhead because they avoid transitions between user and kernel space as well as extra branch mispredictions and cache misses within the program.[1] Thus, for a given acceptable amount of application slowdown, user interrupts can enable much more frequent preemption. For example, for a slowdown of at most 10%, user interrupts can support a preemption quantum of 5 μs, whereas for signals this is only about 30 μs.

While user interrupts uniformly outperform signals, the comparison with compiler instrumentation is more nuanced. For small quanta (10 μs or lower), Concord can achieve lower overhead than user interrupts for some programs such as *lu-c*. In contrast, for larger quanta (greater than 10 μs), the slowdown from user interrupts is lower or at least comparable to that of compiler instrumentation, especially in programs with tight loops, where it is 15–25% lower, as seen in *matrix_multiply* and *reverse_index*. This is because, regardless of the quanta, programs using compiler instrumentation must

[1] For example, using perf [12] to measure the *linpack_bench* program [8], we observed that user interrupts result in 29% fewer L1 cache misses and 12% fewer branch mispredictions compared to signals at a 5 μs quantum.

check the instrumented probes at the same locations as they would for smaller quanta, whereas user interrupts incur overhead only when preemption actually occurs.

2.4 Summary

We find that user interrupts consistently provide lower overhead than signals, across all programs we evaluated. While the overhead of compiler instrumentation can be lower than that of user interrupts for very fine-grained preemption with some programs, this overhead varies significantly and can be quite high. Furthermore, striking a good balance between preemption overhead and timeliness can require careful tuning of multiple parameters. While it may be possible to mitigate some of these issues with more sophisticated compiler techniques, with existing approaches to compiler instrumentation, fine-grained preemption remains fragile, entailing unpredictable and potentially high costs. User interrupts offer a compelling alternative due to their consistent overheads, which remain low for all but the highest preemption frequencies.

3 Preemptive User-Level Schedulers

To understand how user interrupts impact preemptive scheduling in practice, we implemented preemptive user-level scheduling in two different runtime systems. The two systems allow us to explore different regions of the design space. First, Aspen-KB extends Caladan [33], a user-level runtime and scheduler implemented in C that uses kernel-bypass networking and is heavily optimized to achieve microsecond-scale latencies. Second, Aspen-Go extends the popular Go runtime, which relies on the underlying operating system's network stack and is designed for programmer productivity rather than bleeding-edge performance. At a high level, these two runtime systems have similar overall architectures. Each runtime core has its own runqueue(s) of user-level threads (or goroutines) and runtime cores balance work using work stealing.

To enable preemptive scheduling, both runtime systems dedicate one core in each runtime to act as the timer core. The timer core periodically sends user interrupts to runtime cores, preempting the currently running user threads. Note that the timer core is not required to process incoming or outgoing packets (though in Go it may), enabling the timer core to scale to support larger numbers of cores than in systems where the timer core is also responsible for processing all incoming network traffic and dispatching packets to cores [37, 39, 43].[2]

In the remainder of this section, we describe four general factors that impacted the design of both of these preemptive schedulers (§3.1), as well as the specific design and implementation of our kernel-bypass scheduler Aspen-KB (§3.2) and Go-based scheduler Aspen-Go (§3.3).

[2]This decision involves a tradeoff. Distributed load balancing via work stealing enables better scalability than systems in which a centralized core dispatches incoming tasks, such as Shinjuku [39], Concord [37], or TQ [43]. Yet, this comes at the cost of potentially less effective load balancing [44].

3.1 Design Factors

3.1.1 Unnecessary Preemptions

Existing systems implement a preemption policy that issues preemptions to all runtime cores at a given interval to preempt all currently running tasks [37,38,42]. However, preemption is not always necessary—for example when there are no queued tasks for a core to run or when a thread has just yielded voluntarily—and preempting unnecessarily slows down running tasks. Eliminating these unnecessary preemptions can significantly reduce overhead.

To reduce the number of preemptions as much as possible, the timer core requires visibility into all sources of queued work (e.g., threads, incoming network packets) so that it can determine if there is another task that the core could handle. In addition, the timer core needs visibility into when context switches between user-level threads occur, so that it can avoid preempting a thread that just began running.

3.1.2 Non-Preemptible Code

User-level threads are not always safe to preempt, often because of nonreentrant code. For example, code that may not be safe to preempt includes:

Scheduler code that holds locks. Since the thread scheduler runs in userspace, a user interrupt may arrive while a thread is executing scheduler code. If that thread holds a lock that protects scheduler state, then it may be unsafe to preempt it, because it will not be possible to acquire the scheduler's lock to reenter the scheduler and context switch to a different user-level thread. This is an example of non-reentrant code.

Other code that holds locks. In addition, preempting other code that holds locks may lead to poor performance, e.g., if it delays critical functionality such as polling the network stack, or if it prevents other threads from being able to run.

Application code that uses kernel-level-thread state. Existing applications often rely on state that is associated with a kernel-level thread. For example, thread-local storage (TLS) is associated with kernel-level threads, and many library functions such as `malloc` rely on TLS [27, 54]. When existing applications are ported to use user-level threads, regions of code that use TLS or library functions that use TLS may not be safe to preempt. This is because multiple user-level threads can reside on the same kernel-level thread, thus sharing the same kernel-thread-local state. As a result, preemption may cause them to have interleaved access to the same state, resulting in incorrect behavior.

In some cases, non-preemptible code is isolated in distinct regions of code, e.g., in calls into the scheduler or to `malloc`. Existing runtimes typically handle these kinds of non-preemptible code by detecting when a signal arrives during non-preemptible code, and deferring or skipping the preemption. For example, in Go, the runtime consults metadata generated by the compiler to determine if the program is at

Task	Rate of Non-Preemptible Code Regions	Use of Extended Registers
SPEC-mcf	only a few	none
linpack_bench	only a few	continuously
base64	only a few	only at initialization
RocksDB GET	4 calls/μs	30 calls/μs
RocksDB SCAN	0.039 calls/μs	38 calls/μs

Table 1: The frequency of non-preemptible code regions and vector register usage across four programs. Numbers in the table with the unit "calls/μs" represent the number of calls into functions involving non-preemptible code or extended registers per microsecond.

an unsafe point, and simply returns to the interrupted goroutine if so [6]. In other runtimes, the linker or developer wraps non-preemptible regions of code so that the runtime can track whether it is currently executing code that is not safe to preempt. If a signal arrives during one of these regions, the signal handler defers the preemption until the user-level thread exits the region [27, 33, 37, 45].

User interrupts provide a promising new approach to handling non-preemptible code, which is to defer interrupt delivery in hardware. With this approach, preemption is deferred by using the new `clui` and `stui` instructions to disable and reenable delivery of user interrupts. The benefit of this approach is that if multiple user interrupts arrive within a region of non-preemptible code, only one will be delivered when user interrupts are reenabled; this interrupt coalescing avoids the 0.4 μs overhead of receiving each unnecessary user interrupt.

However, we found that hardware-based deferral can add significant overhead, because of the cost of the current hardware instructions and because non-preemptible code regions can be short and frequent. We found that a single pair of `stui` and `clui` instructions costs about 18 ns. This is much lower than the 600 ns required to mask and unmask signals, but is still non-trivial. Table 1 shows that some programs execute non-preemptible regions of code infrequently, but that requests for RocksDB [15], a key-value store, frequently call `malloc` and `free`. Deferring preemption in hardware would slow down RocksDB GET requests by 7%, whereas deferring preemption in software adds only 1-2 ns of overhead for each region of code (less than 1% slowdown overall).

3.1.3 Extended Registers

When context switching between user-level threads, it is essential to properly save and restore registers, including extended registers such as SIMD or matrix registers. Table 1 presents the frequency and usage patterns of extended registers in some application code. With compiler instrumentation, the compiler can save caller-saved registers, including extended registers if necessary. In contrast, with user interrupts, the hardware only saves the flags, instruction pointer, and stack pointer, requiring the user-level scheduler to save extended registers. Our approach conservatively saves all registers that might be used.

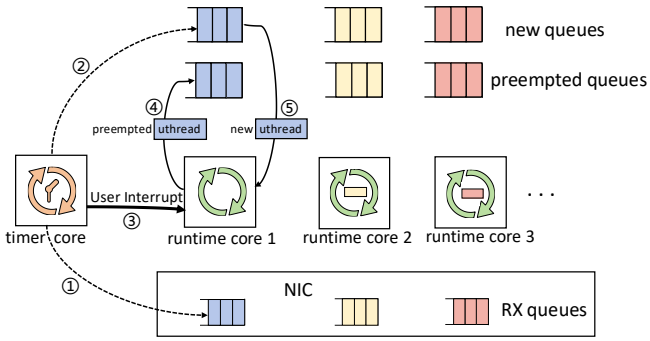


Figure 3: Architecture of Aspen-KB. To preempt a user thread on a runtime core, the timer core first checks if there are pending tasks in its RX queue ① or *new queue* ②. If so, the timer core sends a user interrupt to the runtime core ③. Then, the currently running user thread is suspended and added to the *preempted queue* ④ and the core dequeues and runs the next thread from the *new queue* ⑤.

While this increases the saved state size per thread (e.g., by 2 KB), our measurements show that the overhead is negligible in most cases. A detailed discussion of this and alternative approaches is provided in Appendix A.

3.1.4 Head-of-Line Blocking

Even with periodic preemption of running threads, head-of-line blocking can still occur. First, if the network stack is not polled frequently, newly arriving tasks can experience queueing delays there while lower priority tasks occupy the CPU. Second, if newly runnable tasks are enqueued to a runqueue behind preempted tasks, the new tasks must wait for each preempted task to run for a preemption quantum before they are able to run. Prior work has addressed the latter problem by identifying the task type during processing in the network stack and explicitly prioritizing latency-sensitive tasks [30, 39], but we avoid this approach because task types or service times are not always available a priori. Instead, to prevent head-of-line blocking throughout the runtime, we focus on frequently polling for incoming network packets and prioritizing newly incoming tasks over preempted tasks.

3.2 Kernel-Bypass Preemptive Scheduler

Our first preemptive user-level scheduler, Aspen-KB, is built by extending Caladan [33], a user-level runtime and scheduler implemented in C. Figure 3 shows the architecture of Aspen-KB. Aspen-KB bypasses the kernel for both threading and networking. Each core in each runtime has its own runqueue of user-level threads and its own RX queue which it polls to receive incoming packets. Runtime cores balance work—including both threads and incoming packets—among themselves using work stealing. Aspen-KB dedicates one core per runtime to serve as the timer core.

Unnecessary preemptions. In Aspen-KB, the timer core checks if preemption is necessary at a fixed preemption quan-

tum. However, the timer core may choose not to preempt every runtime core. The timer core has visibility into incoming packet queues (RX queues) and scheduler state (a timestamp indicating when the currently running thread started running) via shared memory, and only preempts the currently running thread if it has been running for at least the preemption quantum and there is other work (incoming packets, runnable threads, etc.) available on that core. We found that this approach significantly reduces the rate of preemptions (§4.4).

Head-of-line blocking. Aspen-KB avoids head-of-line blocking with two techniques. First, it polls the RX queue for newly arriving packets after every preemption; the overhead from this is small due to the kernel-bypass network stack. Second, each core employs a simple two-queue scheduling approach. This is a simplified variant of multi-level feedback queues [29] in which newly runnable threads are initially enqueued to the *new queue*, but once they are preempted, they are enqueued to the *preempted queue* instead. Each core prioritizes the new queue over the preempted queue and uses a longer quantum (e.g., 100 μs) for the preempted queue. With this approach, the system requires no a priori knowledge of how long each thread will run for, and latency-critical threads that run for less than the preemption quantum are still handled quickly.[3]

Implementation. We built Aspen-KB by extending Caladan [33] with a timer core, scheduler features to support user interrupts, and the two-queue scheduling approach. Aspen-KB handles extended registers by saving all general-purpose and AVX-512 registers on every context switch. Additionally, it extends Caladan's shim layer to fully support kernel-level-thread state, including TLS. Specifically, each user-level thread maintains a private copy of its kernel-level-thread state, preventing interleaved access to the same state. Aspen-KB does not modify Caladan's software-based approach to handling non-preemptible code, which defines the functions `preempt_disable()` and `preempt_enable()` to disable and re-enable preemption, respectively, and wraps them around non-preemptible scheduler code. All modifications are implemented in 1,849 LOC. Running an application on Aspen-KB requires a `makefile` to link it with the runtime.

To enable an apples-to-apples evaluation of different preemption mechanisms, we also implement signal-based preemption and compiler instrumentation in Aspen-KB. To implement *signal-based preemption*, we modify Aspen-KB's timer core to send signals in lieu of user interrupts. To implement *compiler instrumentation*, we integrate Concord's approach into Aspen-KB [37]. Specifically, we borrow Concord's LLVM instrumentation passes and apply them to applications. The instrumented applications then run on Aspen-KB, and Aspen-KB's timer core preempts user-level threads by writing to shared memory.

[3] Under overload conditions, preempted threads can languish in the preempted queue. However, we assume that a load balancer will prevent any server from operating in a continuously overloaded state.

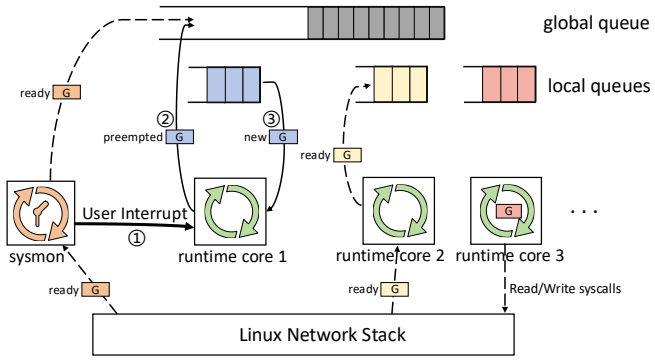


Figure 4: Architecture of Aspen-Go. To preempt a goroutine on a runtime core, sysmon sends a user interrupt to the core ①, triggering it to suspend the current goroutine and place it in the global queue ②, and dequeue a new goroutine from the local queue ③. When an application invokes Go's network APIs, the Go runtime issues a nonblocking system call on its behalf (core 3). To handle I/O completions, runtime cores poll the network stack and add readied goroutines to their local runqueue (core 2). Sysmon can poll as well, but places goroutines in the global queue.

While Junction [32]'s implementation bears similarity to Aspen-KB's—both extend Caladan [33] with support for user interrupts—Aspen-KB differs in its two-queue scheduling approach, its support for compiler instrumentation and signal-based preemption, and how it handles extended registers.

3.3 Go-Based Preemptive Scheduler

Our second preemptive user-level scheduler, Aspen-Go, is built by extending the Go runtime [6] (Figure 4). The Go runtime is designed to support high concurrency and implements user-level scheduling via lightweight threads called goroutines. The scheduler maintains one queue of runnable goroutines for each active core in the runtime, as well as a global queue for preempted goroutines or those that become runnable when a system call returns; goroutines in per-core queues are prioritized over those in the global queue. For networking, Go relies on the underlying operating system's network stack. To perform network I/O, the runtime typically issues non-blocking system calls (e.g., `read` and `write`), and a runtime component called *netpoller* handles polling for completion of those operations (e.g., via `epoll` or `kqueue`).

As described above (§2.2), Go implements two forms of preemption. First, it leverages compiler instrumentation by instrumenting function entry points. To handle programs that do not contain frequent function calls (e.g., those with tight loops), Go also uses signals for periodic preemption [14]. A dedicated thread called *sysmon* preempts running goroutines periodically (every 10 ms by default). The sysmon thread also polls the network stack for incoming packets occasionally, if no other runtime core has polled the network stack for at least 10 ms. The sysmon thread sleeps when it is inactive.

Our goal with Aspen-Go is not to overhaul the Go runtime, but rather to make the minimal set of changes necessary in

order to support fine-grained preemption with user interrupts. We configured the sysmon thread to busy spin instead of sleeping, to enable precise preemption at fine granularity, and additionally made the modifications described below.

Unnecessary preemptions. The sysmon thread has visibility into when a context switch last occurred on each core, and so it can skip preemptions when the current goroutine has been running for less than the preemption quantum. However, because the Go runtime relies on the OS's network stack, the sysmon thread cannot easily determine whether there are packets queued or not. Thus if the current goroutine has exceeded its quantum, the sysmon thread will preempt it, even if there is no other work (e.g., goroutine or packet) available.

Head-of-line blocking. In Go, solely decreasing the preemption quantum provides little performance benefit for networked applications, because incoming packets are not necessarily processed in a timely manner. Incoming packets can be received in one of three ways: (1) an application issues a non-blocking read to check for packets for a specific connection, (2) the scheduler invokes the netpoller to check for incoming packets on any connection; this only occurs when there are no runnable goroutines in the local or global runqueue, or (3) the sysmon thread invokes the netpoller every 10 ms. Thus the Go scheduler generally prioritizes runnable goroutines over incoming network packets. Even with fine-grained preemption of goroutines, packets can remain queued in the network stack until all goroutines in the local and global runqueues have completed or the sysmon thread invokes the netpoller.

To minimize head-of-line blocking in the network stack, the scheduler could invoke the netpoller after every preemption, to check for newly arriving packets. However, in Go, this requires a system call, adding at least 2 μs of overhead to every preemption. We found that this degrades performance prohibitively, especially for applications that rarely perform network I/O. Instead, Aspen-Go modifies the sysmon thread to invoke the netpoller more frequently, every 100 μs. While this allows queued packets to be processed much more quickly, it does not entirely eliminate head-of-line blocking. This is because once sysmon processes incoming packets and marks their corresponding goroutines as runnable, it enqueues them to the global runqueue (to avoid contention for per-core runqueues), where they must queue behind preempted goroutines.

An alternative approach for those willing to tolerate more significant changes to Go would be to leverage a kernel-bypass network stack. For example, this could be done using Junction [32]. In this case, polling the network would be much cheaper and could be performed more frequently. However, given our goal of making minimal modifications to Go and to how it can be deployed, we do not adopt these approaches.

Implementation. We modified the Go runtime (version 1.21) to support preemption via user interrupts and modified the sysmon thread as described above. Additionally, Aspen-Go disables preemption at `Lock()` and re-enables it at `Unlock()` to prevent preemption in critical sections. This is particularly useful since locks are commonly used in networking packages such as `fasthttp`. These modifications amount to 733 LOC.

4 Evaluation

We answer four questions in this evaluation:

1. How do different preemption mechanisms impact the tail latency and throughput of applications in Aspen-KB and Aspen-Go? (§4.1)
2. What is the cost of an individual preemption in Aspen-KB and Aspen-Go? (§4.2)
3. How does the preemption mechanism impact the ideal choice of preemption quantum? (§4.3)
4. What is the impact of Aspen-KB's design decisions? (§4.4)

Experimental setup. We conduct experiments using two dual-socket servers, each with 28-core Intel Xeon Gold 5420+ CPUs operating at 2.0 GHz and 256 GB of RAM. Each server is equipped with a 100 Gbits/s Mellanox ConnectX-6 Dx NIC and the two NICs are connected via a 100GbE Mellanox SN2700 switch. We disable hyperthreads because we found that our workloads achieved higher performance in this configuration. We disable TurboBoost, frequency scaling, and c-states. Both of our servers run Ubuntu 22.04.4. Our load-generating machine uses kernel version 6.8.0 while our server machine uses a custom kernel provided by Intel that is based on kernel version 6.0.0 and supports user interrupts [9, 20].

Kernel-bypass systems evaluated. Ideally we would directly compare Aspen-KB against state-of-the-art systems that use other preemption mechanisms, such as Shinjuku [39] and Concord [37]. However, comparing against these systems is challenging. First, both were designed to run with an older kernel version (4.4.185), which is not supported by Aspen-KB. Second, all three systems use different policies for load balancing work across cores; Shinjuku dispatches work from a single centralized queue, Concord uses JBSQ [40], and Aspen-KB uses work stealing. These differences in load-balancing policy could significantly impact performance for multicore experiments [44], making it difficult to isolate the impact of the preemption mechanism itself.

Instead, we evaluate five variants of Aspen-KB (§3.2). This includes preemptive schedulers with *signal*-based preemption[4] and *user-interrupt*-based preemption (the default version of Aspen-KB). For compiler instrumentation, we evaluate *Concord*, which uses Concord's default configuration that instruments both function calls and loop backedges [37]. We also evaluate *Concord fine-tuned*, in which we carefully configure the subloops parameter, degree of loop unrolling, and whether function calls are instrumented, to achieve the

[4] Shinjuku uses virtualization hardware to further lower the overhead of preemption [39], so its performance would likely fall between that of our signal-based approach and our user-interrupt-based approach.

best performance for each application. We also evaluate *non-preemptive*, a run-to-completion baseline, by running Aspen-KB with preemption disabled. Unless stated otherwise, we configure the non-preemptive baseline to dedicate 25 cores to the evaluated application, and all other systems to dedicate 24 cores to the application and 1 core to the timer.[5]

Go variants evaluated. We evaluate Go under five configurations. *Unmodified* Go refers to the standard Go runtime, version 1.21, which uses both compiler instrumentation and signal-based preemption, every 10 ms. We also applied fine-grained preemption on it, triggering every 50 µs. The remaining configurations run Aspen-Go, as described above (§3.3). *Aspen-Go UINTR* and *Aspen-Go Signals* use user interrupts and signals, respectively. We also evaluate compiler instrumentation alone (*Aspen-Go Compiler*) by disabling preemption via signals or user interrupts. Unless stated otherwise, we configure 8 cores for the application and 1 for sysmon. Go dynamically adjusts its core usage, so at any given time it may use fewer cores.

Workloads. We evaluate three applications:

1. RocksDB [15] (version 5.15.10), a popular key-value database developed by Facebook, runs on Aspen-KB. We run bimodal workloads that include GET and SCAN operations, which take about 1 μs and 260 μs, respectively.
2. C++ DataFrame [2] (version 1.19.0), a data analysis application, runs on Aspen-KB. Our evaluation runs a workload with *decay*, *ad* (Accumulation/Distribution), *rmv* (Rolling Mid Value), *ppo* (Percentage Price Oscillator) and *kmeans*. Each task type makes up 20% of the total number of tasks.
3. BadgerDB [1] (version 1.6.2), a fast key-value database in Go, runs on Aspen-Go. We populate the database with 10 million keys. Our evaluation involves a bimodal workload, with GET operations taking about 5 μs and RangeSCAN operations around 800 μs.

We generate load using Caladan's load generator [33]. For RocksDB and DataFrame, the load generator is configured to send requests over UDP with a Poisson arrival distribution. For BadgerDB, it generates requests over HTTP connections, also following a Poisson arrival distribution.

4.1 Application Performance

In this section, we evaluate the impact of different preemption mechanisms on end-to-end application performance in Aspen-KB and in Aspen-Go. Because the preemption quantum that yields the best performance varies across preemption mechanisms and workloads, we show results with the best-performing quantum. We define the "best" quantum as the preemption quantum for which short operations achieve the highest throughput while tail latency remains below a threshold (e.g., in RocksDB, GET tail latency of at most 50 μs).

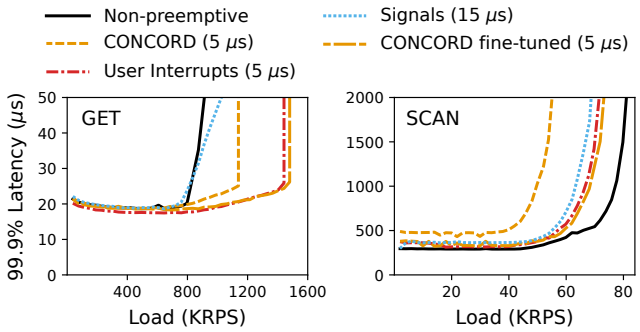


Figure 5: RocksDB performance in Aspen-KB under different preemption mechanisms. The workload is 95% GET and 5% SCAN. KRPS on the x-axis represents kilo-requests per second.

4.1.1 RocksDB (Aspen-KB)

RocksDB [15] exemplifies applications that benefit from an extremely low preemption quantum, as it has short-running GET tasks, which typically take less than 1 µs. Figure 5 shows the results of running RocksDB with 95% GET requests and 5% SCAN requests; experiments with different distributions of GET and SCAN requests (95.5%/0.5% and 50%/50%) yielded similar conclusions.

User-interrupt-based preemption. As shown in Figure 5, user interrupts (UINTR) are able to mitigate head-of-line blocking and significantly improve performance compared to a system without preemption. For example, for a tail latency limit of at most 50 μs, user interrupts improve GET throughput by 58.2% compared to the non-preemptive system.

Signal-based preemption. In contrast, signal-based preemption provides little benefit for GETs compared to no preemption and it increases the tail latency for SCANs compared to most other approaches. There are three reasons for this lackluster performance. First, as illustrated above, receiving a signal entails an overhead of about 2.4 μs (§2.3), which can increase execution time for SCANs by up to 16.7% with a preemption quantum of 15 μs. Second, with signals, the timer thread suffers from poor scalability (§4.4), due to the overhead of sending signals. For this workload, the timer core can only trigger 43% as many preemptions with signals as it can with user interrupts; it is not able to reliably sustain the target preemption quantum. Finally, receiving signals also exhibits poor scalability due to lock contention [54], so adding additional timer cores does not improve performance.[6]

Compiler instrumentation. Compiler instrumentation can yield similar improvements as user-interrupt-based preemption, as illustrated by the "CONCORD fine-tuned" lines in

[5] We also dedicate one core to Caladan's IOKernel. The IOKernel is responsible for reallocating cores across applications, but we disable this. Thus in our experiments the IOKernel is only used during initialization; this functionality could be combined with the timer core in the future.

[6] Note that Shinjuku's virtualization-based approach reduces the overhead of sending/receiving interrupts to 14%/48% of the overhead with signals and enables Shinjuku to scale to support at least 22 cores [39].

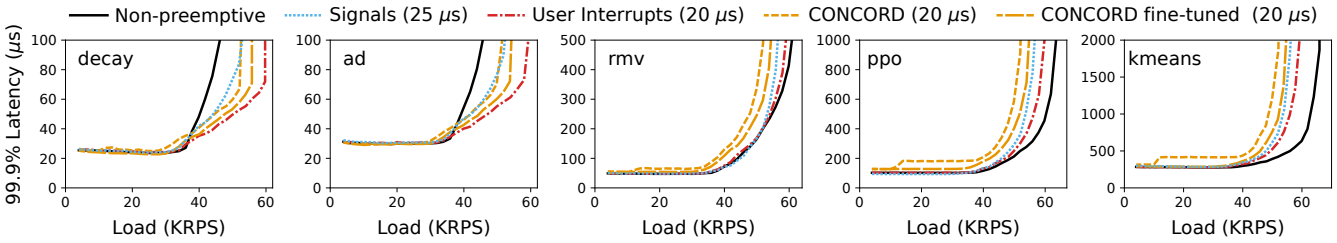


Figure 6: DataFrame performance in Aspen-KB under different preemption mechanisms. Each task type constitutes 20% of the total task count.

Figure 5. However, achieving this performance required deviating from Concord's default instrumentation. With the default instrumentation, a single SCAN operation triggers over 95,000 preemption checks, resulting in a 31.2% slowdown with a 5 μs quantum (as shown by "CONCORD"). GET operations suffer similarly. In the fine-tuned variant, we disable instrumentation of function calls, which are frequent in RocksDB. This reduces the number of preemption checks per SCAN to 5,000 and decreases the slowdown to only 2.3%.

4.1.2 DataFrame (Aspen-KB)

We evaluate DataFrame [2] with a workload that includes 2 short tasks (*decay* (5 μs) and *ad* (7 μs)), one medium task (*rmv* (28 μs)), and two longer tasks (*ppo* (75 μs) and *kmeans* (250 μs)). Because its tasks are longer than the short GETs in RocksDB, DataFrame generally performs best with a longer preemption quantum. Figure 6 shows the results.

With a tail latency limit of 100 μs, user interrupts achieve the highest throughput for the two types of short-running tasks, about 30% higher than the non-preemptive system and 9% higher than the fine-tuned Concord. For longer tasks, user interrupts do sacrifice some throughput compared to the non-preemptive system. However, with the 20 μs preemption quantum, user interrupts typically cause less than 2% slowdown for programs (§2.3), thus improving the throughput of short requests with minimal sacrifice.

Compiler instrumentation improves performance for short tasks, but at the cost of significant throughput for longer tasks. Despite efforts to fine-tune Concord's parameters (the number of instrumented subloops and loop unrolling count), both *rmv* and *ppo* still experience significant preemption overhead (3.3 μs) due to tight loops (§4.2). In contrast, user interrupts incur only 0.37 μs and 0.32 μs of overhead, respectively.

4.1.3 BadgerDB (Aspen-Go)

We ran BadgerDB [1] with unmodified Go and Aspen-Go to evaluate how effectively different Go variants can mitigate head-of-line blocking. As shown in Figure 7, simply decreasing the preemption frequency (e.g., to 50 μs) in unmodified Go does not significantly reduce head-of-line blocking. This is because even with frequent preemption, the unmodified Go runtime does not attempt to poll incoming packets from network stack as long as there are available goroutines in

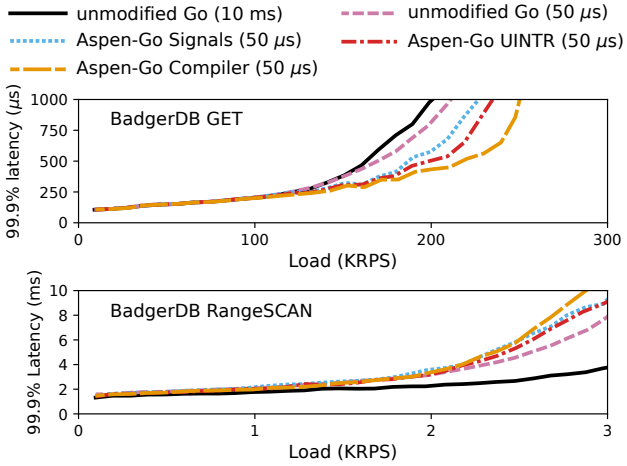


Figure 7: BadgerDB performance with unmodified Go and Aspen-Go. The workload consists of 99% GET and 1% RangeSCAN.

local or global queues (§3.3). As a result, the incoming packets remain blocked in the network stack and more frequent preemption only adds extra slowdown to running goroutines.

In contrast, Aspen-Go UINTR can achieve 17.5% higher throughput for GET requests than unmodified Go (while maintaining a tail latency of at most 1000 μs), primarily because Aspen-Go actively polls ready packets from the network stack, allowing them to run earlier than in the unmodified version. However, the performance gains from user interrupts in Aspen-Go are significantly smaller than those in Aspen-KB for the following reasons. First, goroutines resulting from newly arrived packets may be placed at the back of the global runqueue (e.g., if they are polled by sysmon) and still need to wait, whereas new user threads in Aspen-KB can be placed in local queues dedicated to new threads and processed more quickly. Second, sysmon cannot detect whether packets are queued or not due to its lack of visibility into the Linux network stack, and thus longer tasks are preempted even when there are no other tasks to handle. Third, the cost of context switching during preemption in Aspen-Go is higher than in Aspen-KB (§4.2). Overall, the design of the Go runtime makes it difficult to adopt some policies that could help reduce head-of-line blocking.

Finally, compiler instrumentation (Aspen-Go Compiler) achieves 6% higher GET throughput than user interrupts, for two reasons. First, polling at function calls serves multiple purposes beyond preemption and, therefore, cannot be disabled in

Aspen-Go UINTR/Signals. Thus, Aspen-Go-UINTR/Signals still incur the overhead of polling. Second, even though these systems are configured with the same preemption quantum (50 μs), Aspen-Go UINTR/Signals experiences 22% fewer preemptions than Aspen-Go Compiler, making it less effective at mitigating head-of-line blocking. This is because with compiler instrumentation, preemptions occur only at safe points, whereas user interrupts or signals can be delivered anywhere and are ignored if they arrive at unsafe points. In the BadgerDB workload, most unsafe points occur during the execution of `memmove`, which is heavily used in Go, e.g., in assignment statements.

4.1.4 Summary and Takeaways

We found that the preemption mechanism does not significantly impact performance in systems like Go, which are not fully designed for fine-grained preemption. Below, we summarize key takeaways for Aspen-KB and similar low-overhead, kernel-bypass preemptive systems.

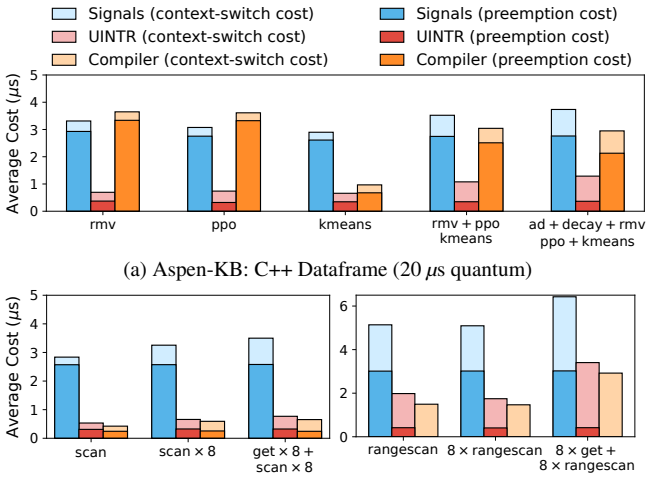
The best choice of preemption mechanism for a given application depends on the optimal preemption quantum. One heuristic for determining a reasonable preemption quantum is that it should be at least as long as the tail runtime of "short" tasks, i.e., tasks that should not be preempted. The guidance in § 4.3 can help more precisely identify the optimal quantum, which might be longer than the runtime of short tasks in order to avoid over-preemption. In either case, if the chosen preemption quantum is at least 10 μs, user interrupts generally deliver better or similar performance compared to compiler instrumentation. The DataFrame results and some benchmark programs in Figure 1 illustrate this.

In scenarios where extremely low quanta (less than 10 μs) are necessary, compiler instrumentation may offer better performance, as long as an application is not dominated by tight loops (§2.2). However, the performance gains are limited, as observed in the RocksDB results. This is because compiler instrumentation only reduces the overhead of the preemption mechanism and does not address the context switching overhead (§4.2). Overall, we conclude that user interrupts are likely a better choice for the majority of applications.

4.2 The Cost of Preemption

In Figure 1 we measured the cost of preempting a thread, executing an empty handler function, and then immediately returning to the same thread (*preemption cost*). In this section, we also report the *context-switch cost* of preemption, which includes the costs of traversing the scheduler, context switching to a different thread, and any resulting cache contention. Together, the preemption cost and context-switch cost represent the total overhead incurred per preemption.

Figure 8 illustrates the preemption and context-switch costs for various preemption mechanisms, applications, and task combinations. Each application uses its optimal quantum as identified in §4.1. The Aspen-Go results (Figure 8c) compute costs relative to a different baseline than those in Aspen-KB. In Go, compiler instrumentation is used for multiple purposes (e.g., to check if more stack space is needed) and cannot easily be disabled. Thus all data in Figure 8c is collected with compiler instrumentation enabled and we cannot quantify the preemption cost associated with compiler instrumentation.



(a) Aspen-KB: C++ Dataframe (20 μs quantum)

(b) Aspen-KB: RocksDB (5 μs)

(c) Aspen-Go: BadgerDB (50 μs)

Figure 8: The costs of preemption (*preemption cost*) and context switching (*context-switch cost*), for 3 applications. The x-axis represents different task combinations, e.g., $get \times 8 + scan \times 8$ indicates that 8 GET tasks and 8 SCAN tasks run concurrently on one core.

We make four observations. First, with larger preemption quanta, the compiler instrumentation costs are amortized over fewer preemptions, yielding higher per-preemption costs, as illustrated by the higher costs for Dataframe than for RocksDB; this is consistent with Figure 1. Second, the context-switch cost in Aspen-KB is not negligible and can even exceed the preemption cost in some workloads (e.g., with 8 GET tasks and 8 SCAN tasks). This further indicates that, although compiler instrumentation may incur lower overhead than user interrupts in some workloads, the improvement in total cost is relatively small. Third, context-switch costs are slightly higher with user interrupts than with compiler instrumentation for some workloads, e.g., 36% higher for the *rmv*, *ppo*, *kmeans* combination. This may be due to saving extended registers (§3.1.3), which is unique to user interrupts. Finally, the context-switch cost in Aspen-Go is higher than in Aspen-KB (1.3-3.0 μs vs. 0.2-0.9 μs), mainly because of the Go runtime's more complex scheduler logic (e.g., checking safepoints). Despite this, user interrupts and compiler instrumentation still incur significantly lower total cost compared to signals.

4.3 Preemption Quantum

We evaluate how the preemption quantum impacts performance in Aspen-KB by running RocksDB with several different preemption quantum. In each case, we determine the

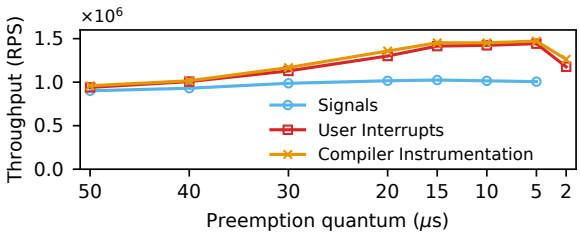


Figure 9: RocksDB GET throughput with different quanta.

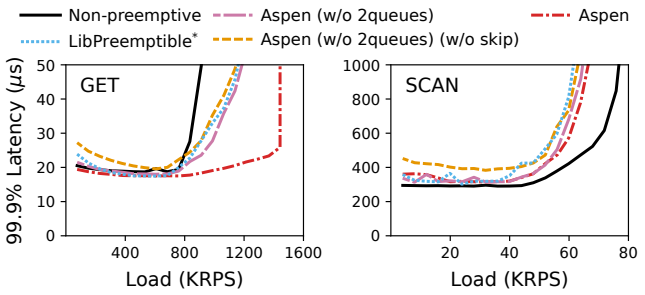


Figure 10: RocksDB performance achieved by LibPreemptible*, Aspen-KB, and other variants. "w/o 2queues" indicates that the 2-queue policy is disabled, while "w/o skip" means that the policy of skipping unnecessary preemption is disabled.

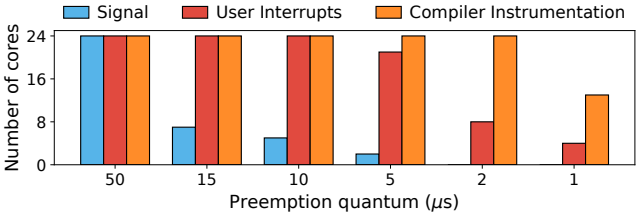


Figure 11: The number of application cores that a single timer core can support under the three preemption mechanisms.

max GET throughput for which GET operations maintain a tail latency of 50 μs or less; Figure 9 shows the results. For all mechanisms, decreasing the preemption quantum improves performance up to a point; beyond this point, the overhead of extra preemptions outweighs the benefits of mitigating head-of-line blocking and throughput degrades. For both user interrupts and compiler instrumentation, the best quantum for this workload is 5 μs; for signals it is 15 μs. The best quantum depends on the workload; across different applications and ratios of GET/SCAN requests in RocksDB, we have observed ideal quanta for user interrupts ranging from 2 μs to 20 μs.

In Aspen-Go, we found that with a slowdown tolerance of at most 5%, signal-based preemption limits the quantum to no lower than 100 μs. In contrast, user interrupts further reduce it to 30–50 μs, providing tighter tail latency guarantees (Appendix B).

4.4 Impact of Aspen-KB's Design Decisions

Preemption Polices. We evaluated the effectiveness of Aspen-KB's policies and compare them to the approach of LibPreemptible [42], a recent system that uses user interrupts for preemption. We run Aspen-KB with various policies disabled, and also approximate LibPreemptible's policies in Aspen-KB. Specifically, for a given load, we test different preemption quanta and select the best result as LibPreemptible's result, modeling its ability to find a suitable quantum through adaptive adjustment. This approximation achieves the best possible performance for a workload that does not have significant variation over time; we denote this as LibPreemptible*.While LibPreemptible's APIs allow users to use two queues to prioritize short requests over long, the user-defined queue for preempted tasks must be shared across all cores, as users can not control which cores tasks run on. Thus it would be difficult to support a two-queue policy in a scalable way with LibPreemptible, so we do not apply the two-queue policy to LibPreemptible*.

Figure 10 show the performance of RocksDB achieved by LibPreemptible, Aspen-KB, and other variants of Aspen-KB. Aspen-KB's two-queue policy yields significant performance improvements for both kinds of requests, by allowing incoming requests to be prioritized, and by reducing the amount of preemptions experienced by longer SCAN requests. The policy of skipping unnecessary preemptions benefits SCAN requests the most at lower loads, when there is less likely to be another task to run. Under such load, even though the preemption quantum is 5 μs, most preemptions can be skipped and the interval between consecutive preemptions is typically more than 30 μs.

Compared to LibPreemptible*, Aspen-KB handles head-of-line blocking more effectively, keeping the tail latency of short requests low. This is largely because of Aspen-KB's two-queue policy. Furthermore, Aspen-KB reduces the slowdown to preempted tasks. LibPreemptible adjusts the preemption quantum based on past latencies and offered load but updates slowly (every 10 seconds), making it unresponsive to real-time load changes. Instead of explicit quantum adjustments, Aspen-KB skips unnecessary preemptions, allowing rapid, real-time adaptation based on the current load.

Timer Scalability. We evaluated the scalability of Aspen-KB's timer core. To stress the timer core, we run a workload that consists of long-running tasks so that the timer core needs to issue a preemption to each core every preemption quantum. We vary the quantum, and for each quantum measure the maximum number of application cores to which the single timer core can reliably send at least 99% of the expected number of preemptions. Figure 11 shows the results.

With a preemption quantum as large as 50 μs, all three mechanisms are able to support all 24 cores that are available to them. With this large quantum, the timer core mostly calls `rdtsc` and busy-spins. With smaller quanta, the timer scalability depends on the cost of triggering preemptions, which varies significantly across the three mechanisms. Sending a signal takes 1.7 μs, a user interrupt 200 ns, and the compiler-based approach incurs negligible overhead, requiring just a

single shared memory write. Consequently, with a 5 μs quantum, the timer scales to only 2 cores with signals but reaches 22 with user interrupts and 24 with compiler instrumentation. This benchmark represents a lower bound on the performance of the timer core; in more realistic workloads one timer core can typically support more application cores due to Aspen-KB's preemption policy, which can skip 70% of preemptions.

5 Related Work

User-level threading. To avoid the overhead of kernel-based scheduling, many systems schedule threads in userspace, adopting an *M:N* threading model. Examples include the Go runtime [6], μThreads [18], Caladan [33], Arachne [49], and many others [10, 21, 37, 45, 53, 54, 57]. Aspen-KB and Aspen-Go are examples of such user-level schedulers.

Preemption mechanisms. Many existing user-level schedulers adopt cooperative concurrency; those that are preemptive typically employ signal-based preemption or compiler instrumentation. For example, Go [6] and Argobots [53] use signal-based preemption. Shinjuku improves upon signals by virtualizing the APIC for lower overhead IPIs [39].

Compiler instrumentation can be implemented with or without a timer core. With a timer core, the compiler inserts instrumentation that checks shared memory—which is written to by the timer core—to determine when the core should yield. Go [6], Wasmtime [19], and Concord [37] take this approach. Alternatively, the instrumentation itself can estimate how much time has elapsed, by tracking the number and/or duration of instructions that have elapsed, and yield the CPU once the metric exceeds a threshold [24, 34, 43].

Both user-level interrupts [46, 51] and user-level exceptions [55] were first proposed decades ago. The new "user interrupts" feature in Intel CPUs makes user-level interrupts available as a preemption mechanism for userspace schedulers today [20]. LibPreemptible [42], Junction [32], and Skyloft [38] use user interrupts for core scheduling or timeslicing. These works are complementary to ours; none studies how user interrupts compare to compiler instrumentation, addresses head-of-line blocking within the network stack, or studies fine-grained preemptive scheduling in the context of a widely used runtime such as Go. Finally, xUI proposes processor extensions that further reduce the overhead of receiving user interrupts [22]; these can be used to improve the performance of both Aspen variants.

6 Conclusion

Preemption can allow us to build more efficient and responsive systems by enabling precise scheduling and mitigating head-of-line blocking. We explored implementing preemption with user interrupts, a new hardware feature. We found that when a system is not constrained by its scheduler's design, user interrupts can offer performance comparable to or better than compiler-instrumentation, while offering a simpler developer experience and more predictable performance.

Acknowledgments

We thank our shepherd Malte Schwarzkopf and the anonymous reviewers for their feedback. We also thank the Supermicro JumpStart program for providing access to 4th generation Intel Xeon processors for our initial benchmarks. This work was funded in part by a Google Research Scholar Award and a gift from Cisco.

References

[1] BadgerDB: Fast key-value db in go. https://github.com/dgraph-io/badger.

[2] C++ DataFrame for statistical, financial, and ML analysis. https://github.com/hosseinmoein/DataFrame.

[3] Deep diving into llvm loop unroll. https://yashwantsingh.in/posts/loop-unroll.

[4] Excessive loop unrolling. https://github.com/llvm/llvm-project/issues/42332.

[5] Go performance dashboard. https://perf.golang.org/search?q=upload%3A20171003.1+%7C+upload-part%3A20171003.1%2F3+vs+upload-part%3A20171003.1%2F1.

[6] The Go programming language. https://go.dev/.

[7] Intel® 64 and ia-32 architectures software developer's manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[8] LINPACK_BENCH - the linpack benchmark. https://people.math.sc.edu/Burkardt/c_src/linpack_bench/linpack_bench.html.

[9] Linux kernel with support for user interrupts. https://github.com/intel/uintr-linux-kernel.

[10] Loom - fibers, continuations and tail-calls for the JVM. https://openjdk.org/projects/loom/.

[11] Optimizing subroutines in assembly language. https://www.agner.org/optimize/optimizing_assembly.pdf.

[12] perf: Linux profiling with performance counters. https://perfwiki.github.io.

[13] Phoenix benchmark suite. https://github.com/kozyraki/phoenix.

[14] Proposal: Non-cooperative goroutine preemption. https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md.

[15] RocksDB: A persistent key-value store for flash and RAM storage. https://github.com/facebook/rocksdb.

[16] Stanford parallel applications for shared-memory (SPLASH-2) programs. https://github.com/staceyson/splash2.

[17] User interrupt compiler guide. https://github.com/intel/uintr-compiler-guide/blob/uintr-gcc-11.1/UINTR-compiler-guide.pdf.

[18] uThreads: Concurrent user threads in C++(and C). https://github.com/samanbarghi/uThreads.

[19] Wasmtime. https://docs.wasmtime.dev/.

[20] x86 User Interrupts support. https://lwn.net/Articles/869140/.

[21] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.

[22] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. Extended user interrupts (xUI): Fast and flexible asynchronous notification without polling. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025.

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

[24] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *Proceedings of ACM International Conference on Programming Language Design and Implementation*, pages 1249–1263, 2021.

[25] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems*, 34(4):1–39, 2016.

[26] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[27] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *USENIX Annual Technical Conference*, pages 465–477, 2020.

[28] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. Bytehtap: bytedance's htap system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment*, 15(12):3411–3424, 2022.

[29] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, pages 335–344, 1962.

[30] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 621–637, 2021.

[31] Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–25, 2013.

[32] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 55–73, 2024.

[33] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 281–297, 2020.

[34] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. Compiler-based timing for extremely fine-grain preemptive parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[35] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[36] Kaisong Huang, Jiatang Zhou, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. Low-latency transaction scheduling via userspace interrupts. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2025.

[37] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 466–481, 2023.

[38] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. Skyloft: A general high-efficient scheduling framework in user space. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 265–279, 2024.

[39] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 345–360, 2019.

[40] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *USENIX Annual Technical Conference*, pages 863–880, 2019.

[41] Guoliang Li and Chao Zhang. Htap databases: What is new and what is next. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 2483–2488, 2022.

[42] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 922–936, 2024.

[43] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient microsecond-scale blind scheduling with tiny quanta. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–319, 2024.

[44] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for microsecond-scale tasks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–18, 2022.

[45] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 361–378, 2019.

[46] Mike Parker. A case for user-level interrupts. *ACM SIGARCH Computer Architecture News*, 30(3):17–18, June 2002.

[47] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner*, pages 4–20, 2014.

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

[49] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware thread management. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 145–160, 2018.

[50] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 315–332, 2020.

[51] Daniel Sanchez, Richard M Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. *ACM SIGARCH Computer Architecture News*, 38(1):311–322, 2010.

[52] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 498–514, 2023.

[53] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level

threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018.

[54] Shumpei Shiina, Shintaro Iwasaki, Kenjiro Taura, and Pavan Balaji. Lightweight preemptive user-level threads. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 374–388, 2021.

[55] Chandramohan A Thekkath and Henry M Levy. Hardware and software support for efficient exception handling. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, 1994.

[56] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[57] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 195–211, 2021.

A Extended Registers

When context switching between user-level threads, care must be taken to ensure that the appropriate registers are saved and restored. This includes any extended registers that are being used by the preempted thread, such as SIMD registers or matrix registers.

When compiler instrumentation is used as the preemption mechanism, preempting a user-level thread simply involves calling a function. In this case, the registers that are in use are known at compile time, and the compiler can save just the caller-saved registers that are in use, including extended registers if necessary [31].

In contrast, with user interrupts, the user-level scheduler is responsible for saving almost all registers. When a user interrupt is delivered, the hardware saves the flags, instruction pointer, and stack pointer. The compiler also saves general-purpose callee-saved registers if they are used by the interrupt handler function [17]. However, the scheduler is responsible for saving any additional registers, including SIMD registers. The scheduler typically has no knowledge of which registers a program is using at runtime, and thus for correctness must conservatively save all extended registers.

We ran a microbenchmark program to measure the cost of saving extra extended registers. In this benchmark, multiple user-level threads perform pointer chasing, each within their own independent region of memory. All threads run on the same core; we preempt them periodically and measure the overhead per preemption. We found that, in the worst case, saving all AVX-512 registers (2 KB) could add over 700 ns of overhead to each preemption, compared to saving only the general-purpose registers. This worst-case overhead occurred when the memory accessed collectively by the threads just barely fit into a given cache layer (e.g., the L1 cache), so that the extra memory consumed by saving extra registers caused evictions to the next layer of cache. These evictions make the benchmark program run much slower, as the chasing pattern was deliberately designed to be highly random, minimizing the effectiveness of cache prefetching. However, for other memory region sizes, the overhead of saving all AVX-512 registers per context switch in this benchmark program remained below 40 ns.

We considered three alternative approaches for handling extended registers. First, one could defer preemption during code that uses them, but unfortunately this approach does not work well for many applications. As shown in Table 1, some workloads such as linpack use SIMD registers continuously throughout their execution, with no clear point to defer preemption to. With RocksDB operations, SIMD registers are confined to specific functions, but these functions are called so frequently that deferring preemption would slow down operations by several percent.

A second approach is to use the instructions `XSAVEC` and `XRSTOR` to save and restore registers, as in prior work [32].

These instructions detect which "state components" are in use and save only active components [7]. However, we found that using these instructions added 100-150 ns to each context switch, compared to only 10-30 ns when saving all AVX-512 registers one by one, due to the high fixed costs of these instructions. Furthermore, these instructions track state components only at coarse granularity (e.g., all AVX-512 registers are in the same state component), limiting their ability to optimize which registers are saved.

Finally, it may be possible to reduce overheads by having the compiler identify the set of registers that are used by each program and create custom functions that save and restore only those registers. For the applications we evaluated, saving and restoring all AVX-512 registers on every context switch yielded sufficiently low overhead, but if register state continues to grow in the future, or if applications use additional extended registers such as matrix registers, it may be beneficial to explore this approach.

B Preemption Quantum in Aspen-Go

We ran several microbenchmarks on both Aspen-Go UINTR and Aspen-Go Signals to gain a broader understanding of the preemption overhead at different preemption quanta in Aspen-Go. For each benchmark program, we ran 10 goroutines of the same program concurrently on a single core, with preemption occurring at every quantum. These microbenchmarks are from the benchmark suite [5], which Go developers used to measure preemption slowdown [14]. Figure 12 presents the results. With a slowdown tolerance of at most 5%, using signals as the preemption mechanism limits the quantum to no lower than 100 μs. In contrast, user interrupts provide finer-grained preemption, allowing for preemption quanta as small as 30-50 μs.

C Preemption Slowdown

Table 2 presents the preemption overhead of three mechanisms—signals, user interrupts, and compiler instrumentation—measured across 24 programs from the Splash-2, Phoenix, and Parsec benchmark suites. Section 2 highlights representative benchmarks and compares the three preemption mechanisms.

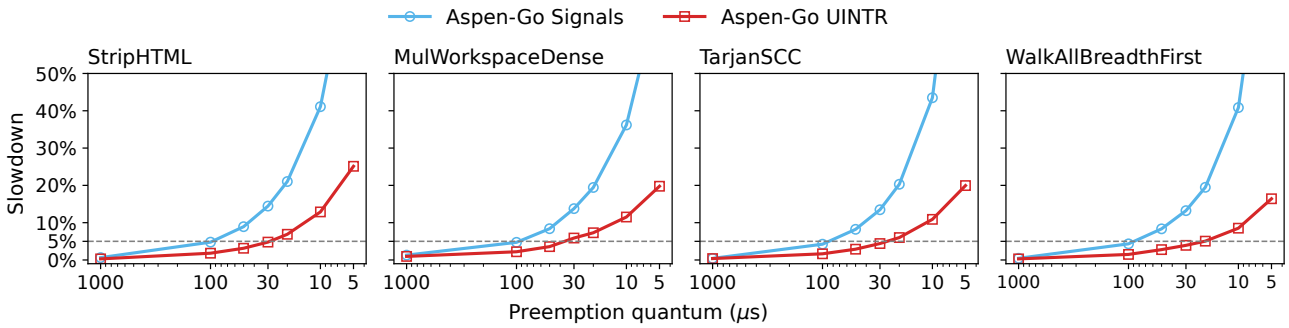


Figure 12: The preemption slowdown with different preemption quanta in Aspen-Go UINTR and Aspen-Go Signals.

Program	**User Interrupts**			**Concord**			**Signals**		
quantum	50 μs	20 μs	5 μs	50 μs	20 μs	5 μs	50 μs	20 μs	5 μs
water-nsquared	0.6%	1.2%	6.2%	0.7%	1.2%	3.3%	5.1%	12.6%	48.7%
water-spatial	1.3%	1.9%	9.3%	1.5%	1.8%	3.5%	5.1%	12.2%	47.6%
ocean-cp	0.3%	1.9%	6.1%	1.7%	2.4%	3.9%	4.8%	11.1%	47.7%
volrend	0.8%	1.9%	7.8%	2.2%	2.6%	4.6%	4.2%	11.1%	45.3%
fmm	0.6%	1.6%	7.3%	-1.7%	-1.7%	0.7%	5.2%	13.4%	47.6%
raytrace	0.3%	1.7%	7.1%	1.8%	2.2%	3.9%	5.5%	12.6 %	52.1%
radix	0.6%	1.7%	6.5%	-0.6%	-0.4%	-0.2%	4.7%	11.1%	43.7%
fft	0.8%	1.9%	7.3%	2.5%	2.7%	4.4%	5.2%	12.2%	43.0%
lu-c	0.7%	1.8%	7.4%	0.4%	0.8%	1.6%	5.7%	13.4%	49.6%
lu-nc	0.7%	1.6%	6.3%	-0.8%	-0.7%	-0.4%	3.2%	10.2%	44.3%
cholesky	0.9%	2.0%	7.7%	6.9%	7.3%	9.6%	4.8%	12.0%	46.9%
radiosity	0.5%	1.8%	8.5%	6.5%	7.1%	9.8%	8.6%	16.4%	55.2%
histogram	0.7%	1.8%	6.8%	3.6%	3.8%	5.5%	4.8%	11.4%	45.7%
pca	0.9%	2.2%	7.9%	7.9%	8.4 %	10.7%	5.8%	13.5%	51.2%
string_match	0.7%	1.8%	7.0%	4.5%	4.9%	7.1%	4.3%	11.2%	45.3%
linear_regression	0.7%	1.7%	6.9%	0.1%	0.5%	2.5%	4.7%	11.8%	44.0%
word_count	0.6%	1.6%	6.7%	1.9%	2.1%	3.0%	5.0%	12.4%	47.3%
matrix_multiply	0.6%	1.7%	7.0%	24.0%	24.2%	25.8%	4.9%	12.1%	47.9%
reverse_index	1.0%	2.4%	8.2%	15.6%	15.8%	17.6%	5.2%	13.3%	54.6%
blackscholes	0.6%	1.8%	7.8%	-0.1%	0.1%	1.5%	5.8%	14.5%	59.8%
fluidanimate	0.5%	1.5%	7.7%	0.0%	0.1%	0.9 %	4.6%	12.5%	55.9%
swaptions	0.3%	1.4%	7.3%	5.9%	5.7%	7.2%	7.4%	17.4%	65.5%
canneal	0.5%	2.4%	7.2%	-0.2%	-1.1%	0.9%	5.1%	12.5%	58.5%
streamcluster	-0.5%	1.8%	6.6%	4.9%	6.2%	7.0%	6.1%	16.3%	60.8%

Table 2: Preemption slowdown with different preemption mechanisms in benchmark programs.

Securing Public Cloud Networks with Efficient Role-based Micro-Segmentation

Sathiya Kumaran Mani*[1] Kevin Hsieh*[1] Santiago Segarra[2]
Ranveer Chandra[1] Yajie Zhou[3] Srikanth Kandula[1]
[1]*Microsoft* [2]*Rice University* [3]*University of Maryland*

Abstract

Securing network traffic within data centers is a critical and daunting challenge due to the increasing complexity and scale of modern public clouds. Micro-segmentation offers a promising solution by implementing fine-grained, workload-specific network security policies to mitigate potential attacks. However, the dynamic nature and large scale of deployments present significant obstacles in crafting precise security policies, limiting the practicality of this approach. To address these challenges, we introduce a novel system that efficiently processes vast volumes of network flow logs and effectively infers the roles of network endpoints. Our method integrates domain knowledge and communication patterns in a principled manner, facilitating the creation of micro-segmentation policies at a large scale. Evaluations with real-world deployment demonstrate that our solution significantly surpasses existing algorithms in role inference accuracy. We implement our solution as an end-to-end system and demonstrate that it is up to 21.5× more cost-efficient than Apache Flink, a widely used open-source stream processing system.

1 Introduction

The growth and criticality of public clouds are accompanied by persistent security threats. Prominent security incidents, such as those involving SolarWinds [2], Midnight Blizzard [71], and Snowflake [68], reveal a spectrum of attack vectors from compromised credentials to exploited software providers and trust chains. These incidents often lead to massive leaks of sensitive data, significant economic losses, and deterioration of customer trust [1, 86, 87]. Consequently, there is a pressing need for the development of enhanced security technologies to protect the integrity of data centers.

One promising solution that has emerged is *micro-segmentation* [89], a technique grounded in the zero-trust security model [63, 78]. Recognizing that preventing every cyberattack is a nearly unattainable task, micro-segmentation operates under the assumption that every endpoint can be compromised. It therefore aims to contain the impact of potential breaches by creating isolated μsegments within the network. Under this model, resources in different μsegments can only communicate with each other if explicitly permitted by a set of fine-grained, workload-specific policies. This restrictive approach significantly limits the potential damage of a breach. Several vendors have already introduced micro-segmentation offerings [28, 41, 55, 88]. The market for micro-segmentation is worth several billion dollars in annual revenue and boasts a robust growth rate of over +20% year-over-year [25].

However, two significant challenges arise when implementing micro-segmentation on a large scale: precise security policies and cost-effective monitoring. First, existing solutions [28, 41, 55, 88] rely on human administrators to assign a μsegment label to *every* endpoint. This μsegment creation process necessitates a deep understanding of the workloads and continuous updates to keep up with changes in software or environment. This manual approach does not scale for large systems that leverage millions of resources [72] and have large teams owning different parts of cloud software [19, 33, 45]. Mislabeling or outdated labels can result in legitimate access being denied or unnecessary exposure to potential attacks. Second, a micro-segmentation solution necessitates the capability to observe and monitor network behavior. This involves the near real-time gathering and analysis of network telemetry, a process that can inflate costs significantly. Indeed, leading solutions [28, 55] add a substantial *extra* cost to the infrastructure, such as virtual machines (VMs) and containers, ranging from 16% to as high as 71%. This financial burden presents a considerable obstacle to their extensive adoption.

Objectives and Techniques. We introduce ZTS (Zero Trust Segmentation), a novel micro-segmentation system designed for large-scale public cloud networks. The objective of ZTS is to facilitate the creation of precise, scalable security policies, while also offering a cost-effective solution for gaining visibility into network behavior and conducting efficient monitoring. ZTS consists of two core components:

(1) Deployment-Specialized Auto-Segmentation: To gen-

* These authors contributed equally.

erate μsegments on a large scale, we introduce a novel auto-segmentation algorithm predicated on the inference of network endpoints' roles. Our algorithm is grounded in the observation that within a large deployment, a multitude of network nodes often fulfill identical roles, such as replicas, microservices, lambdas, etc. These roles can be designated as the μsegments, as nodes performing the same function can adhere to the same network security policy. A key insight underpinning our role inference algorithm is that the definition of role varies across different functionalities and deployments. As such, our algorithm judiciously combines domain-specific knowledge, communication patterns, and deployment-specific information. This approach then generates an autoencoder that is uniquely tailored to each deployment, thereby enhancing its accuracy in inferring roles and μsegments.

(2) Cost-Effective Communication Graph Generator: To minimize the costs of observing and monitoring network behavior, we carefully design a unique system architecture that efficiently generates a complete communication graph from a large volume of network flow logs at a very low cost. Our architecture leverages compute endpoints (e.g., VMs) to process and batch a substantial number of network flow logs into a database endpoint (e.g., SQL servers) and employs advanced query optimization techniques to construct communication graphs. This design enables our system to accommodate large-scale deployments with a minimal amount of compute, memory, and network resources.

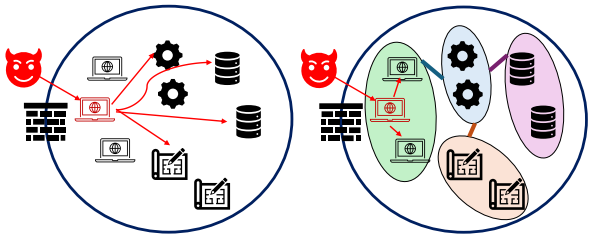
Implementation and Evaluation. We build ZTS as a complete micro-segmentation system that provides continuous communication graph generation, advanced role inference for auto-segmentation, and security policy authoring and enforcement. To evaluate the accuracy of our auto-segmentation algorithm, we curate ground truth data across 11 first-party and third-party deployments, encompassing communication graphs with tens of thousands of nodes and millions of edges. Our evaluation demonstrates that our role inference algorithm significantly outperforms state-of-the-art algorithms [56, 59, 72, 93], achieving an average Adjusted Rand Index (ARI) [53][1] improvement of 0.37.

To evaluate cost-effectiveness in network observation and monitoring, we implement an alternative communication graph generator using Apache Flink [38], a popular open-source stream processing system. Our evaluation across different deployments shows that ZTS is 7.5× faster than Apache Flink at only 35% of the cost, making it 21.5× more cost-efficient. Deploying ZTS only introduces an extra 0.5% of the cost of the VMs, which is significantly cheaper than existing micro-segmentation solutions.

Contributions. We make the following contributions:

- We introduce a novel algorithm that infers network endpoints' roles by incorporating domain-specific knowledge and deployment-specific information in a principled manner. This approach significantly enhances the accuracy and scalability of micro-segmentation.
- We design a cost-effective system architecture for generating communication graphs from network flow logs, which reduces resource usage and operational costs in implementing micro-segmentation at a large scale.
- We build an end-to-end micro-segmentation system and show significant improvements in role inference accuracy and cost-efficiency compared to existing solutions, validated using large-scale real-world deployments.

[1] ARI is a widely used metric for measuring clustering quality, adjusted for chance. It ranges from -0.5 for highly discordant clusterings to 1.0 for identical clusterings.



(a) Without Micro-segmentation (b) With Micro-segmentation

Figure 1: An example of how micro-segmentation prevents lateral movement by attackers.

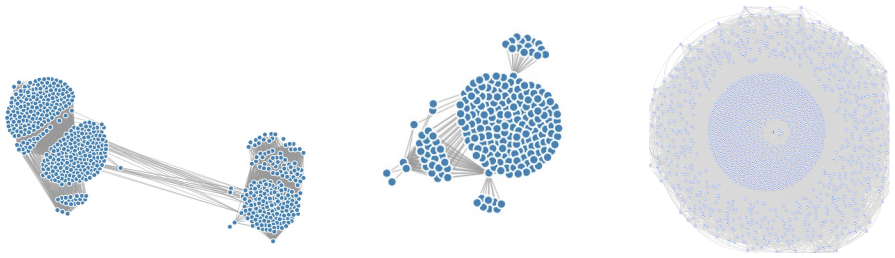
2 Background and Motivation

We begin by providing background on micro-segmentation, followed by a detailed discussion of the major challenges in realizing this concept in practice.

2.1 Micro-Segmentation

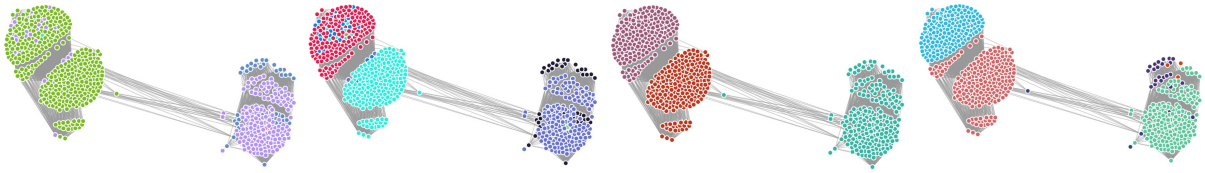
Micro-segmentation [89] is a security technique that involves dividing a network into smaller, more manageable, so-called, *μsegments*, each with its own set of security controls. The concept emerged as a response to the increasing complexity and interconnectivity of modern IT environments, where traditional perimeter-based defenses proved insufficient. As illustrated in Figure 1(a), a compromised network node permits an attacker to propagate laterally across other nodes without limitations, potentially resulting in significant data breaches and system damage. However, by implementing micro-segmentation, as depicted in Figure 1(b), each μsegment effectively minimizes the attack surface. This is achieved by confining communications to predefined pathways, thereby enhancing the containment of breaches. Today, micro-segmentation is considered a core component of zero-trust security architectures [63, 78].

Deploying a micro-segmentation solution involves two key phases to ensure effective implementation and enhanced security. The first phase is *μsegment definition and policy authoring*, where administrators conduct a comprehensive assess-



(a) K8s PaaS IP-graph (b) Portal IP-graph (c) KQuery IP-graph

Figure 2: Unsegmented IP-graphs for the three real-world deployments



(a) Simrank segmentation (b) Simrank++ segmentation (c) Conn.-weight. modularity (d) Byte-weighted modularity

Figure 3: Applying other segmentation strategies, inspired by prior work, on K8s PaaS's IP-graph

ment of the current network architecture and communication patterns. This is often facilitated by generating a comprehensive communication graph using network flow logs from the workloads. Following this assessment, administrators define the relationships between assets and μsegments, authoring security policies that specify which μsegments can communicate with each other and under what conditions. A pair of resources can communicate with each other only if explicitly allowed by the policies; i.e., the default will be to deny [34, 39].

The second phase is *implementation and monitoring*, where these security policies are enforced using a micro-segmentation platform, which may involve configuring virtual firewalls and security controls at the hypervisor or network level. Once implemented, the system should be continuously monitored to ensure compliance with the defined policies. Additionally, μsegments and policies need to be regularly updated to accommodate changes in configurations, deployments, and workloads.

2.2 Challenges in Authoring Security Policies

While creating micro-segmentation may appear straightforward in simple examples, such as in Figure 1, in practice, creating and maintaining μsegments can be a daunting challenge for administrators. To illustrate this issue, we examine three real-world deployments in Table 1.

Figure 2 depicts the unsegmented IP-level communication graphs for each deployment. As these graphs show, the number of nodes and communication patterns can quickly become complex. Our interviews with developers reveal that even they often do not fully understand these graphs, as communication can be initiated by various layers in a complex software stack. The current state-of-the-art requires a human

	#IPs mon.	Graph Size: #nodes (#edges)		#Records
		IP Graph	IP-port Graph	/minute
Portal	4	4K (5K)	13K (13K)	332
K8s PaaS	390	541 (12K)	1.3M (3M)	68K
KQuery	1400	6K (1.3M)	12M (79M)	2.3M

Table 1: Cloud deployments and some aspects of their communication graphs that we built and analyzed in this paper. The source telemetry is flow-level summaries every minute with schema as in Table 3.

administrator to manually tag each node with a μsegment label so that the system can suggest security policies. However, this approach is neither scalable nor error-free. Our investigation of real-world deployments (detailed in Table 5) reveals that only 12% to 23% of network nodes possess useful information—such as machine functions, tags, or function-based names—that could potentially serve as μsegment labels. However, the availability and consistency of such information exhibit considerable variation across different deployment scenarios. Furthermore, administrators need to stay updated on all role changes—such as when workloads migrate, scale up, or scale down [13], or when a software update alters communication behavior—which occurs frequently in large-scale deployments.

Study of Existing Role-Inference Algorithms. Could an algorithm assist with μsegment labeling? Fundamentally, there are far fewer roles than resources in a cloud setting because, for redundancy and scalability, it is common for multiple resources to share the same role. Intuitively, nodes that play the same role share communication patterns and may be inferred based on the similarity of neighbors a resource communicates with and the nature of the conversation. This is similar to the role inference problem in the graph mining literature [51, 58]. Figure 3 presents the outcomes using some established tech-

Core #	VM	Illumio	Illumio vs. VM	Akamai	Akamai vs. VM
2	$549		65%		71%
4	$1,100	$354	32%	$390	35%
8	$2,194		16%		18%

Table 2: Pricing comparison between Azure general-purpose VMs [7], Illumio Core [14], and Guardicore Segmentation [3]. All pricing represents annual costs, normalized to a single VM and shown in US Dollars. The pricing for Illumio Core and Guardicore Segmentation was from Azure Marketplace.

niques on one deployment. There are clear discrepancies among them, and it is unclear which one is more correct.

To deepen our understanding, we conduct a study utilizing popular, state-of-the-art role inference algorithms, including (1) Jaccard: similarity of neighboring nodes [59], (2) SimRank [56]: graph structural similarity, (3) GAS [93]: structural embeddings with graph neural networks, and (4) CloudCluster [72]: grouping nodes based on the adjacency matrix. We curate ground truth data for 11 real-world deployments (details in Section 6.2). The full results are summarized in Table 6 (Section 6.2). Our findings indicate that, although certain algorithms demonstrate good performance in specific deployments, none consistently achieve high performance across a diverse range of deployments. Additionally, the performance of all algorithms diminishes as the scale of deployments increases. Overall, existing algorithms tend to produce role inference results that significantly deviate from the ground truth, with none achieving an average Adjusted Rand Index (ARI) greater than 0.5.

Our analysis reveals that effective and consistent role inference for micro-segmentation remains a major challenge in practice, attributable in part to several factors: (1) The phenomenon driving role similarity in cloud communication graphs—namely, identical code executing across multiple resources for redundancy and scalability—does not stem from human interactions, unlike in social and product recommendation graphs [51]. (2) Structural resemblance within a communication graph does not guarantee role similarity; thus, an algorithm capable of harnessing additional communication attributes is necessary to differentiate such instances. (3) Given that the optimal role inference algorithm may vary among deployments or evolve over time, a dynamic, learning-based algorithm that adjusts according to feedback could potentially surpass all static approaches.

2.3 Cost of Micro-Segmentation Solutions

Micro-segmentation is an expanding market with solutions from multiple vendors [28, 41, 55, 88]. We examine the costs associated with two leading vendors specializing in micro-segmentation platforms [28, 55], comparing these expenses to the infrastructure costs of VMs with varying core counts.

As Table 2 shows, the pricing of both solutions is comparable and incurs significant extra costs, constituting 16% to 71% of the total VM costs. Although licensing fees may not directly equate to computational costs, this analysis highlights the substantial financial burden on organizations seeking to implement micro-segmentation.

Our analysis finds that the primary cost of micro-segmentation lies in the collection and analysis of network telemetry to achieve near real-time network visibility. This involves the continuous processing of extensive network flow logs to generate a comprehensive view. Initial attempts with off-the-shelf solutions, such as Apache Flink [38] and Apache Spark [92], proved to be cost-prohibitive, adding over 10% to the overall VM expenses.

Summary. The lack of an effective role inference algorithm for network endpoints, coupled with the high costs of network behavior monitoring, are significant barriers to achieving the full potential of micro-segmentation. These challenges form the core of our design requirements for ZTS.

3 Overview of ZTS

We introduce a novel end-to-end system, ZTS, specifically designed to facilitate micro-segmentation in public cloud environments at scale, while maintaining cost efficiency. The system achieves scalable security policy formulation with an innovative role inference algorithm, which integrates network domain knowledge and communication patterns based on the characteristics of each deployment. The algorithm allows for effective collaboration with deployment owners or network administrators to fulfill the diverse requirements of various deployments. Additionally, ZTS minimizes the costs associated with micro-segmentation through a cost-efficient communication graph generator. Figure 4 provides an overview of the ZTS architecture.

Telemetry and Graph Generator. The input to ZTS consists of network flow telemetry, which periodically summarizes the network flows generated by cloud assets such as VMs and containers within a deployment. Most major cloud providers offer this telemetry with minimal impact on customer workloads [10, 11, 17] (Section 5.1 provides more details). This telemetry stream is processed by ZTS's *communication graph generator* (① in Figure 4) to produce a communication graph, similar to those in Figure 2, where each node represents an IP address and each edge summarizes all flows between the corresponding IP pairs. Section 5.2 outlines the detailed system architecture of our communication graph generator.

Policy Authoring. During the security policy authoring phase, ZTS's *role inference trainer* (② in Figure 4) facilitates the creation of μsegments by inferring the roles of each network endpoint. The role inference trainer takes into account: (1) communication patterns from the graphs, (2) detailed information from flow telemetry (e.g., communication ports), (3) information from cloud resources (e.g., asset type), and (4)

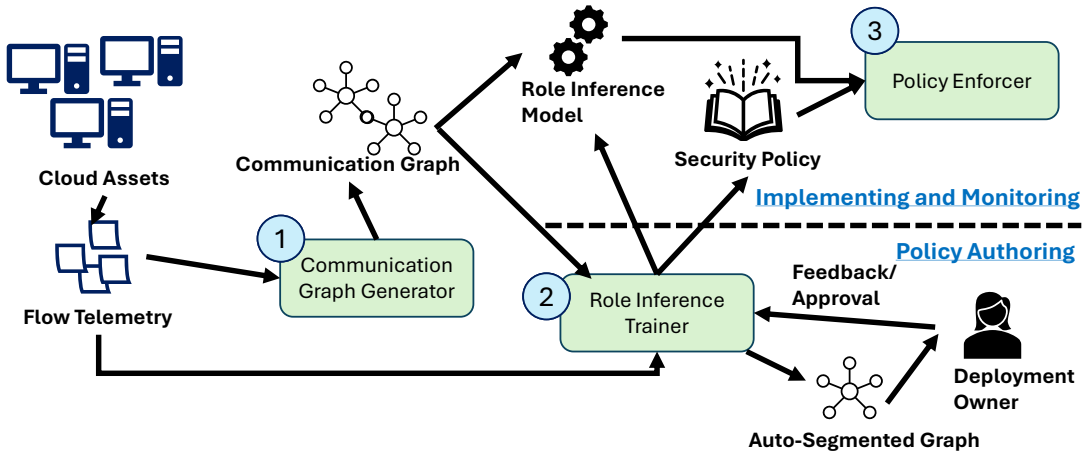


Figure 4: Overview of ZTS.

feedback from the deployment owner (e.g., tags of a subset of assets). The trainer iteratively collaborates with deployment owners to stabilize the micro-segmentation and suggests security policies for each μsegment based on the auto-segmented graphs. Section 4 describes the details of our role inference algorithm.

Implementing and Monitoring. The policy authoring phase generates a role inference model and a set of security policies. During the implementation phase, the *policy enforcer* (③ in Figure 4) maps these security policies from the μsegment level to the IP address level and writes the low-level policies into network security controllers (e.g., [8]) for enforcement. ZTS continuously detects role changes and infers roles for new network endpoints using the role inference model. These changes are applied by the policy enforcer and sent to deployment owners for periodic reviews and model retraining.

4 Role Inference

4.1 Problem Formulation

Our data collection procedure described in §5 enables the construction of a featured IP graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{A}, \mathbf{X})$. The node set $\mathcal{V}$ of cardinality $|\mathcal{V}| = N$ contains all IP addresses in the network, and a directed edge in $\mathcal{E}$ exists between two IPs if traffic was measured from one to the other. The total amount of traffic (measured in bytes) from node i to node j is encoded by the entry A_{ij} of the directed adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$. Notice that we can interpret the i-th row $\mathbf{a}_i^\top$ of $\mathbf{A}$ as features associated with node i capturing the total traffic that this node sent to every other IP in the network. The matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$ contains additional node features, where the i-th row $\mathbf{x}_i^\top$ collects the D features associated with IP i. These features encapsulate information beyond the total traffic sent to other IPs, which is already encoded in $\mathbf{A}$. This includes details such as the primary ports used by an IP, statistical information about the connections—such as the mean and variance of bytes per connection or bytes per packet—and even the count of graphlets [26] or motifs [84] within the communication graph to which a given node belongs. Additionally, they encompass system-related information, such as names and cloud resource SKU information. Our goal is to leverage the information in $\mathcal{G}$ to cluster the IPs into roles $\mathcal{R}$, where two IPs within the same cluster have similar activity. Ultimately, this can allow us to perform communication graph visualization and network management at the role level (instead of the IP level), thus facilitating the generation of rules and improving interpretability (§2.1). The procedure to achieve this goal is detailed next and summarized in Figure 5.

4.2 Infusing Domain Knowledge into Role Inference

In principle, we can concatenate $\mathbf{a}_i$ and $\mathbf{x}_i$ to generate a summary of the activity of IP i, and use these representations to cluster all IPs. However, notice that there is noise and redundancy in these representations, motivating a dimensionality reduction procedure. To be more precise, even if two IPs have the same role, we would not expect the corresponding rows of $\mathbf{A}$ to be identical [72]. Instead, we can think of these as noisy measurements of the canonical activity of that role. In order to recover these main modes of activity, we can perform principal component analysis (PCA). Similarly, the different features in $\mathbf{X}$ originate from the traffic data, so we can expect redundancy among the features, leading to an effectively rank deficient matrix. Based on these considerations, we perform a two-step dimensionality reduction procedure where we first apply a linear method (PCA) followed by a non-linear method (autoencoder). Formally, we consider the singular value decomposition (SVD) of $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ and keep only the top p components to form $\tilde{\mathbf{X}} = \mathbf{U}_p\Sigma_p$, where $\mathbf{U}_p \in \mathbb{R}^{N \times p}$ contains the p leading (left) singular vectors and $\Sigma_p \in \mathbb{R}^{p \times p}$ is a diagonal matrix containing the corresponding singular values. We select p such that the amount of variance that needs to

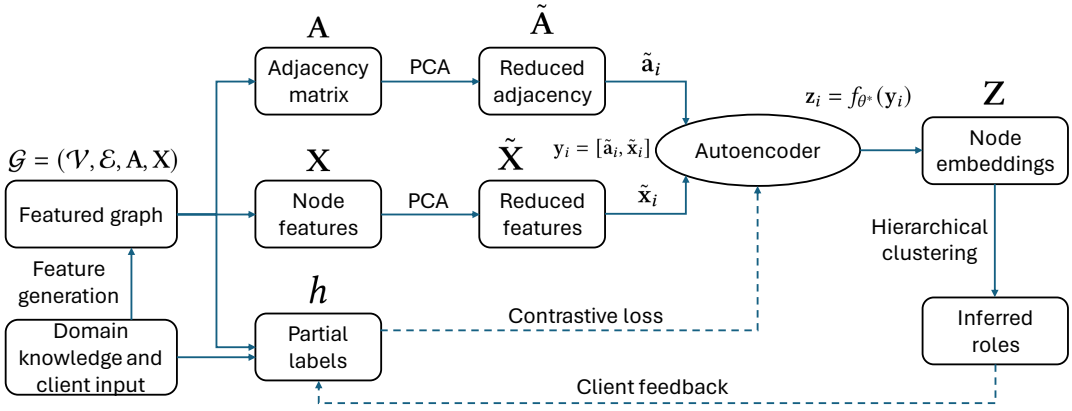


Figure 5: Role inference pipeline. Node embeddings, obtained through an autoencoder, are hierarchically clustered to group the nodes into role classes. Domain knowledge and potential client input is incorporated through feature generation and partial role labels.

be explained is greater than a pre-specified threshold, and we use 99% as a default value in our experiments. A similar procedure is followed to obtain $\tilde{\mathbf{A}} \in \mathbb{R}^{N \times p}$. We denote the concatenation of the i-th rows of $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{X}}$ by $\mathbf{y}_i = [\tilde{\mathbf{a}}_i, \tilde{\mathbf{x}}_i] \in \mathbb{R}^{2p}$.

We further reduce the dimensionality of $\mathbf{y}_i$ via an autoencoder regularized by a contrastive loss. To do so, we assume that we have access to the true roles of a small subset of the IPs. This can be achieved either by relying on domain-inspired rules, such as specific ports associated with pre-established roles, names given to the virtual machines, or an IP-based lookup table, or by obtaining feedback from the network administrator. We denote this partial labeling heuristic as $h : \mathcal{V} \to \mathcal{R} \cup \emptyset$ such that $\hat{r}_i = h(i)$ is the estimated role of node i. Notice that the heuristic might not have enough information to determine the role of a given node. In these cases, $\hat{r}_i = \emptyset$. Let us denote by $\mathcal{L}$ the set of nodes assigned some role label by this heuristic, i.e., $\mathcal{L} = \{i \in \mathcal{V} \,|\, \hat{r}_i \neq \emptyset\}$. Respectively denoting by $f_\theta : \mathbb{R}^{2p} \to \mathbb{R}^d$ and $g_\psi : \mathbb{R}^d \to \mathbb{R}^{2p}$ the parametric encoder and decoder (both multi-layer perceptrons) of our autoencoder, we train the parameters θ and ψ so as to minimize the loss

$$L(\theta, \psi) = \sum_{i \in \mathcal{V}} \|\mathbf{y}_i - g_\psi(f_\theta(\mathbf{y}_i))\|_2 - \tag{1}$$

$$\alpha \sum_{r \in \mathcal{R}} \sum_{\substack{i,i' \in \mathcal{L} \\ h(i)=h(i')=r}} \log \left(\frac{\exp(\mathrm{sim}(f_\theta(\mathbf{y}_i), f_\theta(\mathbf{y}_{i'}))/\tau)}{\sum_{\substack{i'' \in \mathcal{L} \\ h(i'') \neq r}} \exp(\mathrm{sim}(f_\theta(\mathbf{y}_i), f_\theta(\mathbf{y}_{i''}))/\tau)} \right).$$

In (1), sim is a pre-specified similarity measure in the embedding space (such as cosine similarity), τ is a scalar temperature parameter, and α controls the relative weight between both terms of the loss. Notice that the first term in (1) is the classical autoencoder loss that seeks to minimize the difference between the input and output, while the second term pushes together the embeddings of nodes known to have the same role (and separates those known to have different roles). Denoting by θ^* and ψ^* the parameters that minimize the loss in (1), we consider the embedding $\mathbf{z}_i = f_{\theta^*}(\mathbf{y}_i)$ as our concise representation of the activity of IP i, which we gather in the matrix $\mathbf{Z} \in \mathbb{R}^{N \times d}$. Finally, similar to prior work [72], we implement a hierarchical agglomerative clustering algorithm to $\mathbf{Z}$ to obtain our inferred roles for the IPs in the network.

We want to emphasize the following aspects of our role inference algorithm:

1) *User input and feedback*: The labeling heuristic h permits the incorporation of user input into the embeddings $\mathbf{Z}$ through the contrastive loss term in (1). Moreover, after executing the role inference procedure once, the user can be consulted regarding the clusters identified, in a manner similar to active learning [80]. This could involve querying the user about clusters that are the least separable. If discrepancies are found between the clustering output and the user's understanding, the user's knowledge can be fed back seamlessly into h, resulting in improved role inference.
2) *Incorporation of domain knowledge*: Unlike generic role inference methods for graphs [58], domain experts have prior knowledge of what shapes the role of an IP in a network in terms of, e.g., ports used and traffic statistics. Both the partial labeling heuristic h and the node features $\mathbf{X}$ are ideal vehicles to infuse this domain knowledge into our role inference method.
3) *Relation to existing work*: Our pipeline (Figure 5) subsumes most existing methods that cluster nodes based on structural similarities [49, 72]. For example, the method in CloudCluster [72] boils down to performing clustering directly on the rows of $\tilde{\mathbf{A}}$, the adjacency matrix after dimensionality reduction. The incorporation of the autoencoder and, more importantly, the consideration of node features $\mathbf{X}$ and partial labels h constitute key additions with respect to this baseline. We emphasize their value both with a toy example (§4.3) as well as in real-world networks (§6.2).

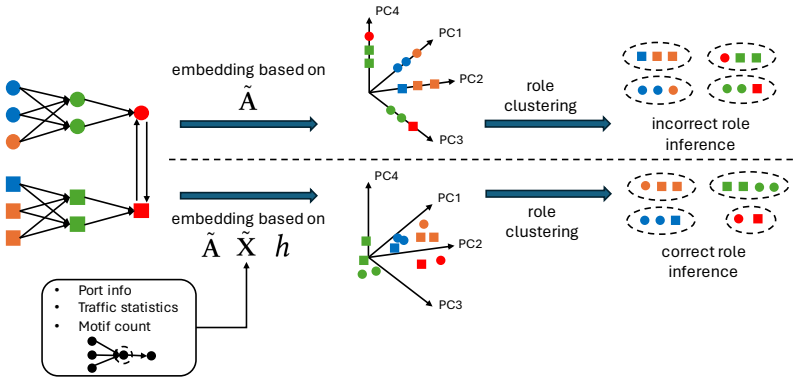


Figure 6: Illustrating the value of domain knowledge. The incorporation of domain knowledge through node features and partial labels leads to the correct inference of roles in the toy network under study.

4.3 Illustrating the value of our proposed pipeline

Let us consider a simple toy network shown in Figure 6-left, where the $N = 12$ nodes have been colored according to their role and the shapes (circles versus squares) represent the top and bottom subnetworks. Traffic originates from two classes of nodes corresponding to different services (blue and orange), this is then relayed through the green nodes to the red nodes. This can be an idealistic representation of a microservice architecture, where the blue and orange nodes are different microservices, the green nodes are the memory object caching systems, and the red nodes are the databases.

We first analyze the roles inferred in this toy network based solely on the adjacency (structural) embeddings $\tilde{\mathbf{A}}$ in Figure 5. Recall that, as explained in §4.2, this is effectively the method proposed in CloudCluster [72]. Assuming that all communication links in Figure 6 have similar total load, the adjacency matrix $\mathbf{A}$ has four dominant principal components. To see this, notice that the rows of $\mathbf{A}$ encode who a given node is talking to, and there are four distinct cases in our toy network (the green circles, the green squares, the red circle, or the red square). Moreover, in $\tilde{\mathbf{A}}$, all the nodes will have an embedding lying on top of one of these four principal components, as shown in Figure 6. Given this marked separation in $\tilde{\mathbf{A}}$, CloudCluster outputs the (incorrect) role labels in Figure 6-top right, irrespective of the specific hierarchical clustering algorithm of choice. This undesirable outcome is resolved by incorporation of $\mathbf{X}$ and h as in our proposed pipeline. For example, the incorporation of port-based features can help distinguish the blue from the orange nodes. Similarly, adding features based on small motifs such as the one illustrated in Figure 6 can bring the embeddings of all green nodes together (notice that only green nodes are at the center of these motifs). Additionally, partial labels of, e.g., blue nodes in the top and bottom parts of the network will help bring the embeddings of all blue nodes together through the contrastive loss in (1). As a result, the 4-dimensional embedding in Figure 6 is more informative than that solely based on $\tilde{\mathbf{A}}$, thus ultimately leading to the correct role inference.

Comparison (and combination) with other baselines. The description of the example above was centered on highlighting the benefits of our proposed pipeline with respect to CloudCluster [72], a state-of-the-art method for extracting roles developed within the networking community. However, it is also important to establish our advantages with respect to more traditional (not domain specific) role inference methods in graphs. This includes simple algorithms based on Jaccard similarity [59] as well as more sophisticated structural embeddings such as SimRank [56] and GAS [49]. Notice that two nodes are deemed close to each other by Jaccard if they share similar neighbors. This essentially means that the similarity of two nodes is given by the similarity between their corresponding rows of $\mathbf{A}$. Consequently, Jaccard inherits the same limitations established for CloudCluster in the example above. Other structural embeddings [30, 49, 56] go beyond comparing similarities in the one-hop neighborhoods and can detect that, e.g., both pairs of green nodes play a similar role in the network. Nonetheless, these methods do not incorporate node features or partial labels, thus falling short of our proposed solution. In our evaluation, we show how our proposed pipeline improves upon all these baselines in real networks. Finally, notice that if the structural embeddings output by any of the existing methods are preferred to the reduced adjacency in Figure 5, we can simply replace $\tilde{\mathbf{A}}$ with the embeddings from any of these methods and implement our pipeline *talis qualis*. In this sense, we can think of our pipeline as a principled way of combining structural embeddings with domain knowledge through node features and partial labels, irrespective of the structural embedding of choice.

5 Low-Cost Telemetry and Analytics

In this section, we discuss the rationale behind our choice of telemetry and our experience in building a cost-effective and scalable communication graph generator for ZTS.

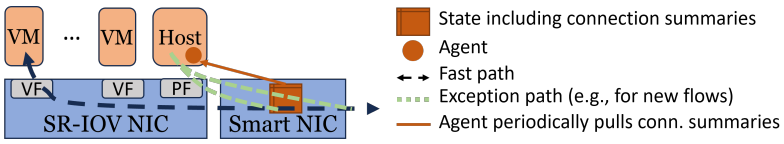


Figure 7: Illustrating how one can capture connection summaries with zero impact to VMs in a public cloud.

Time	Local		Remote		#Packets		#Bytes	
	IP	Port	IP	Port	Sent	Rcvd.	Sent	Rcvd.

Table 3: Schema of connection summaries.

	Azure	AWS	GCP
Name	NSG Flow Logs	VPC Flow Logs	
Agg. Intrvl	1 min	1 min	5s or higher
Content	As illustrated in Table 3		
Sampling	N/A	N/A	3% of Pkts, 50% of Flows
Price	about 0.5$/GB to collect		

Table 4: Details on available connection summaries at three large cloud providers.

5.1 Telemetry Source

As Section 3 mentions, the telemetry source for ZTS is network flow (or connection) summaries [10, 11, 17]. Tables 3 and 4 depict the data format and key details. We select this telemetry source because it (i) can be gathered with minimal disruption to customer workloads, (ii) is cost-effective, allowing for continuous collection, and (iii) is tamper-proof. Consequently, connection summaries are readily accessible to all customers across major cloud providers. Rather than developing a custom telemetry source, we opportunistically use the widely available connection summaries, which contributes to the practical viability of our system.

Cloud providers typically gather connection summaries using programmable NICs directly attached to all hosts in public clouds (Figure 7). These NICs perform various network functions [5, 6, 12, 35, 64, 75]. Connection summaries are stored in smartNIC memory, and an agent on the host periodically retrieves and forwards these summaries to a cloud store or service endpoint. Relative to capturing packet traces [21], logging in the kernel [62, 79] and sampling packets on switches [22], the method in Figure 7 has a few advantages. The size of the logs and memory usage are proportional to the number of concurrent flows. Since these cards already maintain per-flow state [23, 35], recording a few additional counters imposes minimal overhead. Importantly, the impact on customers is minimal; the performance interference from updating a few counters is negligible compared to the extensive network function processing occurring in network virtualization [23, 24, 43, 47]. Additionally, the same functionality can be implemented in the software stacks that handle network virtualization [43].

Collecting packet traces requires the network stack to copy packets, which is more costly compared to the method of generating flow summaries described in Figure 7, especially at NIC speeds reaching 400Gbps. Packet processing elements in switches typically do not have memory to spare to record per-flow state [15, 16], so they sample packets and summarize them at the control CPUs [22, 66]. Some cloud providers use a similar approach, sampling packets and flows to further reduce costs when generating connection summaries [11]. The cost-effective nature of connection summaries enables cloud providers to offer telemetry collection services at a price that makes continuous monitoring required for micro-segmentation feasible, typically $0.25 to $0.5 per GB of collected telemetry [4, 18, 20]. The total telemetry collection cost depends on the number of VMs and connection records generated per hour. In our study, this amounts to approximately $8 to $19 per VM per year. This telemetry is then securely stored to prevent tampering by customers, who do not have access to the host, ensuring availability even if VMs are breached. Customers incur additional storage costs for the telemetry, which can be minimized with an appropriate retention policy. These telemetry collection and storage costs are common to all micro-segmentation solutions and are in addition to the costs illustrated in Table 2. It is also possible that customers have telemetry collection and storage enabled for other monitoring use cases. Hence, we consider these costs orthogonal to our study's scope. Although alternatives like network stack hooks such as eBPF are beneficial, deploying agents on every VM in a public cloud is difficult due to the diverse OS types and the limited control providers have over guest OSes. Even if these agents are deployed, their telemetry can be compromised by malicious code within the VM, making them less reliable during breaches compared to the connection summaries used by our system.

5.2 System for Graph Generation

Our primary goal is to use systems that are available in most large public clouds today to build an analytics system that can analyze roughly 1000 VMs worth of telemetry (e.g., connection summaries at one minute granularity) using a handful of VMs worth of resources. This is roughly a 0.5% surcharge, and such a low COGS is crucial for extensive adoption.

The structure and volume of our telemetry source present challenges in designing a scalable, low-cost analytics system. Cloud providers organize telemetry into numerous small files (e.g., one JSON file per hour per VM) with connection summaries embedded as delimited strings within nested JSON structures. The structuring of the telemetry as many small files is a consequence of the design described in Figure 7, while the nested JSON keeps files compact in representing relationships between flows and various cloud resources. The

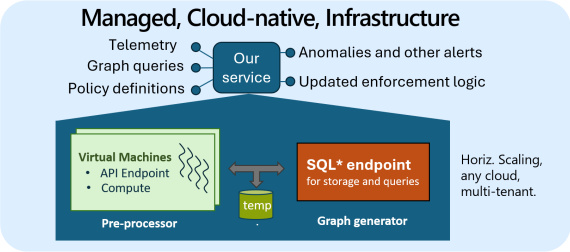


Figure 8: Our analytics system architected as a software-as-service (SaaS) that can adapt to load.

analytics system must efficiently process these small files, unroll nested JSON structures, and parse delimited strings to extract connection summaries. Additionally, it should integrate other information streams, such as IP Address Managers (IPAM) and Virtual Network (VNET) configurations, to enrich the summaries with domain knowledge for the role inference pipeline. The huge volume of connection summaries thus extracted (e.g., #records/min in Table 1) can result in very large communication graphs (e.g., an IP graph is roughly 10x to 100x the number of monitored VMs) requiring careful considerations to keep the processing time and cost low.

Given the telemetry structure, we faced a design choice between a stream processing system for consuming the large influx of small JSON files to generate graphs in real time, and a batch processing system that generates graphs from batched connection summaries. We chose the latter and separated the compute-intensive JSON processing from graph generation, resulting in two processing phases. This approach allows us to pipeline and parallelize the pre-processing stage, preparing batched connection summaries in an optimal format for large batch processing systems, resulting in low-latency processing.

Our system is architected as an always-on, cloud-native service as shown in Figure 8. We structure our system into two main components: (1) a pre-processor that leverages scalable compute resources to handle the large telemetry volume, (2) a graph generator based on a batch processing SQL system [9] to generate the communication graphs from the batched connection summaries. For low-latency graph generation, the batch processing system should be able to generate and complete an efficient query execution plan to process the connection summary batches into graphs. Naïvely, this is a group-by-aggregation query. That is, accumulate the byte, packet, and connection counts between pairs of nodes. The memory need is proportional to the number of node pairs in the graph which would be prohibitive for very large graphs. One mitigation we employ is to focus on the heavy hitters. That is, remote IPs and ephemeral ports that do not individually account for a sizable share of traffic are collapsed together. In fact, the graph sizes in Table 1 collapse IPs contributing less than 0.1% of bytes, packets or connections into one node in the IP-graph. Further, we optimize our query to use Common Table Expressions (CTEs) to eliminate intermediate materialization and to effectively use the system query planner. This approximation and the ability of the SQL system to handle large graphs by effectively using the available memory and disk resources allow us to construct communication graphs on subscriptions with 1000s of VMs in real time using just a few machines worth of resources.

We iterate over our system design and implementation over several developer months as shown in Figure 9, introducing several optimizations: (1) retiring our serverless PaaS system to a custom implementation of our pre-processor, (2) normalizing the tables and related query optimization to improve the batch processing performance and (3) optimizing resource management on the batch processing system. Such iterations enable us to keep pace with some of the large deployments in in Table 1 and keep cost low (e.g., 3 boxes for 5000 VMs).

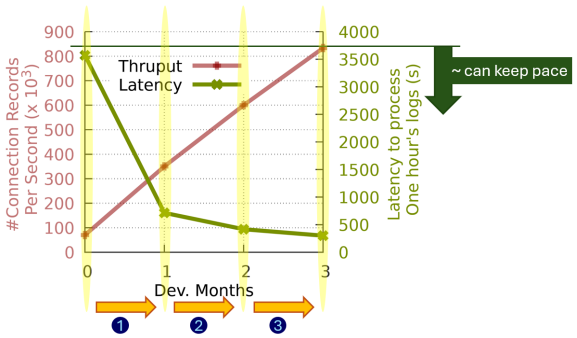


Figure 9: Iterative optimizations applied to our analytics system.

6 Evaluation

We seek to address the following key questions:

- Does the role inference algorithm developed by ZTS demonstrate significantly higher accuracy compared to existing algorithms in real-world applications?
- Does enhanced role inference contribute to the development of improved security policies?
- To what extent can ZTS's graph generator reduce costs compared to existing alternative systems?

6.1 Methodology

Datasets. To evaluate the effectiveness of our role inference algorithm, we curate a dataset comprising network telemetry from 11 real-world deployments within a large public cloud. This includes both first-party and third-party applications of varying sizes, patterns, and requirements. Due to infrastructure differences, the first six deployments utilize telemetry consisting of 5-tuples and flow summaries, while the latter five deployments use telemetry consisting of 5-tuples and process information, and the latter telemetry was also enabled at a larger scale. We conduct developer interviews to ascertain the role of each IP address in each deployment, which becomes the ground truth label in the communication graph. Table 5 summarizes the characteristics of the datasets.

	Telemetry	Graph Size		
		#nodes	#edges	#roles
Deployment A	Flow summary	200	500	14
Deployment B		200	5,000	28
Deployment C		500	9,000	21
Deployment D		200	3,000	28
Deployment E		100	100	12
Deployment F		100	2,000	17
Deployment G	Flow+Process summary	1,500	5,000	30
Deployment H		1,900	17,000	20
Deployment I		12,000	150,000	60
Deployment J		25,000	165,000	84
Deployment K		5,200	90,000	87

Table 5: Real-world deployments used in role inference evaluation: The graph sizes are rounded to preserve workload confidentiality.

Role-inference Baselines. We compare ZTS's role inference algorithm with four baselines that we discussed in §2.1 and §4.3: (1) Jaccard [59], (2) SimRank [56], (3) GAS [49], and (4) CloudCluster [72]. As for clustering algorithms, we use the hierarchical clustering algorithm described in CloudCluster [72] to infer roles for methods that output node embeddings (GAS, CloudCluster, and ours), and we use the Louvain community detection algorithm [37] for methods that output similarity scores (Jaccard and SimRank).

Hyperameters. We use the default hyperparameters from the corresponding papers for the baselines. For our algorithm, we set α in Equation 1 to 20 and τ to 0.05. Our autoencoder is a three-layer neural network with an encoded embedding size that is $\frac{2}{3}$ the size of the input embedding. We train our autoencoder for 1000 epochs with a learning rate of 0.002 and a weight decay of 0.001. To simulate existing asset tags and deployment of owner feedback, we randomly select 10% of the labels as h in our training pipeline. We find that our algorithm performs similarly well across a wide range of values for α, τ, and different autoencoder network architectures.

6.2 Role Inference Performance

Results Summary. Table 6 summarizes the Adjusted Rand Index (ARI) [53] between each role inference algorithm and ground-truth role labels. The ARI measures the quality of clustering and is widely used in role inference literature because it is adjusted for chance. We make three major observations.

First, ZTS's role inference algorithm significantly outperforms all baselines in all but one deployment. The average ARI of our algorithm is 0.77, while the average ARI for Jaccard, SimRank, GAS, and CloudCluster are only 0.33, 0.43, 0.39, and 0.34, respectively. This result demonstrates the effectiveness of ZTS's role inference algorithm in real-world deployments. The only deployment where ZTS does not outperform the best baseline is Deployment C, where our algorithm achieves an ARI of 0.96 compared to the best baseline's ARI of 0.97. This is because the ground truth of Deployment C is partially derived from the Jaccard method, which makes the Jaccard method perform particularly well in this deployment. Despite this, our algorithm still achieves a similar ARI in this deployment.

Second, the performance of state-of-the-art role inference algorithms varies drastically across different deployments. CloudCluster [72] generally performs well for smaller deployments (Deployments A to F) but lags behind other graph mining methods in larger deployments (Deployments G to K). This is partly because the adjacency matrix in larger deployments tends to be more sparse and noisy, and CloudCluster cannot find structural similarities effectively without diving deeper into the graph structures. The performance variation across deployments also underscores the criticality of incorporating domain-specific knowledge and developer insights into role inference processes, as varying deployment environments display distinct traits, and no single static algorithm can universally cater to all cases.

Third, all algorithms tend to perform better at smaller scales (A to F) than at larger scales (G to K). The average ARI of the baseline algorithms decreases from small to large deployments by 0.18, 0.19, 0.20, and 0.64 for Jaccard, SimRank, GAS, and CloudCluster, respectively. ZTS also sees performance degradation from small to large deployments, but at a more moderate 0.14, even with a much better starting point compared to the baselines. These results highlight the necessity of evaluating role-inference algorithms using large-scale deployments to better understand performance in practice. This also means that the performance gap between ZTS and other baseline algorithms increases with the deployment scale.

6.3 Role Inference and Security Policy

To evaluate the correlation between role inference performance and quality of the corresponding security policy, we conduct an assessment of *auto policy generation*. We collect telemetry data from the five larger deployments (Deployments G to K) over one day as the training set. We then infer the roles of each node daily using role inference algorithms. Cluster identities are aligned across different days based on node similarity. Each role is treated as a μsegment, allowing a pair of μsegments to communicate if such communication exists in the training set. We subsequently use telemetry data from the same deployment, four days after the training set, to evaluate the policy violation rate. This rate is defined as the fraction of edges that violate the security policy derived from the training set and the inferred roles. Table 7 summarizes the results.

As Table 7 shows, the security policy generated based on ZTS incurs a significantly lower policy violation rate than the policy generated using other baselines. This is mostly because, while the roles do not change often, the underlying communicating nodes can vary significantly across different days (e.g., talking to a different replica of the same role). If

	Jaccard	SimRank	GAS	CloudCluster	ZTS
Deployment A	0.47 ± 0.00	0.49 ± 0.01	0.50 ± 0.00	0.78 ± 0.00	$\mathbf{0.88 \pm 0.04}$
Deployment B	0.37 ± 0.00	0.33 ± 0.00	0.50 ± 0.09	0.81 ± 0.00	$\mathbf{0.85 \pm 0.03}$
Deployment C	$\mathbf{0.97 \pm 0.00}$	0.93 ± 0.00	0.89 ± 0.00	0.78 ± 0.00	0.96 ± 0.01
Deployment D	0.40 ± 0.00	0.06 ± 0.00	0.16 ± 0.06	0.50 ± 0.00	$\mathbf{0.63 \pm 0.05}$
Deployment E	0.55 ± 0.00	0.60 ± 0.00	0.33 ± 0.14	0.72 ± 0.00	$\mathbf{0.75 \pm 0.02}$
Deployment F	0.32 ± 0.00	0.10 ± 0.00	0.44 ± 0.06	0.73 ± 0.00	$\mathbf{0.90 \pm 0.03}$
Deployment G	0.44 ± 0.00	0.38 ± 0.02	0.41 ± 0.03	0.24 ± 0.00	$\mathbf{0.64 \pm 0.00}$
Deployment H	0.18 ± 0.00	0.14 ± 0.01	0.48 ± 0.01	0.18 ± 0.00	$\mathbf{0.61 \pm 0.03}$
Deployment I	0.41 ± 0.02	0.34 ± 0.01	0.25 ± 0.00	0.06 ± 0.00	$\mathbf{0.70 \pm 0.01}$
Deployment J	0.32 ± 0.00	0.14 ± 0.00	0.10 ± 0.01	-0.01 ± 0.00	$\mathbf{0.85 \pm 0.00}$
Deployment K	0.38 ± 0.00	0.13 ± 0.00	0.11 ± 0.02	-0.09 ± 0.00	$\mathbf{0.65 \pm 0.02}$

Table 6: The Adjusted Rand Index (ARI) between each role inference algorithm and ground-truth role labels. ARI is bounded below -0.5 for highly discordant clusterings and 1.0 for identical clusterings, being close to 0.0 for random labeling regardless of cluster or sample size. We report the mean and standard deviation across 10 runs for each experiment.

	Jaccard	SimRank	GAS	CloudCluster	ZTS
Deployment G	2.1%	11.1%	3.2%	38.4%	0.1%
Deployment H	3.5%	3.8%	2.3%	4.7%	0.2%
Deployment I	6.9%	5.2%	17.3%	6.1%	1.8%
Deployment J	6.5%	9.2%	8%	11.8%	0.7%
Deployment K	7.7%	8.3%	16%	20.7%	2.1%

Table 7: The policy violation rate of auto-generated security policies using role inference results and observed network behavior

the role inference algorithm does not assign the μsegment correctly, these changes will incur many policy violations. Note this evaluation is meant to assess how role inference algorithms can facilitate policy authoring, and the system is not designed to fully automate the process. Hence, a fraction of policy violations in a fully automated setting is expected.

6.4 Cost Effectiveness

To assess the cost of ZTS's graph generation system, we conducted a comparative analysis between our batch processing-based implementation and an alternative solution based on the open-source stream processing system, Apache Flink [38]. Our Flink implementation jointly handles the JSON file preprocessing and graph generation. Our choice of Flink for this alternative solution is based on two considerations. First, Flink is an enterprise-ready system with strong community support, making it a practical choice for our large-scale deployment. Second, we chose Flink over Spark [92], another popular open-source data processing system, because it has demonstrated superior performance in both streaming and batch processing with lower resource consumption [73], which aligns with our experiences with Spark in this context.

Our evaluation involved real-world telemetry data from regional deployments of two first-party services within a prominent public cloud. The details of these regional deployments are summarized in Table 8.

deployment	#nodes	data volume (#recs/hr)
Regional-Portal-Product (D1)	92	1.6M
Regional-K8s Paas (D2)	88	3.9M

Table 8: Characteristics of the regional deployments used in the evaluation of the analytics system. We use regional deployments to preserve workload confidentiality.

COGS. We run ZTS on an infrastructure consisting of a single 8-CPU VM with 32G of memory and 1 compute-unit of the batch processing system that costs in total $845/month. Similarly, Apache Flink runs on a single 64-CPU VM with 256G of memory, and this infrastructure costs in total $2406/month. We choose a larger VM for Apache Flink because its parallelism can benefit from more CPUs and memory so that it can achieve more competitive performance.

Dataset	ZTS	Flink
D1 (1hr)	78	344
D2 (1hr)	109	590
D1 (10hrs)	444	576
D2 (10hrs)	753	1195

Table 9: Processing completion times (in seconds) for the regional deployments. We report both single and multi-hour telemetry processing.

Completion Time. We compare the telemetry processing completion time for both 1-hour and 10-hour datasets across regional deployments. Our system consistently outperforms Apache Flink, as demonstrated in Table 9. Notably, Apache Flink can process a maximum of 5 hours' worth of data concurrently, resulting in its completion time for 10 hours of telemetry being merely double that of 1 hour. Conversely, ZTS delivers superior completion times in all instances while incurring only 35% of the COGS.

Scaling. We also evaluate the scaling behavior of both systems by scaling the telemetry volume by $10\times$ for the regional de-

ployments. We observe that our system outperforms Apache Flink even more significantly in this case, as shown in Table 10. ZTS is 7.5× faster than Apache Flink with only 35% of the COGS, or 21.5× more cost efficient. This is because Apache Flink is not designed to handle such large volumes of data, and its performance degrades significantly as the volume of data increases. In contrast, our system is designed to handle large volumes of data and can scale much better as the volume of data grows.

Dataset	ZTS	Flink
D2 (scaled by 10x)	765	5748

Table 10: Processing completion times (in seconds) for the scaled regional deployment.

It is worth noting that even when utilizing the SQL query optimizations discussed in Section 5.2, the Flink-based streaming system is not as cost-effective as our batch processing system for the scale of telemetry we consider. While Flink and Spark are designed for a wider range of use cases, our results highlight the importance of application-specific design and implementation to achieve significant cost savings, addressing a key obstacle to adopting micro-segmentation systems. We leave the exploration of our batch processing system's applicability to other use cases for future work.

7 Related work

Our key contributions are as follows:

- A novel role inference algorithm for nodes in communication graphs that incorporates domain knowledge and communication patterns.
- A low-cost system that constructs complete and dynamic communication graphs in cloud environments.
- Analysis over communication graphs from many large-scale real deployments.

Here we discuss related work in these areas.

Graph mining and analyses. Existing research on general temporal graph mining [29, 60, 74, 76] largely focuses on social networks, with limited studies addressing challenges in communication networks [57, 83]. Our work tackles novel issues specific to the networking context by analyzing network structures, communication patterns, and incorporating domain-specific knowledge to enhance the inference and application of security policies.

Network telemetry. Many systems analyze passively collected network-wide telemetry [42, 62, 65, 79], but have not achieved widespread deployment or consistent net positive revenue. Highly accurate and scalable sketching algorithms offer estimated answers to network telemetry queries [36, 67, 82, 91, 94], yet they lack generality, addressing only specific query types. Systems like PingMesh [48] and MALT [70] offer insights into network performance and topology but do not measure actual communication flow along data paths. Additionally, most research focuses on packet traces from a few collection points [32, 44, 54, 61, 69, 81], leaving the scalability and generalizability of these techniques to provide network-wide summaries uncertain.

Role inference in graphs and networks. Node embeddings have been utilized to encapsulate structural similarities within graphs [58]. These structural embeddings can subsequently be aggregated through clustering to facilitate role discovery or inference. The majority of existing methods adhere to a two-step procedure, initially discerning structural attributes associated with each node and subsequently transposing these attributes into an embedding space using various techniques [58]. These techniques encompass low-rank matrix factorization [50, 51], random walk-inspired methods [27, 77], and approaches based on (graph) neural networks [49, 85]. Within the domain of networking, the closest work CloudCluster [72], is also based on structural similarities. Section 4 and Section 6.2 discuss the shortcomings of these approaches and evaluate these predominant approaches. In contrast to these methods, our approach allows for the flexible integration of domain-specific knowledge through meticulously crafted node features coupled with a contrastive loss function. Furthermore, it is worth noting that should any existing structural embeddings be deemed more appropriate, our algorithm is designed to integrate them seamlessly into our processing pipeline.

Other studies have leveraged network traffic to infer attributes of certain segments within the communication graph. For instance, Xu et al. [90] classify internet hosts according to traffic similarities, utilizing this classification to identify malicious activities. In the realm of cloud services, patterns of network traffic have been instrumental in pinpointing network nodes or flows that may be compromised [31, 46, 52], as well as selecting suitable candidates for migration [40].

8 Conclusion

In this work, we show that realizing the vision of micro-segmentation at a large scale remains a difficult challenge. We present an end-to-end system that consists of an effective role inference algorithm for network communication graphs and a unique system implementation that drastically reduces the cost of generating communication graphs for network behavior understanding and monitoring. Using 11 large-scale real deployments, we demonstrate the system's capability to more accurately infer roles and generate communication graphs with minimal expense. We hope the findings and insights from our work will stimulate further research to ease the burden of securing public clouds with micro-segmentation.

Ethics: The use of production traces in this paper was governed by an institutional privacy review.

Acknowledgments

We extend our gratitude to our shepherd, Amin Vahdat, and the anonymous reviewers for providing invaluable and constructive feedback. Our thanks also go to our research, engineering and product collaborators, including Anatoliy Panasyuk, Ashish Bhargava, Eliran Azulai, Eli Schwartz, Ido Frizler, Trevor Eberl, Jamie Lee, Kiran Muthabatulla, Kris Zentner, Mariana Alanis Tamez, Prateesh Goyal, Priyatham Allala, Sachin Ashok, and Vahab Jabrayilov for their contributions to production traces, project planning, deployment automation, system optimization, and testing.

References

[1] 2017 Equifax data breach. https://en.wikipedia.org/wiki/2017_Equifax_data_breach, Retrieved on 2023-04.

[2] 2020 United States federal government data breach. https://en.wikipedia.org/wiki/2020_United_States_federal_government_data_breach, Retrieved on 2023-04.

[3] Akamai guardicore segmentation plans and pricing. https://azuremarketplace.microsoft.com/en-us/marketplace/apps/akamai-technologies.akamai_segmentation?tab=PlansAndPrice, Retrieved on 2024-09.

[4] Amazon CloudWatch Pricing - Amazon Web Service (AWS). https://aws.amazon.com/cloudwatch/pricing/.

[5] AWS: Data Transfer Costs for Common Architectures. https://go.aws/3cg5J3O.

[6] Azure: Bandwidth Pricing. https://bit.ly/3Cou81Z.

[7] Azure linux virtual machines pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/, Retrieved on 2024-09.

[8] Azure network security groups. https://learn.microsoft.com/en-us/azure/virtual-network/network-security-groups-overview.

[9] Azure synapse analytics. https://learn.microsoft.com/en-us/azure/synapse-analytics/.

[10] Flow logging for network security groups. https://learn.microsoft.com/en-us/azure/network-watcher/network-watcher-nsg-flow-logging-overview.

[11] GCP: VPC flow logs. https://cloud.google.com/vpc/docs/flow-logs.

[12] Google Cloud: Bandwidth Pricing. https://bit.ly/3Cw83i9.

[13] Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[14] Illumio core plans and pricing. https://azuremarketplace.microsoft.com/en-us/marketplace/apps/illumioinc1629822633689.illumio_core?tab=PlansAndPrice, Retrieved on 2024-09.

[15] Intel tofino. https://intel.ly/3wxWT8w.

[16] Intel tofino 2. https://intel.ly/3QTeD6F.

[17] Logging IP traffic using VPC flow logs. https://docs.aws.amazon.com/vpc/latest/userguide/flow-logs.html.

[18] Network pricing | Google Cloud. https://cloud.google.com/vpc/network-pricing?hl=en#network-telemetry.

[19] A new walmart 'cloud factory' will accelerate digital innovation, boost business efficiency. https://shorturl.at/amwHI.

[20] Pricing - Network Watcher | Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/network-watcher/.

[21] tcpdump. http://ee.lbl.gov/tcpdump.tar.Z.

[22] Using netflow filtering or sampling to select the network traffic to track. https://rb.gy/83mcu.

[23] VFP: A virtual switch platform for host SDN in the public cloud. In *NSDI*, 2017.

[24] Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, 2018.

[25] Microsegmentation - global strategic business report. https://www.researchandmarkets.com/report/microsegmentation, 2023.

[26] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient graphlet counting for large networks. In *2015 IEEE International Conference on Data Mining (ICDM)*, pages 1–10. IEEE Computer Society, 2015.

[27] Nesreen K. Ahmed, Ryan A. Rossi, John Boaz Lee, Theodore L. Willke, Rong Zhou, Xiangnan Kong, and Hoda Eldardiry. Role-based graph embeddings. *IEEE Transactions on Knowledge and Data Engineering*, 34(5):2401–2415, 2022.

[28] Akamai. Akamai Guardicore Segmentation. https://www.akamai.com/products/akamai-guardicore-segmentation.

[29] Alessia Antelmi, Gennaro Cordasco, Mirko Polato, Vittorio Scarano, Carmine Spagnuolo, and Dingqi Yang. A survey on hypergraph representation learning. *ACM Computing Surveys*, 56(1):1–38, 2023.

[30] Ioannis Antonellis, Hector Garcia Molina, and Chi Chao Chang. Simrank++: Query rewriting through link analysis of the click graph. In *VLDB Endowment*, 2008.

[31] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack Stokes, Geoff Outhred, and Lechao Diwu. PrivateEye: Scalable and Privacy-Preserving compromise detection in the cloud. In *NSDI*, 2020.

[32] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David Maltz, and Ming Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM*, 2007.

[33] Janet Bailey and Bradley Jensen. Walmart and azure. https://shorturl.at/vMTY0.

[34] Hitesh Ballani, Yatin Chawathe, Sylvia Ratnasamy, Timothy Roscoe, and Scott Shenker. Off by default! In *HotNets*, 2005.

[35] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *NSDI*, 2023.

[36] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 127–140, 2017.

[37] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Statistical Mechanics: Theory and Experiment*, 2008.

[38] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, et al. Apache Flink: Stream and batch processing in a single engine. In *ICDE*, 2015.

[39] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. *ACM SIGCOMM Computer Communication Review*, 37(4):1, October 2007.

[40] Marco Cello, Kang Xi, Jonathan H. Chao, and Mario Marchese. Traffic-aware clustering and vm migration in distributed data center. In *ACM SIGCOMM Workshop on Distributed Cloud Computing*, page 41–42, 2014.

[41] Cisco. Cisco Tetration. https://www.cisco.com/c/en_sg/products/data-center-analytics/tetration-analytics/index.html.

[42] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: High performance network monitoring with an sql interface. In *SIGMOD*, 2002.

[43] Michael Dalton et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*, 2018.

[44] Cristian Estan, Stefan Savage, and George Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *SIGCOMM*, 2003.

[45] Bob Evans. Walmart cio: We picked microsoft for huge cloud deal to accelerate digital transformation. https://shorturl.at/aABI3.

[46] Chuanpu Fu, Qi Li, and Ke Xu. Detecting unknown encrypted malicious traffic in real time via flow interaction graph analysis. In *30th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2023.

[47] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *SIGCOMM*, 2014.

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

[49] Xuan Guo, Wang Zhang, Wenjun Wang, Yang Yu, Yinghui Wang, and Pengfei Jiao. Role-oriented graph auto-encoder guided by structural information. In *Database Systems for Advanced Applications*, pages 466–481, 2020.

[50] Mark Heimann, Haoming Shen, Tara Safavi, and Danai Koutra. REGAL: Representation learning-based graph alignment. In *ACM International Conference on Information and Knowledge Management*, page 117–126, 2018.

[51] Keith Henderson, Brian Gallagher, Tina Eliassi-Rad, Hanghang Tong, Sugato Basu, Leman Akoglu, Danai Koutra, Christos Faloutsos, and Lei Li. Rolx: Structural role extraction & mining in large graphs. In *KDD*, 2012.

[52] Kevin Hsieh, Mike Wong, Santiago Segarra, Sathiya Kumaran Mani, Trevor Eberl, Anatoliy Panasyuk, Ravi Netravali, Ranveer Chandra, and Srikanth Kandula. NetVigil: Robust and low-cost anomaly detection for east-west data center security. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[53] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of classification*, 2:193–218, 1985.

[54] Marios Iliofotou, Prashant Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *IMC*, 2007.

[55] Illumio. Zero Trust: the security paradigm for the modern organization. https://www.illumio.com/solutions/zero-trust.

[56] Glen Jeh and Jennifer Widom. Simrank: A measure of structural-context similarity. In *KDD*, 2002.

[57] Weiwei Jiang. Graph-based deep learning for communication networks: A survey. *Computer Communications*, 185:40–54, 2022.

[58] Pengfei Jiao, Xuan Guo, Ting Pan, Wang Zhang, Yulong Pei, and Lin Pan. A survey on role-oriented network embedding. *IEEE Transactions on Big Data*, 8(4):933–952, 2021.

[59] Ruoming Jin, Victor E. Lee, and Hui Hong. Axiomatic ranking of network role similarity. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 922–930. ACM, 2011.

[60] Wei Ju, Zheng Fang, Yiyang Gu, Zequn Liu, Qingqing Long, Ziyue Qiao, Yifang Qin, Jianhao Shen, Fang Sun, Zhiping Xiao, et al. A comprehensive survey on deep graph representation learning. *Neural Networks*, page 106207, 2024.

[61] Srikanth Kandula, Ranveer Chandra, and Dina Katabi. What's Going On? Learning Communication Rules in Edge Networks. In *SIGCOMM*, 2008.

[62] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *IMC*, 2009.

[63] John Kindervag, Stephanie Balaouras, and Lindsey Coit. Build security into your network's DNA: The zero trust network architecture. *Forrester Research Inc*, 27, 2010.

[64] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[65] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. *SIGCOMM CCR*, 2005.

[66] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.

[67] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[68] Mandiant. UNC5537 targets Snowflake customer instances for data theft and extortion. https://cloud.google.com/blog/topics/threat-intelligence/unc5537-snowflake-data-theft-extortion, Retrieved on 2024-06.

[69] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *NDSS*, 2018.

[70] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *NSDI*, 2020.

[71] MSRC. Microsoft actions following attack by nation state actor midnight blizzard. https://msrc.microsoft.com/blog/2024/01/microsoft-actions-following-attack-by-nation-state-actor-midnight-blizzard/, Retrieved on 2024-01.

[72] Weiwu Pang, Sourav Panda, Muhammad Jehangir Amjad, Christophe Diot, and Ramesh Govindan. Cloudcluster: Unearthing the functional structure of a cloud

service. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, (NSDI)*, pages 1213–1230. USENIX Association, 2022.

[73] Shelan Perera, Ashansa Perera, and Kamal Hakimzadeh. Reproducible experiments for comparing apache flink and apache spark on public clouds, 2016.

[74] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *KDD*, 2014.

[75] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vSwitch. In *NSDI*, 2015.

[76] Ahmad Rawashdeh and Anca Ralescu. Similarity measure for social networks – a brief survey. volume 1353, 2015.

[77] Leonardo F.R. Ribeiro, Pedro H.P. Saverese, and Daniel R. Figueiredo. struc2vec: Learning node representations from structural identity. In *KDD*. ACM, 2017.

[78] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. Technical report, National Institute of Standards and Technology, 2020.

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

[80] Burr Settles. Active learning literature survey. 2009.

[81] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *ACM SIGCOMM 2003*.

[82] Haifeng Sun, Qun Huang, Jinbo Sun, Wei Wang, Jiaheng Li, Fuliang Li, Yungang Bao, Xin Yao, and Gong Zhang. {AutoSketch}: Automatic {Sketch-Oriented} compiler for query-driven network telemetry. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1551–1572, 2024.

[83] Prohim Tam, Inseok Song, Seungwoo Kang, Seyha Ros, and Seokhoon Kim. Graph neural networks for intelligent modelling in network management and orchestration: a survey on communications. *Electronics*, 11(20):3371, 2022.

[84] Charalampos E. Tsourakakis, Jakub Pachocki, and Michael Mitzenmacher. Scalable motif-aware graph clustering. In *Proceedings of the 26th International Conference on World Wide Web (WWW)*, pages 1451–1460. ACM, 2017.

[85] Ke Tu, Peng Cui, Xiao Wang, Philip S. Yu, and Wenwu Zhu. Deep recursive network embedding with regular equivalence. In *KDD*, 2018.

[86] Verizon. 2023 data breach investigations report. https://www.verizon.com/business/resources/reports/dbir/.

[87] Verizon. Data breach investigations report: 2008 – 2022. https://www.verizon.com/business/resources/reports/2022/dbir/2022-data-breach-investigations-report-dbir.pdf.

[88] VMWare. VMware NSX. https://www.vmware.com/products/nsx.html.

[89] VMWare. Data center micro-segmentation: A software defined data center approach for a "zero trust" security strategy. Technical report, 2014. https://blogs.vmware.com/networkvirtualization/files/2014/06/VMware-SDDC-Micro-Segmentation-White-Paper.pdf, Retrieved on 2024-09.

[90] Kuai Xu, Feng Wang, and Lin Gu. Network-aware behavior clustering of internet end hosts. In *2011 proceedings ieee infocom*, pages 2078–2086. IEEE, 2011.

[91] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[92] M. Zaharia et al. Spark: Cluster Computing with Working Sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, 2010.

[93] Wang Zhang, Xuan Guo, Wenjun Wang, Qiang Tian, Lin Pan, and Pengfei Jiao. Role-based network embedding via structural features reconstruction with degree-regularized constraint. *Knowl. Based Syst.*, 218:106872, 2021.

[94] Fuheng Zhao, Punnal Ismail Khan, Divyakant Agrawal, Amr El Abbadi, Arpit Gupta, and Zaoxing Liu. Panakos: Chasing the tails for multidimensional data streams. *Proceedings of the VLDB Endowment*, 16(6):1291–1304, 2023.

Mitigating Scalability Walls of RDMA-based Container Networks

Wei Liu[1,2], Kun Qian[2], Zhenhua Li[1]✉, Feng Qian[3], Tianyin Xu[4], Yunhao Liu[1]
Yu Guan[2], Shuhong Zhu[2], Hongfei Xu[2], Lanlan Xi[2], Chao Qin[2], Ennan Zhai[2]
[1]*Tsinghua University* [2]*Alibaba Cloud* [3]*University of Southern California* [4]*UIUC*

Abstract

As a state-of-the-art technique, RDMA-offloaded container networks (RCNs) can provide high-performance data communications among containers. Nevertheless, this seems to be subject to the RCN scale—when there are millions of containers simultaneously running in a data center, the performance decreases sharply and unexpectedly. In particular, we observe that most performance issues are related to RDMA NICs (RNICs), whose design and implementation defects might constitute the "scalability wall" of the RCN. To validate the conjecture, however, we are challenged by the limited visibility into the internals of today's RNICs. To address the dilemma, a more pragmatic approach is to *infer* the most likely causes of the performance issues according to the common abstractions of an RNIC's components and functionalities.

Specifically, we conduct *combinatorial causal testing* to efficiently reason about an RNIC's architecture model, effectively approximate its performance model, and thereby proactively optimize the NF (network function) offloading schedule. We embody the design into a practical system dubbed ScalaCN. Evaluation on production workloads shows that the end-to-end network bandwidth increases by 1.4× and the packet forwarding latency decreases by 31%, after resolving 82% of the causes inferred by ScalaCN. We report the performance issues of RNICs and the most likely causes to relevant vendors, all of which have been encouragingly confirmed; we are now closely working with the vendors to fix them.

1 Introduction

Containers have become a pivotal technology in cloud computing and serverless computing [34, 35, 56, 70], which allow developers and service operators to deploy applications or functions onto the cloud rapidly and conveniently. Different from a heavyweight and full-fledged virtual machine, a container is merely a lightweight and standalone software bundle that includes all the code and dependencies needed by a specific application [30, 43]. The portability, lightweight, and isolation characteristics make containers an ideal choice for running a variety of tasks in the cloud, such as training foundation models [27, 31, 64] and building microservices [51, 53].

Container networks enable containers to communicate with each other seamlessly based on their hosts. To achieve high-performance data communications, almost all of today's container service providers (CSPs) utilize RDMA NICs (RNICs) to offload data traffic and network functions (NFs) from the host OS to dedicated hardware (RNICs) [35, 57]. The resulting RDMA-offloaded container network (RCN) is reported to have at least three times of performance gain compared to the traditional network stack inside the host OS [35, 39, 40], paving an efficient and generic network runtime for cloud applications and stateful/statelss functions.

Nevertheless, as a mainstream CSP, we observe that the performance gain of the RCN seems to be subject to its scale. During our two-year (03/2021 – 03/2023) operations on a production RCN (which involves ∼8K hosts equipped with ∼40K RNICs, serving 0.5M active containers simultaneously on average and ∼1M containers at peak hours), we notice that the end-to-end performance in terms of bandwidth and latency often decreases sharply and unexpectedly. For instance, when the number of active containers grows from 0.4M to 0.8M, the end-to-end bandwidth can decrease by 87%, and the packet forwarding latency can increase by 34×. In other words, there are "scalability walls" in the production RCN, which is rarely understood as existing studies mostly examine the RCN in small-scale and controlled settings [35, 58].

From Phenomena to Symptoms. To deeper understand the problem, we built a continuous monitoring infrastructure on top of our production RCN. It gathers cross-layer runtime data on each host, such as container resource consumptions, virtual switch flow tables, kernel logs, and RNIC statistics, during our monitoring period from 04/15/2023 to 04/15/2024.

The collected data reveal that the scalability wall of the RCN mainly (94%) manifests eight symptoms as listed in Table 1. They occur when an RNIC's aggregated throughput or latency is not at the expected level as in their official specifications (referred to as a *performance issue*). These symptoms come from different RCN layers such as the virtual switch (i.e., Open vSwitch), the kernel driver, and hardware, but are surprisingly all related to RNICs. In total, they have caused 13,396 performance issues during our monitoring period.

- Symptom S1 manifests on the virtual switch when there are 7K+ container flows running over an RNIC. The victim flows repetitively fall back to software processing, thus incurring significant performance degradation (>18×).
- The interactions between the RNIC's driver and hardware can cause kernel stagnation (S2) or crash (S3), especially when the driver creates or deletes flows, thus blocking the execution of all containers on the victim host.
- The remaining five symptoms occur at the layer of RNIC

Table 1: RNIC-related symptoms of the "scalability wall" in our production RCN, as well as their most likely causes.

No.	Symptom	Layer	Ratio	Most Likely Cause (C1 – C8)
S1	Repetitive flow re-offloading	Virtual Switch	17.1%	Flow entries in the RNIC are deleted although they are not aged.
S2	Kernel stagnation	RNIC driver	5.9%	The driver cannot handle the timeout of the RNIC's executing an operation.
S3	Kernel crash on new flows	RNIC driver	5.2%	The driver frees a null pointer when the RNIC fails to create a flow entry.
S4	Slow flow state maintenance	RNIC hardware	11.4%	Flow deletion in the RNIC takes much longer (9×) time than expected.
S5	Intermittent software forwarding	RNIC hardware	15.3%	Flow counters are not updated timely. The virtual switch reads a "dirty" value.
S6	Poor performance of specific flows	RNIC hardware	29.9%	Flow entries with different masks are queried sequentially in the RNIC.
S7	PCIe link down when unbinding VFs	RNIC hardware	8.4%	Race condition emerges when the RNIC cleans up allocated resources.
S8	RNIC unresponsiveness	RNIC hardware	6.8%	VXLAN encapsulation contexts exceed the RNIC's buffer capacity.

hardware. For example (S6), a specific flow's performance can be persistently poor when ∼10K flows are offloaded to the RNIC hardware (we did not discover any anomalies in the software stack). Even if the workload drops off greatly (e.g., from ∼10K to 600 flows) afterwards, the performance cannot recover in a short time.

From Symptoms to Causes. Given the various symptoms, we wish to figure out their root causes, so that we can take measures to proactively optimize the performance or at least prevent further degradation. Unfortunately, we encounter a key challenge that we have very limited visibility into the internals of today's commodity RNICs [32, 36]. The internal design of RNICs is mostly close-sourced and hardly mentioned in their specifications. Resorting to the vendors also turns out to be ineffective in most cases, since they have neither encountered nor understood such "elusive" symptoms. After all, these symptoms are only manifested in large-scale deployments that involve extremely complicated workloads/environments and push the RNIC's capability to its limit, which are not easy for the vendors to reproduce and analyze. Furthermore, we are not allowed to share real-world workloads with the vendors, making their troubleshooting difficult, if not impossible.

To address this, a more pragmatic approach is to *infer* the most likely causes according to common abstractions of an RNIC's components and functionalities. An RNIC on a host provides RDMA verb functionalities (e.g., `read` and `write`) to each hosted container, which allow the container to operate on the resources in the RNIC's conceptual components (e.g., send/receive queues). Besides, today's RNICs in data centers all implement *match-action* based embedded switch (or *eSwitch* for short) components to support hardware-accelerated packet switching [38, 41, 55], which are widely used for efficient container network virtualization. Our approach fundamentally differs from the existing blackbox testing techniques [67] or whitebox analyses [54], as it leverages the domain knowledge of RNICs' common architecture to construct a highly likely RNIC model for actual testing and optimization, and thus is more of a "greybox" approach.

With the common abstractions of an RNIC, however, we are still confronted with a prohibitively enormous search space, stemming from the components' possible combinations and the functionalities' possible configurations. For example, highly configurable eSwitch tables can have $O(k^n)$ possible matchers, where n is the number of tables and k is the average number of table entries. We conduct *combinatorial causal testing* to overcome the difficulty in a principled way. First, we leverage topological restrictions across components to efficiently construct valid component combinations (i.e., the architecture model of the RNIC). This is achieved by inspecting components' dependencies to filter out the combinations that lead to packet loops or unreachability, which reduces the magnitude of combinations to a quadratic polynomial level.

We then test the RNIC's functionalities with real and synthetic workloads on each valid architecture model. Once a symptom occurs, we perform a *local sensitivity analysis* [19] to deduce the causal relations between the RNIC's input (workload) and manifested performance. Specifically, we strategically tune the value of each dimension (e.g., the number of table entries) in the workload to test whether the symptom can be eliminated or alleviated. If so, the component(s) related to this dimension are a likely cause of the symptom, which might lie in a critical path of the RNIC's packet processing pipeline. Next, we fine-check each concerned component in the critical path by means of *permutation removal* [15], which eliminates the component whose removal would not influence the performance. In this way, we sort out the critical path(s), pinpoint the most likely causes, and effectively approximate the RNIC's performance model for a valid architecture model. Eventually, we choose the architecture model that best matches the actual performance as the final reference.

Optimization, Evaluation, and Validation. Based on the above efforts, we are able to proactively optimize the NF offloading schedule for every RNIC, by reorganizing the offloaded flows to minimize the estimated processing time over the critical path for each flow. Also, we avoid triggering the inferred design/implementation defects of the RNIC by transforming the risky functionality inputs to their equivalents that are unlikely or less likely to cause performance issues.

We implement the whole design into a practical system dubbed ScalaCN, and conduct extensive experiments based on real-world production workloads with six different kinds of RNIC devices (NVIDIA ConnectX-4, -5, -6, -7, BlueField-3, and Intel E810). On average, ScalaCN improves the end-to-end bandwidth by 1.4× and reduces the latency by 31%, after resolving 82% of the inferred causes of performance issues (the remaining 18% are due to hardware limits). Its major overhead comes from continuous monitoring and optimiza-

tion operations, with <5% usage on a single CPU core.

We have reported all performance issues and their most likely causes to relevant vendors, all of which have been encouragingly confirmed. In detail, the vendors have released patches on the driver and firmware to fix the root causes C1, C2, C3, and C5. For the remainder (C4, C6, C7, and C8), we are now closely collaborating with the vendors to fix them as they involve the intricate co-tuning of software and hardware.

Contribution. This work makes the following contributions.

- We conduct the first study to uncover the scalability wall in a large-scale RCN, and pinpoint the culprit to be RNICs.
- We conduct combinatorial causal testing based on RNICs' common abstractions, to efficiently approximate their internals and infer the root causes of performance issues.
- We devise an effective method to accommodate RNICs to RCN scaling. Evaluation on real-world workloads and the feedback from vendors confirm its efficacy. We are now gradually deploying ScalaCN over the production RCN.

The code and data involved in this work have been released in part at https://scala-cn.github.io.

2 Background

Container networks enable transparent data communications among containers, and they are currently realized in three major approaches [35, 70] as listed in Table 2. Different approaches provide different levels of connectivity among containers and different levels of isolation with the host network. Today's mainstream container services widely adopt the overlay (or overlay-like) mode to organize container networks [2, 4, 7, 13, 26], since this mode enables accessibility across hosts while preserving isolation.

Figure 1 shows the typical architecture of an overlay-based container network. Two hosts (A and B) are connected via physical switches/routers in the underlay network. Each overlay container on a host has its own IP address within the same VXLAN [42] subnet. For example, a container on Host A has the IP address 172.16.122.1 within the VXLAN subnet 172.16.122.0/24. When two containers communicate with each other, the software virtual switch on the host OS will determine to which port the packet should be forwarded, and perform VXLAN encapsulation for packets (if needed). For container communications across hosts, the encapsulated packets will go through the hardware NIC and the underlay network.

Hardware Offloading with RDMA NICs. In practice, manipulating and forwarding container packets with software switches is resource-consuming. This is because container applications (e.g., large model training) often generate a large volume of network traffic which is processed by the kernel [3, 6] and the user-space software switch, incurring significant CPU overhead and processing latency [22, 39].

To alleviate the packet processing overhead on the host OS, today's mainstream container service providers (CSPs) take

Table 2: Different modes of container networking, where "isolation" refers to *network* isolation from the host OS.

Mode	Mechanism	Connectivity	Isolation
Bridge	Virtual bridge	Host only	In-bridge
Host	Namespace sharing	Cross-host	No isolation
Overlay	VXLAN	Cross-host	In-overlay

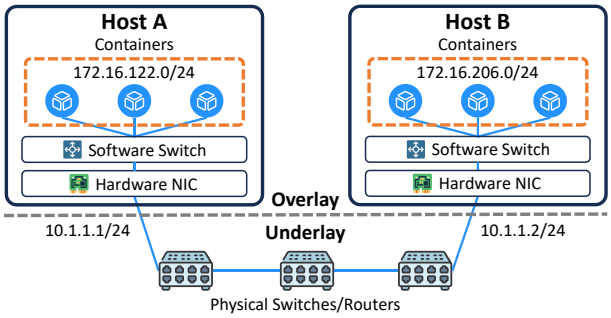


Figure 1: Example of overlay-based container networks.

advantage of the hardware offloading functionality [38, 47] with the single-root IO virtualization (SR-IOV) technique of RDMA NICs (RNICs) [28, 63]. This brings two significant performance benefits. First, all the containers can communicate with each other through the RDMA protocol, without needing the intervention of the OS. Second, the CPU-intensive tasks such as packet forwarding and VXLAN encapsulation also bypass the software stack, i.e., they are all moved from CPUs to RNICs for more efficient processing.

The resulting RDMA-offloaded container network (RCN) significantly improves the network performance. A representative example is illustrated in Figure 2. When the traffic is delivered using TCP, the average end-to-end bandwidth is 21.17 Gbps and the average packet forwarding latency is 87.31 us. When VXLAN encapsulation and packet switching are offloaded to an NVIDIA ConnectX-6 RNIC, the bandwidth grows to 50.31 Gbps, and the latency drops to 54.51 us. If we further use the RDMA protocol to transmit packets, the bandwidth remarkably increases to ∼180 Gbps and the latency decreases to merely 2.7 us.

3 Motivation

Despite the merits of RNIC offloading, an RCN's performance seems to be subject to its scale—during our one-year operations, we noticed that once the network workload hits a "scalability wall", a variety of performance issues occurred and severely hurt the QoS (Quality of Service).

3.1 Continuous Monitoring Infrastructure

To deeply understand the scalability wall, we built a continuous monitoring infrastructure on top of our large-scale production RCN, which carries ∼1,500 PB of traffic for ∼0.5 million containers every day during our monitoring period (from Apr. 15th, 2023 to Apr. 15th, 2024).

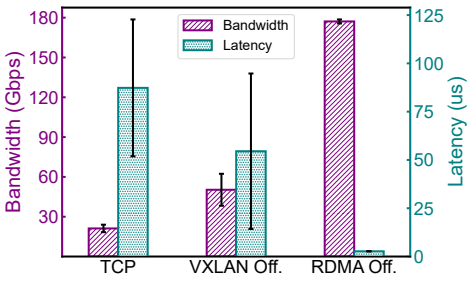


Figure 2: Performance comparison with and without hardware offloading (Off.) in a container network.

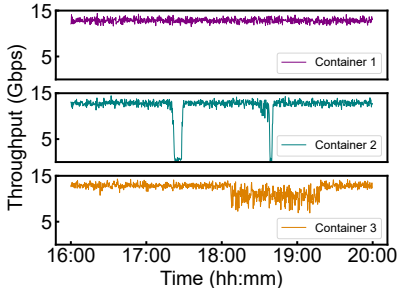


Figure 3: Performance issues caused by S1, where the virtual switch repetitively re-offload flows.

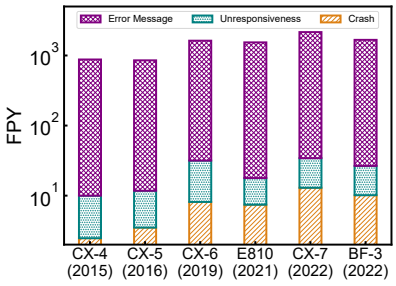


Figure 4: Kernel instability in terms of failures per year (FPYs). Here the RNICs are sorted by release date.

All containers in our RCN are Docker [9] containers. We use Kubernetes [10] to orchestrate Docker containers together with the container network interface (CNI) [8] plugin that we developed in-house. Each host in our RCN is equipped with one to eight RNICs, which are NVIDIA ConnectX-4 (CX-4), ConnectX-5 (CX-5), ConnectX-6 (CX-6), ConnectX-7 (CX-7), BlueField-3 (BF-3), or Intel E810 (E810). We balance the load of these RNICs when binding them to containers. We deploy Open vSwitch (OVS) [46] on each host as the software virtual switch to handle packet forwarding. When containers initiate a flow, the OVS will use the `TC` [5] utility to offload packet processing tasks to the RNIC.

Our continuous monitoring infrastructure gathers basic performance data across different layers on each host as follows, so that we can be aware of the RCN status in time. These data are transformed into concrete states of network components, and indexed in our log services for aggregation and analysis.

- *RNIC and Kernel Statistics.* We collect packet sending and receiving rates as well as packet drops of the RNIC through Linux `sysfs`. We also collect logical flow entries recorded by the kernel using `netfilter` and `conntrack` to track the states of all RDMA connections.
- *OVS Offloading Status.* The OVS determines what packet manipulations should be applied to an RCN flow and whether it needs to be offloaded to the RNIC. To monitor these events, we gather OVS information including table hits, misses, and losses of flow lookups [46].
- *CNI Events.* We collect crucial events on resource allocations triggered by the CNI plugin. We monitor the creation of containers, as well as the allocations of VXLAN subnets, IP addresses, and VFs (virtual functions that containers are one-to-one bound to) of RNICs.

3.2 Symptoms of Performance Issues

Our collected data reveal 14,251 performance issues regarding the scalability wall of the production RCN during our monitoring period. The eight major symptoms are listed in Table 1, accounting for 94% of the performance issues. Although these symptoms come from different layers of the RCN, such as the virtual switch, the kernel driver, and the hardware, they are all related to the RNICs which were supposed to bring high-performance data communications among containers.

Virtual Switch. When the number of flows offloaded to an RNIC increases to a threshold, e.g., 7K+ over a CX-6 RNIC, the virtual switch would repetitively re-offload the flows (Symptom S1). This accounts for 17.1% of RNIC-related performance issues. When this occurs, the end-to-end performance on the affected containers greatly fluctuates.

Figure 3 shows our production traces regarding S1. The three containers are used for training the same AI model with the same configurations (e.g., QoS and VF quotas). The throughput of Container 1 keeps stable at around 15 Gbps, which is the expected performance. However, Container 2 and Container 3 both suffer from performance degradation and present two different patterns of performance issues. The throughput of Container 2 periodically drops to $<$1 Gbps (which is nearly 18$\times$ worse) and then recovers to $\sim$15 Gbps. The throughput of Container 3 becomes unstable, sometimes fluctuating between 5 Gbps and 10 Gbps for an hour. These issues significantly affect the training process of AI models and oftentimes lead to users' complaints.

By analyzing the logs of the OVS, we find that the OVS repetitively re-offloads the flows for Containers 2 and 3 due to constant lookup misses in the RNIC's flow tables, as if the RNIC is "refusing" to handle them under the high workload.

RNIC Driver. We also find that in our production RCN, the RNIC's driver often makes the kernel unstable when it interacts with the RNIC's hardware, which manifests as S2 and S3 in Table 1. When handling a large number of flows offloaded in a short time (e.g., 1K per second), the kernel can become stagnated (S2) after the driver sends certain operation codes to the RNIC. Through detailed timing on the driver's call stack in the kernel, we observe that the driver seems to be always waiting for the RNIC's response, which however cannot be received in time under the high workload. Worse still, the kernel can even crash (S3) when the driver fails to offload the flows to the RNIC. Even though these symptoms occur less frequently (11.1% in total) than the symptom oc-

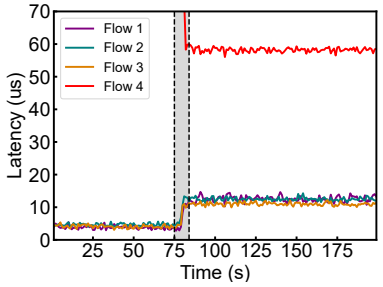
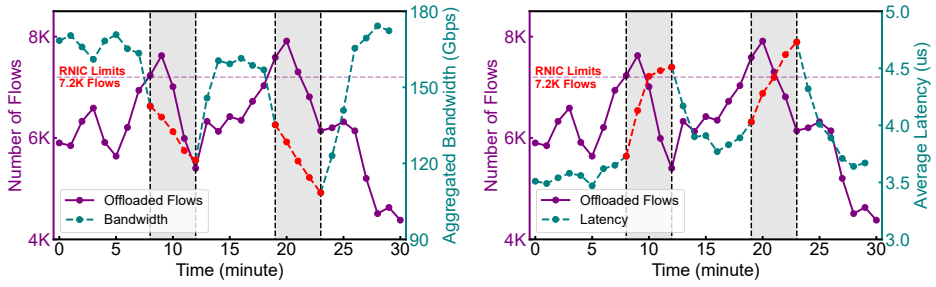


Figure 5: Latency increases when a specific flow (Flow 4) is offloaded.



(a) Aggregated bandwidth.

(b) Average packet forwarding latency.

Figure 6: Workload impacts on the RDMA `write` performance of the CX-6 RNIC.

curring at the virtual switch layer, they are more severe since they make all the containers over the host unavailable.

Figure 4 shows the kernel's failures per year (FPY) with regard to different (models of) RNICs in production. RNICs released recently (e.g., E810, CX-7, and BF-3) bring much more FPYs than those released eight years ago (e.g., CX-4 and CX-5) – note that the Y-axis uses a log scale. The reasons are mainly two folds: 1) newer RNICs have more data to be synchronized, and 2) newer RNICs' kernel drivers are more complex but not time-tested enough. In fact, undesired symptoms regarding the RNIC's driver have not been discovered by the vendors as they did not (and have no means to) test the driver and RNICs under a large-scale production RCN.

RNIC Hardware. The majority (71.8%) of RNIC-related performance issues occur at the hardware layer, which are mainly due to the RNIC's internal defects as we did not find any anomalies in the software stack (i.e., the virtual switch and the driver) when these issues occur. Their root causes are much harder to locate and identify, since today's commodity RNICs are usually a black box to their users.

When network flows are offloaded to an RNIC, the software stack needs to maintain the flow states (e.g., to invalidate the flows when they are aged). During this process, we find that state synchronizations from the control plane to the RNIC's hardware can become extremely slow, making the software unable to offload new flows in time and therefore causing performance degradation (S4). Further, even if the control plane has offloaded the flows to the RNIC, these flows can intermittently go into the software stack for being processed, indicating that the RNIC is not functioning properly (S5).

Moreover, when the workload on the RNIC reaches a certain threshold, the performance of specific flows can become extremely poor, which would in turn affect the performance of other flows (S6). As shown in Figure 5, before the offloading of Flow 4, Flows 1–3 all have a desired latency of ∼4 us on the CX-7 RNIC. When the new Flow 4 is offloaded at around 75s, it does not reach the desired packet forwarding latency of ∼4 us but instead bears a high latency of ∼60 us. Meanwhile, the latencies of Flows 1–3 also increase by 3×.

Another example of S6 is shown in Figure 6. It depicts the performance of the RDMA `write` verb of the CX-6 RNIC with different levels of workloads on it. When there are fewer than 7,200 flows offloaded to the RNIC, the performance is in general negatively correlated with the workload. In contrast, when over 7,200 flows are offloaded, as shown in the shadowed areas of Figure 6a and Figure 6b, the RNIC's aggregated bandwidth instantly drops by 13%–37% and the average latency increases by 27%–34%. Even when the workload drops off to a very low level (e.g., only 600 flows), the performance of the RNIC can hardly recover in quite a while (e.g., 6 minutes) but continues to degrade (cf. §4.2).

For the last two symptoms at the hardware layer (S7 and S8 in Table 1), they are more severe than S5 and S6 since they can make all the containers over the host unavailable. For example, when we de-allocate the VFs of an RNIC from the containers, we observe that the PCIe link state sometimes goes down. Also, when we offload too many VXLAN encapsulations to the RNIC (e.g., 10K on the BF-3 RNIC), the RNIC can become unresponsive and unable to handle any flows. Since our RCN supports multi-tenancy, the number of flows can easily reach 10K+ on a single RNIC, which makes this issue very common in production.

3.3 Challenges

According to the long-term monitoring experiences, we notice quite a few performance issues regarding the scalability of the RNICs in production, where RNICs present undesired performance compared with their official specifications. Thus, CSPs desire an effective approach to understanding the practical limitations of RNICs to prevent performance issues in large-scale deployments. Unfortunately, we have very limited visibility into today's RNICs as their internal design and implementations are usually non-public. Although some of the existing studies have conducted comprehensive tests on commodity RNICs [32, 36, 37, 67], they can only discover performance issues but can hardly tell the root causes and the solutions. Besides, it is rather difficult for RNIC vendors to reproduce and understand such "elusive" symptoms happening in large-scale data centers that involve extremely complicated workloads and environments. In addition, we are not allowed to share real-world workloads with the vendors, making their troubleshooting very difficult.

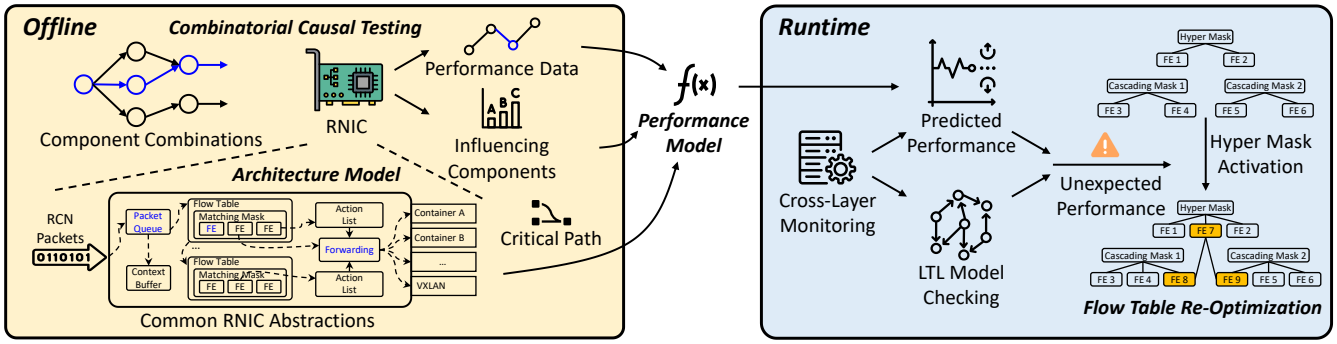


Figure 7: Architectural overview of ScalaCN, a scalable RCN based on the efficient approach of combinatorial causal testing. The blue color of the entities in the offline phase represents their causal relations as a typical example.

4 Design and Implementation

We present ScalaCN, an effective and efficient system for understanding and optimizing RNIC performance in a large-scale RCN through a greybox-like approach. Our key insight obtained from §3 is that the widely existing performance issues on the RNIC's scalability originate from the design and implementation defects of RNICs, which highlights the importance of dissecting the RNICs' internals when these issues occur. Due to the usually non-public implementations of the RNICs, we strive for a more pragmatic approach by *inferring* the most likely causes of the issues according to the common abstractions of an RNIC's components and functionalities.

Figure 7 shows the architecture of ScalaCN, which consists of the offline phase where the combinatorial causal testing is conducted on the RNIC, as well as the runtime phase for RNIC performance prediction and optimization.

- *Combinatorial Causal Testing* (§4.1). In the offline phase, our goal is to reason about the architecture model and approximate the performance model of RNICs based on their common abstractions. However, we are confronted with a prohibitively enormous search space of an RNIC's possible component and functionality combinations. To address this, we first use the topological restrictions among the components to filter out invalid combinations. This significantly reduces the complexity of building architecture models for an RNIC. With respect to each valid architecture model, we further approximate the RNIC's performance model by inferring probable critical paths of the RNIC's packet processing through local sensitivity analysis [19] and permutation removal [15]. Eventually, the most likely performance model becomes our final choice.
- *Performance Interpretation and Prediction* (§4.2). Based on the approximated RNIC's performance model, ScalaCN collects the runtime data of the RNICs and makes the actual predictions on the performance. In this way, ScalaCN prepares to proactively schedule the offloaded network functions. The guidance of the performance model inferred by the combinatorial causal testing enables us to achieve a high prediction accuracy of 98%. We further repair the 2% false predictions by temporal-logic model checking [18, 49].
- *On-Demand Performance Optimization* (§4.3). We do not need the RNIC to always work in its perfect status, but once it is anticipated to undergo a noticeable (empirically set as 5%) performance degradation, we optimize the network function offloading by reorganizing its flow rules to minimize the possible critical path of packet processing in the RNIC hardware. This is achieved by reordering flows with a strong locality into the headmost matching masks in the flow tables that have the lowest delay.

4.1 Combinatorial Causal Testing

Common Abstractions of RNIC Components. As we have mentioned in §2, an RNIC in a production RCN accelerates data communications among containers mainly from two aspects, i.e., RDMA support and packet switching offloading. Specifically, an RNIC provides RDMA verbs (such as the `send`/`receive` and `read`/`write` primitives) to enable each hosted container to use the RDMA protocol to transmit packets without the need for the intervention of the OS. This is typically achieved with the SR-IOV technique [28, 63], where the RNIC is virtualized into multiple virtual functions (VFs) and each VF can be bound to a container.

Besides, an RNIC accelerates the packet switching tasks involved in the RCN. Recall that in Figure 1, an RCN host must be equipped with a virtual switch to decide to which containers a packet should be forwarded (i.e., the switching rules). Also, the virtual switch needs to encapsulate the packet with the VXLAN header (i.e., the applied action) to ensure the packet can cross the underlay network if necessary. These tasks could incur significant overhead if they are all processed in the software stack. Thus, they are offloaded to the RNIC's hardware in the RCN through `TC` or `DPDK` utilities [48].

In other words, the RNIC acts as a programmable switch with RDMA supports to forward packets among the overlay and the underlay. In fact, today's RNICs in data centers all implement *match-action* based embedded switch (or *eSwitch*

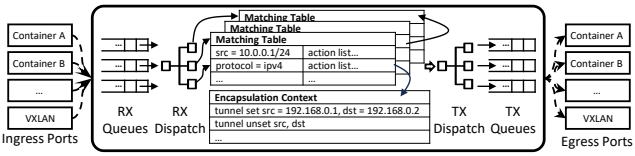


Figure 8: An RNIC's common (abstracted) components in the RCN. The packets come from containers or VXLAN, which are processed by the RNIC and sent to destination ports.

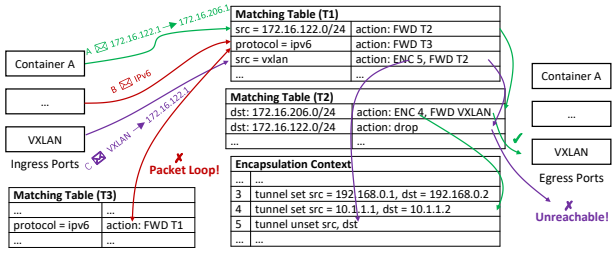


Figure 9: Examples of topological restrictions on the combinations of components.

for short) components to support hardware-accelerated packet switching [38, 41, 55]. This has become the de-facto standard industrial practice and conforms with the kernel's *switchdev* model [1] for handling the data plane in the hardware while keeping the control plane unmodified [12].

Given the above functionalities, we abstract the RNIC's components as shown in Figure 8. Note that we only focus on the *data path* of the RNIC (i.e., the pipeline that processes the packets) since it is the most related to the packet processing performance. Specifically, each container binds to one or more VFs of the RNIC, with which the container can use the RDMA verbs to request the RNIC resources (e.g., queue pairs) and transmit packets through high-performance RDMA protocols. When a packet from the container is received by the RNIC through the VF, it will be enqueued into the container's requested queue pairs and then dispatched to the RNIC's eSwitch component. The eSwitch component is composed of a number of flow tables to match the packets and then apply the recorded actions on them. These tables are offloaded from the control plane in the software stack.

Search Space on Component Combinations. After abstracting the RNIC's components, we aim to infer the likely architecture models of the RNIC. In particular, we need to search for the combinations of components/configurations that are valid inside the RNIC under the environment of our production RCN. If we have such information, we can further reason about the more detailed performance impacts of the RNIC's (abstracted) components and configurations.

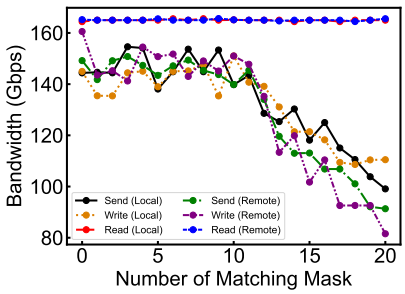
Unfortunately, we are faced with a combinatorial explosion of the possible combinations of the RNIC's components. Today's RNICs are highly programmable and can be offloaded with a large number of combinations of flow tables and RDMA flows. Even for a single flow table offloading, the RNIC can have hundreds of flow entries, each of which can have multiple matching rules and actions. For example, the matcher of a flow entry for NVIDIA CX series RNICs is at least 192 bits long [12], which can cover different protocols, sources, and destinations, from the MAC layer to the transport layer. Each bit of the matching mask is individually configurable. Further, flow tables can be chained together through the forwarding action in each entry of the table for a more flexible packet processing pipeline in the data path, resulting in a more complex structure of the RNIC's components and their interactions during the RCN packet processing.

When putting all possible combinations together, the search space of the combinations is prohibitively large, i.e., $O(k^n)$ possible matchers, where n is the number of tables and k is the average number of table entries. Note that in a production RCN, the actual value of n and k can all be as large as 100.

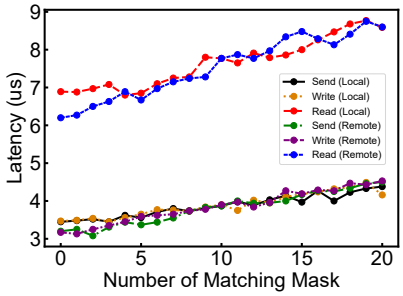
Efficient Searching with Topological Restrictions. In order to reduce the probable combinations of the abstracted RNIC's components and their configurations, we leverage the *topological restrictions* among the components to filter out invalid combinations. Our key idea is that the RNIC's components and configurations are not independent but have strong dependencies and restrictions among them. These restrictions are brought by the network topology and packet reachability.

As shown in the examples of Figure 9, not all the combinations of rules in the table can correctly deliver the packets to the desired destinations. The only valid combination in the figure is the entries of flow tables that packet A goes through. These flow rule configurations enable the RNIC to deliver the packet from container A to the VXLAN interface (i.e., the underlay network). In contrast, the table entries related to Packets B and C violate the topological restrictions. For Packet B, the example combination on flow entries leads to packet loops between flow tables T1 and T3. This is because the IPv6 packets will be forwarded to flow table T3 when they are matched by flow table T2, while T3 will then forward the packet back to T2. Thus, such a combination of flow table entries is not valid in practice. Similarly, Packet C will be forwarded to an unreachable destination, and as a result the corresponding combinations are also invalid. Eventually, we can simply test the eSwitch and queue pair combinations that enable correct end-to-end packet delivery among the containers (i.e., the architecture models of the RNICs), thus reducing the magnitude of component combinations and configurations to $O(k \cdot n^2)$, where k is the number of subnets and n is the average number containers in each subnet.

After pinpointing the valid architecture models in our production RCN for the RNIC, ScalaCN generates test traffic to search for relevant symptoms as we have encountered in Table 1 on each architecture model. To this end, ScalaCN starts to test the flow table combinations from a simple case, i.e., only one flow table. Then, it continuously adds new flow entries to the table with different matching masks, IPs, ports,



(a) Aggregated bandwidth.



(b) Packet forwarding latency.

Figure 10: Sensitivity analysis on the number of matching masks for CX-6.

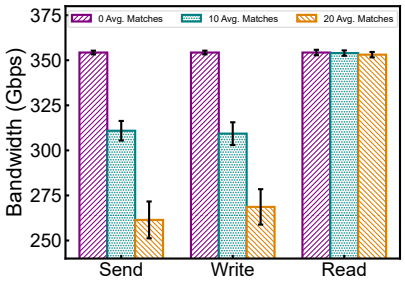


Figure 11: Impacts of the avg. flow table queries of concurrent in-flight packets.

protocols that are reachable in our production RCN, so as to test the RNIC's performance. It also adds forwarding actions to the packet processing rules to connect two flow tables. These rules are installed to the RNIC through the `TC` [5] utility. All these manipulations are conducted with a guarantee that the packets can be correctly delivered. Since we have already filtered out the invalid combinations of the RNIC's components, such testing can be done efficiently and effectively.

Causal Inference. Once the above searching process finds the symptoms (i.e., the performance issues) to occur, ScalaCN conducts a *local sensitivity analysis* to pinpoint the causal relations between the components and the performance issues. ScalaCN incrementally changes each dimension of the input configurations and tests the performance fluctuations. These dimensions come from the test configurations as we mentioned above. During this process, ScalaCN uses the spring-based control model [66] for congestion avoidance to achieve efficient bandwidth testing. If the performance issues can be alleviated or exacerbated after changing the configurations (e.g., increasing or reducing flow tables), we can infer that the corresponding components related to this dimension are likely to lie in the critical path of the RNIC's packet processing.

Here we show an example of the use of local sensitivity analysis to find the possible components that are located in the critical path of the RNIC's packet processing. At first, the searching process of ScalaCN finds that the CX-6 RNIC's performance degrades for certain flows when the number of flow tables increases, whose symptom is similar to S6 in Table 1. However, it is not easy to infer which configurations are likely to be the root cause of the performance issue.

ScalaCN then incrementally changes each testing dimension and examines the resulting performance. Figure 10 shows the performance of CX-6 when a single dimension (i.e., the number of matching masks in offloaded flow tables) of the workload is changed. ScalaCN quickly finds that the RDMA `write` and `send` bandwidth of the CX-6 RNIC is significantly affected by the number of the matching masks in the flow tables, while the `read` verb is hardly affected.

Specifically, the match-action table in RNICs matches packets based on the packet patterns and determines how to process the packet. The offloaded table entries consist of two parts, i.e., the matching mask[1] and (multiple) matching values. The mask indicates which fields of the packets should be matched, while the values specify the content that the packet fields should be if matched. Flow entries with the same mask are grouped and queried together. When the number of masks offloaded to the RNIC increases, ScalaCN finds new flows matched by the newly offloaded masks (i.e., the mask has not been offloaded before) will have a lower performance in the packet processing. Thus, we can infer that the matching masks in the flow tables are likely to lie in the critical path in the RNIC's packet processing. Nevertheless, till now we are still not able to pinpoint the exact components or configurations that really form the critical path in the RNIC.

In order to refine the probable components in the inferred critical path of the RNIC, ScalaCN further performs *permutation removal*. The key idea of this step is to eliminate the components lying in the possible critical path whose removal does not affect the performance of the RNIC. It first sets up a hypothesis H_0 that assumes a specific dimension d is significant to the performance. Then, it permutates the concrete values of the other dimensions, and records the corresponding RNIC's performance. If the probability of the performance degradation is $>95\%$ compared with the tests when d is not set to the current value, we determine that the dimension d is significant (i.e., accept the hypothesis H_0), otherwise we remove the dimension d (i.e., reject the hypothesis H_0)

In this way, we make the critical path more concrete and sort out the real critical paths of the RNIC's packet processing, which guide us to reason about the performance issues and optimize the RNIC's performance. We choose the architecture model with the best accuracy as the final reference.

4.2 Performance Interpretation & Prediction

In this section, we provide detailed interpretations for the symptoms in Table 1, which are then used to predict the

[1] Different vendors have different terminologies on flow tables. NVIDIA names the matching mask as the flow group, while Intel names it as the matching recipe. Although we use the term "matching mask" throughout the paper, its actual meaning is the same as flow group and matching recipe.

RNIC's performance. We conduct a sequential analysis [60] on the packet processing pipeline of the RNICs to validate how packets are delivered and coordinated inside the RNICs.

Flow State Maintenance. We find the possible causes of the state inconsistency problem in the RNICs. First, we note that the flow deletion in the RNICs can sometimes take a much longer time than expected. This symptom is caused by the flow table organization – when a flow is deleted, the RNIC must search the flow table to find the possible flow table dependencies on the specific flow; as a result, if the flow table is large, the deletion process becomes slow, causing S4. Similarly, the flow counters are not updated in time when the RNIC is handling a large number of flows. This makes the software stack determine that the corresponding flow is inactive. Thus, it may delete the offloaded flow from both the software and hardware by mistake, which leads to S1 and S5. These issues mostly happen to the NVIDIA devices.

Flow Tables. With the understanding of matching masks and flow table queries, we demystify the RNIC's performance in Figure 10. For the `read` verb, the local host first requests the remote host's memory address [65]. Its latency increases when the number of local hosts' egress (or remote's ingress) matching masks increase since requesting the addresses involves sending request packets from the local to the remote, which *sequentially* examines the matching masks for the packet's local encapsulation and remote decapsulation.

After the remote host's addresses and keys are requested, the `read` verb only involves packet transmission from the remote to the local [65]. As the number of the remote host's egress-direction masks (or the local host's ingress-direction masks) does not increase, the `read` bandwidth remains stable at 165 Gbps. In contrast, for `send` and `write` verbs, all packets are from the local to the remote. Their bandwidth and latency all degrade when egress-direction masks at the local or ingress-direction masks at the remote increase.

The above also explains the S6 in Table 1. When a matching mask is created, the RNIC sequentially queries the flow table entries. A new mask is queried after the old one, which incurs additional mask query latency. Even if the workload drops off later, the new flow still belongs to the new mask (i.e., it cannot be moved back to the old one which has a lower delay) and the performance cannot recover. We have observe such issues in all the examined NVIDIA and Intel devices. Besides, although the flow tables and matching masks are examined sequentially in the RNIC, the performance of flow matching under the same mask is nearly the same. This indicates that matching values in the RNIC are organized by hash maps under the same group partitioned by the matching mask.

The above micro behaviors of RNICs are actually invisible to the software stack (e.g., OVS) without ScalaCN. As a result, it is difficult for the software stack to schedule the optimal offloading of flow tables and matching masks, which often leads to inferior networking performance in the RCN.

Packet Queuing. We note that before a packet queries flow tables inside the RNIC, it will go through the requested queue in the RNIC. Packet queuing limits concurrent packets processed by an RNIC at the same time. When there is only one flow passing the RNIC, all packets of this flow fill the queue resources and fully utilize the RNIC capacity. However, when there are many flows, queue resources are shared by these flows' requested queue pairs. The bandwidth of a flow is proportional to its allocated queue pairs.

The queued packets affects the RNIC's overall performance jointly with matching mask queries. Figure 11 shows the impacts of queued packets of the CX-7 RNIC. When the average matching mask queries of queued packets increase, the `send` and `write` performance degrades, while that of the `read` verb is still hardly affected as CX-6. For latency, all three verbs are affected by the average matching mask queries similar to the one-flow case (not shown). In other words, an RNIC's overall performance is influenced by the average matching mask queries of queued in-flight packets.

Action Lists and Resource Management. When a packet is matched to a flow table entry, the RNIC applies actions (e.g., encapsulation and forwarding) recorded in the entry. Action lists incur additional delay on packet processing. We notice that VXLAN encapsulation and decapsulation incur performance loss (e.g., $\sim$6% on throughput and $\sim$3% on latency of all examined RNICs). More actions applied on the packet lead to a lower performance. However, as all packets in the overlay-based RCN only require VXLAN actions and port forwarding, such performance loss is constant in our scenario. Further, when the number of offloaded VXLAN actions increases, the corresponding buffers in the RNIC can be exhausted, leading to RNIC unresponsiveness (S2 and S8).

We have reported all the above findings and the inferences on the RNIC's hardware to the relevant vendors, which have been all confirmed. In particular, the inferred causes of S1, S2, S3, and S5 have all been fixed with the driver and firmware updates from the vendors. For the remaining causes, we are closely collaborating with the vendors to fix them.

Performance Prediction. The above combinatorial causal testing provides explainable models for us to understand where the performance issues of RNICs come from. ScalaCN employs the results as a proactive performance predictor to forecast performance degradation. ScalaCN predicts RCN performance on an RNIC basis, i.e., we only focus on monitoring the overall performance of an RNIC.

We use statistical fitting to enable performance prediction on RNICs at runtime. After testing diverse fitting functions including linear functions, polynomial functions, exponential functions, radial basis function [21], and so on, we find that the expected bandwidth BW of various RNIC models can be best fitted with a radial basis function as follows

$$BW(Q_l, Q_r) = u \cdot e^{\frac{-(Q_l-m)^2+(Q_r-n)^2}{2\cdot v^2}} + w, \qquad (1)$$

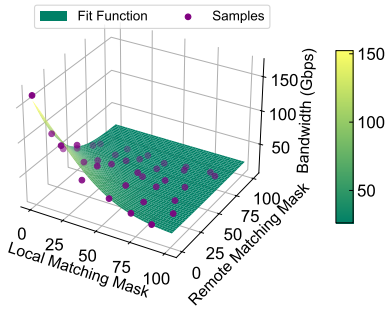


Figure 12: Fitting bandwidth performance of CX-6 with matching masks.

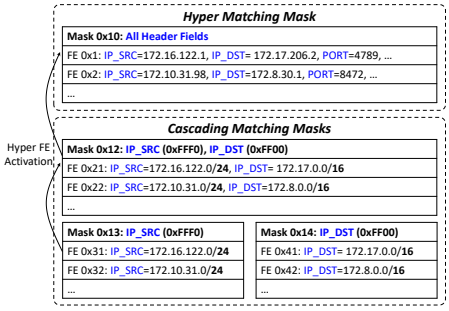


Figure 13: Optimizing RNIC performance by activating the flow entries (FEs).

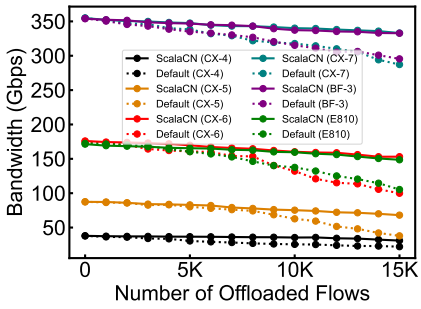


Figure 14: RNIC aggregated bandwidth with and without ScalaCN.

Table 3: Fitting parameters for different RNIC models, where **GD** represents goodness of fit.

RNIC	u	v	m	n	w	a	b	c	GD
CX-4	78.86	39.14	-28.25	-43.19	4.91	0.049	0.063	5.02	0.94
CX-5	148.97	42.34	-29.37	-45.14	12.43	0.051	0.074	4.93	0.93
CX-6	324.47	42.13	-26.92	-49.71	26.28	0.047	0.068	2.69	0.93
CX-7	739.52	48.66	-33.53	-52.88	29.32	0.036	0.045	2.57	0.94
BF-3	748.52	48.01	-33.40	-52.42	30.65	0.037	0.043	2.56	0.94
E810	335.64	43.54	-27.01	-49.55	25.16	0.042	0.069	2.74	0.92

where Q_l and Q_r are average matching mask queries of in-flight packets on local and remote hosts' packet queues, e is the natural base, u, v, m, n, and w are fitting parameters. In particular, u and w control the maximum and minimum bandwidth that an RNIC can reach, and m, n, and v jointly determine the sensitivity of bandwidth changes towards the matching mask queries in the RNIC. Although other packet processing processes in RNICs such as access control also add to performance impacts, they are less dynamic and less significant according to our testing, and thus are also considered as a constant influencing factor w.

The above fitting function conforms with our observation that the bandwidth of RNICs is usually non-linear with a high degree of *rotational symmetry* [24] among the input variables as shown in Figure 10a, where the numbers of matching masks on the local and remote hosts have similar performance impacts. Figure 12 shows the fitting process of CX-6 bandwidth with a high goodness of 0.93.

For the latency of studied RNICs, it can be predicted with a linear function (cf. Figure 10b)

$$LAT(Q_l, Q_r) = a \cdot Q_l + b \cdot Q_r + c, \tag{2}$$

where a, b, and c are fitting parameters. Similarly, a and b jointly determine the sensitivity of latency change towards the matching mask queries, and c implies the constant performance influence of action lists. In fact, a linear function is sufficient for latency prediction as it is the simplest function that can achieve a >90% accuracy. Table 3 shows the fitting parameters of our studied RNIC models.

The average flow table query Q of in-flight packets at time t is calculated as $Q(t) = \frac{1}{n}\sum_{i=1}^{n} q_i^t$, where n is the total number of queued packets, and q_i^t is the number of flow table queries that need to be performed on packet i by the RNIC at time t. The number of in-flight packets is measured by the virtual switch statistics. ScalaCN uses a control-theoretic method, i.e., MPC [45], to keep its monitoring overhead under an acceptable threshold, while quickly responding to RCN events.

4.3 On-Demand Performance Optimization

With the performance predictor, ScalaCN monitors and forecasts RCN performance and states. Recall that the results of combinatorial causal testing indicates the matching masks on flow table queries are the key bottleneck of the performance in all the studied RNICs (cf. §4.1 and §4.2). Thus, once ScalaCN predicts that the performance is about to decline, it optimizes the network function offloading schedule by reorganizing the flow tables so as to reduce the flow table queries.

An RCN can have many packet header patterns for masked matching (cf. §4.1). In our production RCN, a host could have different lengths of IPv4 subnet masks. In practice, the OVS and the kernel offload the corresponding matching mask simply in one shot—once there is a new flow pattern, they will create a matching mask in the RNIC without coordinating with the previous ones (as they are not aware of the RNIC's internals). Thus, the RNIC increases matching masks and degrades performance when flow patterns increase.

Hyper/Cascading Masks. ScalaCN optimizes RNIC performance by minimizing matching mask queries on RNIC packet switching, so as to accommodate RCN scale changes. As shown in Figure 13, ScalaCN reorganizes the RNIC's matching masks in an offloaded flow table into two types, i.e., a *hyper* matching mask and *cascading* matching masks. These two types of masks act like a multi-level cache in a CPU. Each flow table has only one hyper mask ahead of all cascading masks, so that it is queried first with minimum delay.

When there is a new flow pattern offloaded from the OVS and the performance is anticipated to noticeably degrade by an empirical threshold of 5%, ScalaCN creates a corresponding cascading matching mask for it. For example in Figure 13, Mask 0x13 and 0x14 match two new flow patterns (i.e., /24 and /16 IP segments). Once ScalaCN detects a packet between two specific containers matched by existing cascading

matching masks, it will *activate* the concrete flow into the hyper matching mask (e.g., FE 0x1 in Mask 0x10). The hyper matching mask performs an exact match on packet headers and is faster than cascading matching masks since it is located at the headmost. In this way, flow packets always only need to query the hyper matching mask after flow creation.

The first several packets of newly created flows can go through cascading matching masks. To further minimize packet queries on cascading matching masks, we re-prioritize them based on two metrics, i.e., the length of its matching masks (*LM*) and the number of packets in the past 60 seconds it matched (*PM*). A longer matching mask (which matches packets with a more specific pattern and a narrower range) and a larger number of matched packets indicate that the cascading matching mask handles many packets with a strong locality. We thus use the locality score $LS = PM \cdot LM$ to quantify the priority of cascading matching masks. We move the cascading mask with a higher *LS* to the front so that an average packet can query the least number of entries. Besides, we warm up the RNIC's flow tables (i.e., offload probable flows) to accelerate flow creation in case of bursty traffic. This is achieved based on a Gaussian mixture model [52] deriving from our long-term analysis of production traffic patterns.

A hyper matching mask can be filled up by flow entries more quickly since each entry can only exactly match one flow. Thus, ScalaCN needs to deactivate aged flow entries from the hyper mask to preserve the on-chip memory for new flows. Here we utilize the least recently used (LRU) policy to update the flow entries in the hyper matching mask. Similarly, we remove a cascading mask in the flow table when it has no active packets for ten minutes. According to our measurements, removing aged entries in the two types of the matching masks will not affect the performance of the RNICs. This is understandable because most packets will hit the hyper matching mask or higher-level cascading matching masks first when lower-level masks are removed.

4.4 Generalizability

Other RNIC Components. We also observe the performance degradation incurred by the resource contention of on-chip SRAM like queue pair context (QPC), which has been reported in previous studies [62, 67]. ScalaCN can also be used to locate the contention of the on-chip memory. To achieve this, we only need to add a configurable "capacity" dimension on the original queue abstractions before modeling the RNIC's components. Our combinatorial causal testing can then be used on these new dimensions by differentiating the tested performance under different workloads. Nevertheless, as a CSP, we can only optimize the RNIC's performance within the scope of the configurable components of the RNICs, while the on-chip memory contention is usually the inherent hardware limits that require re-architecting the RNICs.

Reusability. ScalaCN is designed to be reusable as it is based on the common abstractions of the RNIC's components involved in an RCN. Specifically, all RNICs provide the support for common RDMA verbs and eSwitch, which can be manipulated through the same set of APIs. Also, ScalaCN builds the performance model for the tested RNIC to validate the modeling accuracy in a closed-loop manner, and each fitting parameter in the model can be individually tuned based on the tested performance of the RNIC. The only manual operation is determining the common abstractions of the RNICs in an RCN (which is a one-shot effort), while the rest of the processes can all be automated through ScalaCN's testing framework. With the above principled methodologies, ScalaCN can be easily adapted to new RNICs.

4.5 Implementation

We implement ScalaCN in 38K lines of C/C++ code and 13K lines of Python code. In order to monitor RCN states, we make use of the OVS command line utilities including `dpctl`, `appctl`, and `ofctl` to acquire data path information. In addition, to transform LTL policies into an automaton, we use the LamaConv tool [11] to generate Moore state machines [16]. ScalaCN does not create new RCN flow rules, but works between the OVS and the RNIC kernel driver to determine the equivalent flow table offloading strategies that maximize the RNIC performance. We modify the OVS module for replacing the offloaded rules on the fly.

5 Evaluation

ScalaCN is gradually used in production, so it is evaluated using both microbenchmarks and production workloads.

5.1 Experiment Setup

The basic settings of the experiments are similar to those in §3.1. For microbenchmarks, we use a middle-scale RCN with 50 hosts, each of which is equipped with four RNICs including either CX-4, CX-5, CX-6, CX-7, BF-3, or E810. We directly use real-world traffic generated by the hosted applications (e.g., large model training and microservices) for evaluation. These applications initiate $O(4\text{M})$ different flows in a day on average and carry out 150–400 Gbps aggregated throughput on each RNIC. We measure the packet forwarding bandwidth and latency of RNICs using the RDMA `perftest` utility [14]. Note that we only measure performance among the same model of RNICs since the inter-operation across different models could result in instability or RNIC failures [67]. In fact, the inter-operations of different models of RNICs are avoided in our production RCN.

5.2 Microbenchmarks

Scalability. Figure 14 and Figure 15 show the scalability in terms of the RNIC's aggregated bandwidth and packet forwarding latency. When the number of flows on the RNIC

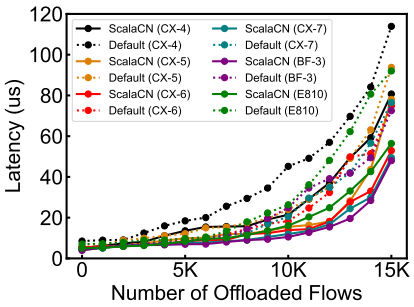


Figure 15: RNIC latency with and without ScalaCN.

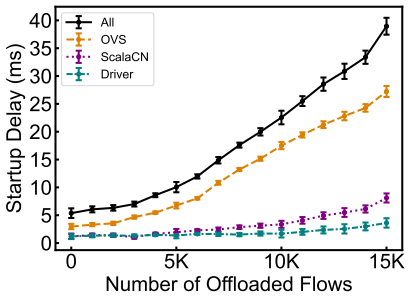


Figure 16: Startup delay of different software-stack components.

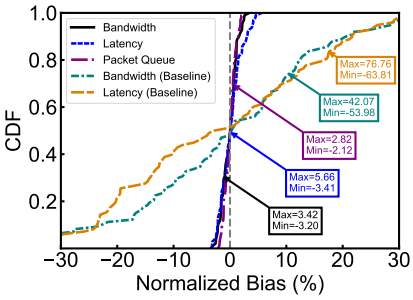


Figure 17: Prediction bias on RNICs' performance.

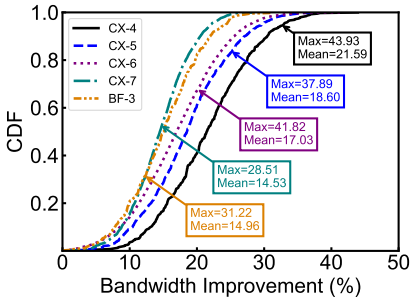


Figure 18: Bandwidth improvement on RNICs in real-world workloads.

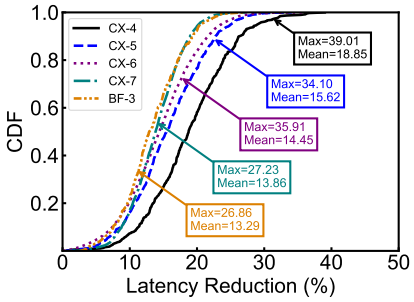


Figure 19: Latency reduction on RNICs in real-world workloads.

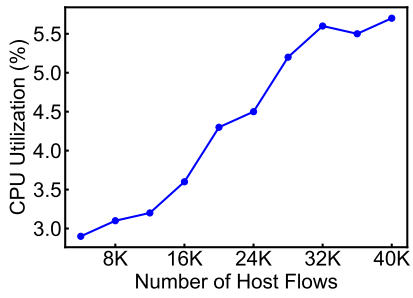


Figure 20: CPU utilization under real-world workloads.

increases, the `send` bandwidth significantly drops on all models with the default OVS offloading strategy. For example, the aggregated bandwidth of the CX-4 RNIC drops by 41%, and the absolute bandwidth drop of the E810 RNIC reaches 70 Gbps. This is because offloading flows to the RNIC intensifies the resource contention inside the packet queue. Meanwhile, the increasing offloaded flows lead to many matching masks created in the RNIC, which increases the processing delay of a packet and thus degrades the throughput. Further, all RNICs present a threshold after which the performance drops more quickly (e.g., 8K for CX-6). Similar trends exist for the RDMA `send` latency as well as `read` and `write` verbs.

In contrast, RNICs' overall aggregated bandwidth with ScalaCN hardly drops. Compared to the default RCN settings with 15K offloaded flows, ScalaCN improves the average aggregated bandwidth of the RNIC by 40.4% and reduces the average packet forwarding latency by 30.5%. This is because ScalaCN proactively reorganizes the RNIC's flow tables when performance degradation is about to happen.

Flow Startup Delay. We measure the startup delay of new flows to show the impacts of ScalaCN on flow initiation with flow table reorganization. Figure 16 depicts the breakdown of the flow startup delay under different numbers of flows. As the RCN scale increases, the startup delay of a flow gradually increases. Nevertheless, most (70%) of the delay is attributed to the OVS, which determines packet forwarding rules in the user space. The additional delay incurred by ScalaCN merely accounts for 18%, which is close to that of the driver (12%). Since the startup delay only affects a couple of packets on flow creation, we feel that such an increase is acceptable given the significant performance improvements of ScalaCN.

Prediction Accuracy. We measure the prediction bias of ScalaCN on the RNIC's performance, and compare it with a baseline prediction method that uses mainstream machine learning techniques (e.g., SVM) with representative flow features like packet loss rate and the number of flows [44, 69]. As shown in Figure 17, ScalaCN achieves a high accuracy on predicting the RNIC's packet queue utilization, with the maximum bias being only +2.82%. Based on this, ScalaCN can further predict the bandwidth and latency with a high accuracy of 98.9% and 98.5%, respectively. In contrast, the baseline method presents poor accuracy, with the maximum bias being -53.98% and +76.76% for bandwidth and latency.

5.3 Real-world Production Workloads

Performance Benefits. We measure the bandwidth improvement and latency reduction on the RNICs that carry our production RCN workloads. As shown in Figure 18 and Figure 19, ScalaCN improves the bandwidth by 17% on an average RNIC, and reduces the average latency by 15%. Also, the maximum bandwidth improvements reach 43.9%, 37.9%, 41.8%, 28.5%, and 31.2% for CX-4/5/6/7, and BF-3, respectively. The latency reductions are 39.0%, 34.1%, 35.9%, 27.2%, and 26.9%, respectively. Similar results exist for the Intel device E810 (not shown due to the space limit).

Different RNIC models benefit differently from ScalaCN. The CX-4 RNIC which was released early benefits the most from ScalaCN. We assume that this is because the early de-

sign of RNICs has a longer delay for each flow table query due to their inferior hardware components. Thus, reducing the number of queries substantially improves efficiency. Note that for the more advanced CX-7 and BF-3 RNICs, ScalaCN still provides considerable performance improvements (i.e., 105 Gbps) since it resolves the hardware bottleneck and reduces the processing complexity for RCN packets.

On the other side, ScalaCN can induce minor performance drops in some cases, in particular the computation-intensive tasks that involve few data communications among containers. The performance drop mainly derives from the flow entry insertion to the hyper matching mask on frequent flow creation. Nevertheless, such a performance drop is trivial ($<$5%) and only occurs to $<$0.03% RNICs in our RCN.

Overhead. ScalaCN needs to continuously monitor the crucial information from software and hardware stacks for performance optimization. As shown in Figure 20, ScalaCN incurs low CPU overheads on different scales of flow offloading. When the host carries more flows, ScalaCN needs to monitor more flow states, and therefore will incur higher CPU overhead. Such overhead is linear to the number of offloaded flows and will finally converge to $\sim$5% on a single core. In the real-world production RCN, the overhead is negligible given the usually tens of CPU cores running on a host.

6 Lessons Learned

Abstraction before Testing. For the performance testing of networking devices such as RNICs and switches, it would be more efficient if we prepare a general abstraction before the actual testing. Such an abstraction can be derived from the hardware datasheet, open-source drivers, and common network APIs. With this abstraction, we can efficiently infer the potential performance bottlenecks and the root causes of the performance issues inside the hardware of commodity devices. In particular, the abstraction helps us identify combinatorial restrictions of the device's input, so that the search space of the performance testing can be largely reduced.

Blackbox Hardware Is Practically Understandable. Although commodity RNICs are mostly close-sourced, we can still shed light on their basic architecture and performance models by concentrating on their critical packet processing pipeline. Based on this realistic insight, CSPs like us can take proactive measures (e.g., optimizing the flow offloading schedule) to mitigate unexpected issues of the RNICs, without the need to modify or re-architect the RNIC (which is costly and at least takes time). Even if we cannot accurately profile the behavior of an RNIC, we can validate some of its key metrics like bandwidth and latency to determine whether a performance degradation is within the acceptable range.

7 Related Work

RNIC Performance Enhancement. Over the past decade, RNICs have gone through a remarkable evolution. Many studies focus on enhancing the performance and scalability of RNICs [25, 29, 33, 59, 62, 63]. SRNIC [63] is a scalable architecture based on FPGA, which minimizes on-chip data structures. StaR [62] balances state maintenance between two communication ends to improve scalability. These studies mostly focus on addressing resource contentions (i.e., hardware limits) of the on-chip memory, while ScalaCN focuses on the on-chip packet processing pipeline that is rarely touched in previous work but is more important for the large-scale use of RNICs. Unlike hardware limits that require re-architecting the hardware, focusing on the packet processing pipeline provides us with an opportunity to enhance the performance even if we have limited visibility into the RNIC's internals.

Troubleshooting Data Center Networks. There have been plenty of studies on troubleshooting data center networks. Most of them focus on verifying the correctness of control-plane rules [17, 20, 23, 50, 61, 68]. Minesweeper [17] and Plankton [50] are two representative approaches to validating network configurations. However, these methods all assume that the underlying RNICs will not fail, and the states as well as performance statistics in the software stack always stay consistent. Such an assumption in fact does not always hold in the production RCN, since RNICs' capability limits can only be discovered in large-scale settings. In contrast, we dissect the RNICs' scalability walls in a large-scale production RCN. This enables us to design ScalaCN to practically address the performance issues when the network quickly scales up.

8 Conclusion

This work presents our efforts towards understanding and mitigating the scalability limit of RCN in a large-scale production environment. In particular, we leverage the efficient approach of *combinatorial causal testing* to interpret the performance issues of RNICs. The derived architecture and performance models guide us to reliably infer the bottlenecks inside RNICs and devise an effective method to overcome the bottlenecks. RNIC vendors' feedback validates our multifold findings and comprehensive evaluation results confirm the efficacy of our solution. In a broader sense, our work provides a principled approach to extracting the high-level mechanisms of proprietary hardware devices, which can benefit a wider range of fields such as hardware testing, verification, and security.

Acknowledgments

We acknowledge Container Network and High-Performance Network teams in Alibaba Cloud that contributed to the success of ScalaCN. We thank our shepherd, Mariano Scazzariello, and the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by the National Key R&D Program of China under grant 2022YFB4500703, the NSFC under grants 62332012 and 62472245, and the Alibaba Research Intern Program.

References

[1] Ethernet Switch Device Driver Model (switchdev) — The Linux Kernel Documentation. https://docs.kernel.org/networking/switchdev.html, 2015.

[2] Introducing Cloud Native Networking for Amazon ECS Containers | AWS Compute Blog. https://aws.amazon.com/blogs/compute/introducing-cloud-native-networking-for-ecs-containers/, 2017.

[3] Advanced Traffic Control - ArchWiki. https://wiki.archlinux.org/title/advanced_traffic_control, 2023.

[4] Configure Azure CNI Overlay networking in Azure Kubernetes Service (AKS) - Azure Kubernetes Service. https://learn.microsoft.com/en-us/azure/aks/azure-cni-overlay, 2023.

[5] Flow Hardware offload with Linux TC flower — Open vSwitch 3.3.90 documentation. https://docs.openvswitch.org/en/latest/howto/tc-offload/, 2023.

[6] tc-flower(8) - Linux Manual Page. https://man7.org/linux/man-pages/man8/tc-flower.8.html, 2023.

[7] Compare network models in GKE | Google Kubernetes Engine (GKE). https://cloud.google.com/kubernetes-engine/docs/concepts/gke-compare-network-models, 2024.

[8] Container Network Interface (CNI). https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/, 2024.

[9] Docker: Accelerated Container Application Development. https://www.docker.com/, 2024.

[10] Kubernetes. https://kubernetes.io/, 2024.

[11] LamaConv—Logics and Automata Converter Library. https://www.isp.uni-luebeck.de/lamaconv, 2024.

[12] NVIDIA Mellanox ConnectX-6 SmartNIC Adapter | NVIDIA. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/, 2024.

[13] Overview - Container Service for Kubernetes - Alibaba Cloud Documentation Center. https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/overview-18, 2024.

[14] perftest. https://github.com/linux-rdma/perftest, 2024.

[15] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. Permutation Importance: A Corrected Feature Importance Measure. *Bioinformatics*, 26(10):1340–1347, 2010.

[16] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–64, 2011.

[17] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proc. of ACM SIGCOMM*, 2017.

[18] Arthur Bernstein and Paul K. Harter. Proving Real-Time Properties of Programs with Temporal Logic. In *Proc. of ACM SOSP*, 1981.

[19] Emanuele Borgonovo and Elmar Plischke. Sensitivity Analysis: A Review of Recent Advances. *European Journal of Operational Research*, 248(3):869–887, 2016.

[20] Tobias Bühler, Romain Jacob, Ingmar Poese, and Laurent Vanbever. Enhancing Global Network Monitoring with Magnifier. In *Proc. of USENIX NSDI*, 2023.

[21] Martin Dietrich Buhmann. Radial Basis Functions. *Acta Numerica*, 9:1–38, 2000.

[22] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding Host Network Stack Overheads. In *Proc. of ACM SIGCOMM*, 2021.

[23] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test OpenFlow Applications. In *Proc. of USENIX NSDI*, 2012.

[24] Sagun Chanillo and Michael Kiessling. Rotational Symmetry of Solutions of Some Nonlinear Problems in Statistical Mechanics and in Geometry. *Communications in Mathematical Physics*, 160(2):217–238, 1994.

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

[26] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proc. of USENIX NSDI*, 2018.

[27] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marcaurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large Scale Distributed Deep Networks. In *Proc. of NIPS*, 2012.

[28] Yaozu Dong, Yu Chen, Zhenhao Pan, Jinquan Dai, and Yunhong Jiang. ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication. *ACM Transactions on Architecture and Code Optimization*, 8(4), 2012.

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

[30] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proc. of USENIX OSDI*, 2022.

[31] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proc. of ACM SOSP*, 2023.

[32] Tao Ji, Divyanshu Saxena, Brent E Stephens, and Aditya Akella. Yama: Providing Performance Isolation for Black-Box Offloads. In *Proc. of ACM SoCC*, 2023.

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

[34] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating Network-based CPU in Container Environments. In *Proc. of USENIX NSDI*, 2018.

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

[36] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *Proc. of USENIX NSDI*, 2023.

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

[38] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing. In *Proc. of ACM SoCC*, 2017.

[39] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. SocksDirect: Datacenter Sockets Can Be Fast and Compatible. In *Proc. of ACM SIGCOMM*, 2019.

[40] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, Jianbo Dong, Lingbo Tang, Hongqiang Harry Liu, Shaozong Liu, Weijie Li, Rui Miao, Yaohui Wu, Zhiwu Wu, Chao Han, Lei Yan, Zheng Cao, Zhongjie Wu, Chen Tian, Guihai Chen, Dennis Cai, Jinbo Wu, Jiaji Zhu, Jiesheng Wu, and Jiwu Shu. Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs. In *Proc. of USENIX OSDI*, 2023.

[41] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-Tenant Networks. In *Proc. of USENIX OSDI*, 2020.

[42] Xu Liu, Peng Zhang, Hao Li, and Wenbing Sun. Modular Data Plane Verification for Compositional Networks. *Proceedings of the ACM on Networking*, 1(CoNEXT3), 2023.

[43] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM Is Lighter (and Safer) Than Your Container. In *Proc. of ACM SOSP*, 2017.

[44] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. A Machine Learning Approach to TCP Throughput Prediction. In *Proc. of ACM SIGMETRICS*, 2007.

[45] Manfred Morari and Jay H Lee. Model Predictive Control: Past, Present and Future. *Computers & Chemical Engineering*, 23(4-5):667–682, 1999.

[46] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martın Casado. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*, 2015.

[47] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC Offloads. In *Proc. of ACM ASPLOS*, 2021.

[48] Nikolai Pitaev, Matthias Falkner, Aris Leivadeas, and Ioannis Lambadaris. Characterizing the Performance of Concurrent Virtualized Network Functions with OVS-DPDK, fd.io VPP and SR-IOV. In *Proc. of ACM ICPE*, 2018.

[49] Amir Pnueli. The Temporal Logic of Programs. In *Proc. of IEEE FOCS*, 1977.

[50] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Brighten Godfrey, and Matthew Caesar. Plankton: Scalable Network Configuration Verification Through Model Checking. In *Proc. of USENIX NSDI*, 2020.

[51] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In *Proc. of USENIX OSDI*, 2020.

[52] Douglas A Reynolds et al. Gaussian Mixture Models. *Encyclopedia of Biometrics*, 741(659-663), 2009.

[53] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *Proc. of ACM SIGCOMM*, 2023.

[54] Jacopo Soldani and Antonio Brogi. Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey. *ACM Computing Surveys*, 55(3), 2022.

[55] Brent Stephens, Aditya Akella, and Michael M. Swift. Your Programmable NIC Should Be a Programmable Switch. In *Proc. of ACM HotNets*, 2018.

[56] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proc. of ACM ISCA*, 2023.

[57] Qiang Su, Chuanwen Wang, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, Dongsu Han, Chun Jason Xue, and Hong Xu. PipeDevice: A Hardware-Software Co-Design Approach to Intra-Host Container Communication. In *Proc. of ACM CoNEXT*, 2022.

[58] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An Analysis and Empirical Study of Container Networks. In *Proc. of IEEE INFOCOM*, 2018.

[59] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proc. of ACM SOSP*, 2017.

[60] Abraham Wald. *Sequential Analysis*. Courier Corporation, 2004.

[61] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and T S Eugene Ng. Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon. In *Proc. of USENIX NSDI*, 2022.

[62] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. StaR: Breaking the Scalability Limit for RDMA. In *Proc. of IEEE ICNP*, 2021.

[63] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. SRNIC: A Scalable Architecture for RDMA NICs. In *Proc. of USENIX NSDI*, 2023.

[64] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *Proc. of USENIX NSDI*, 2023.

[65] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *Proc. of USENIX NSDI*, 2020.

[66] Xinlei Yang, Xianlong Wang, Zhenhua Li, Yunhao Liu, Feng Qian, Liangyi Gong, Rui Miao, and Tianyin Xu. Fast and Light Bandwidth Testing for Internet Users. In *Proc. of USENIX NSDI*, 2021.

[67] Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina. In *Proc. of ACM SIGCOMM*, 2023.

[68] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proc. of USENIX NSDI*, 2014.

[69] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proc. of ACM SIGCOMM*, 2021.

[70] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proc. of USENIX NSDI*, 2019.

Appendix

A Design Details

State Validation. While the performance predictors enable ScalaCN to predict RNIC performance, we still need an effective approach to validating whether the RCN is running in the correct states to make up for possible prediction errors. We take advantage of the linear temporal logic (LTL) [18, 49] to model the expected behaviors RNICs according to the causal testing results. We choose LTL because it can describe the transitions of system states over time. It acts as a test oracle to validate the state correctness and actual performance of the RCN. For example, we use LTL policies "if a flow is offloaded, the RNIC should create a flow entry for it." and "a new flow entry should not be created until a flow is offloaded" to validate the states of RNIC flow entry creation. These two policies can be represented in LTL formulas [49] as $FL_OFFLD \rightarrow \bigcirc FE_CREATE$ and $\neg FL_CREATE \ \mathbf{U} \ FE_OFFLD$. ScalaCN encodes LTL policies to build the LTL validation automaton.

Figure 21 shows the example for the above two specifications. States S_0 and S_1 are undetermined states, S_2 is the error state (i.e., an RCN state inconsistency is detected), and S_3 is the accept state (i.e., the RCN stays correct). If a flow entry is created at the initial state S_0 (without flows offloaded), the automaton goes into the error state and indicates a state inconsistency. If a flow is offloaded to RNICs twice, ScalaCN also determines an inconsistency. Only when the OVS offloads a flow and the RNIC creates the corresponding flow entry, ScalaCN will determine that the RCN stays correct.

It is also possible that ScalaCN cannot identify any issues in the software and hardware stack, but the performance is still very poor (e.g., due to the hardware aging, erosion, or some other environmental factors). To handle these unexpected scenarios, we have a passive monitor on the container network besides ScalaCN. Such a monitor will notify the control plane immediately about the unexpected performance degradation, and the control plane will temporarily disable the allocations of these RNICs to the containers.

Flow Table Manipulations. The flow table policies are initially generated by the software switch. For example, when a new flow is created, it cannot be found in the flow table of an RNIC. In this condition, the RNIC cannot handle the new packet pattern and will transfer the processing of the packet to the software OVS, which will cause a lookup miss event [46] to the OVS. The OVS will then determine which port (i.e., VFs or VXLAN interfaces) this new packet should be sent to and whether it needs encapsulations or decapsulations. After processing the first packet of a new flow, the OVS also offloads the corresponding processing rules to the RNIC hardware. The key problem here is that the OVS is not aware of the RNIC's hardware architecture, and thus cannot offload the optimal flow rule schedule. In particular, it cannot well determine if the flow tables should be reorganized when the number of flow patterns increases. In contrast, ScalaCN can proactively transform these flow rules to the equivalent ones that minimize the overall flow processing delay for the RNIC.

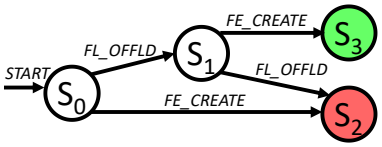


Figure 21: An example automaton from LTL specifications.

Flow Table Locality. The concept of "locality" of a flow rule/table in ScalaCN means that it is frequently hit by the RCN packets. Matching masks with more bits set to "true" indicates that it can match more concrete flows if the table capacity is unlimited. However, it also occupies more flow table entries as each entry is more concrete (e.g., an extreme case is that the full-bit matching of the hyper mask requires each entry to match exactly one specific five-tuple). So the flow table is filled up more quickly for the hyper mask. Considering these two aspects, ScalaCN prioritizes the masks that have a wider range of pattern matching capability and a higher hit rate to the front of the flow table.

B Additional Evaluation Results

Convergence Speed of Causal Testing. ScalaCN completes the entire testing and discovers the performance bottlenecks for each of the tested RNIC models in 3.6 hours on average, which is 60× faster than the brute-force testing that goes through all the possible configurations in more than 9 days. Since our testing is based on the common abstractions of the RNICs in an RCN, the convergence speed for different RNICs is similar. These results demonstrate the effectiveness of our combinatorial causal testing approach in building a high-quality model for the RNICs in a greybox manner.

Impacts on Small Bursty Flows. As we have discussed in §4.3, the introduction of a hyper mask may cause a certain degree of performance degradation for small bursty flows, since their flow rules are not initially prioritized in the whole flow table. We conduct a set of experiments to evaluate the performance of ScalaCN on these small bursty flows, and find that the negative impacts are negligible compared with the performance without ScalaCN. This is because by default the OVS still needs to process the first packet of these small bursty flows without ScalaCN, which usually incurs the same high delay (e.g., tens of milliseconds) due to the lookup miss as in the case with ScalaCN. Also, the first packet of these small bursty flows has a higher probability to be processed by the first cascading mask in ScalaCN due to mask re-prioritizing. Thus, the first-packet latency is still acceptable (e.g., <5 us increase) with RNIC offloading. After the new flow is offloaded to the hyper mask (i.e., the second packet), such a performance degradation will be eliminated.

Eden: Developer-Friendly Application-Integrated Far Memory

Anil Yelam, *UC San Diego*
Stewart Grant, *UC San Diego*
Saarth Deshpande, *UC San Diego*
Nadav Amit, *Technion, Israel Institute of Technology*
Radhika Niranjan Mysore, *VMware Research*
Amy Ousterhout, *UC San Diego*
Marcos K. Aguilera, *VMware Research*
Alex C. Snoeren, *UC San Diego*

Abstract

Far memory systems are a promising way to address the resource stranding problem in datacenters. Far memory systems broadly fall into two categories. On one hand, paging-based systems use hardware guards at the granularity of pages to intercept remote accesses, which require no application changes but incur significant faulting overhead. On the other hand, app-integrated systems use software guards on data objects and apply application-specific optimizations to avoid faulting overheads, but these systems require significant application redesign and/or suffer from overhead on local accesses. We propose Eden, a new approach to far memory that combines hardware guards with a small number of software guards in the form of programmer annotations, to achieve performance similar to app-integrated systems with minimal developer effort. Eden is based on the insight that applications generate most of their page faults at a small number of code locations, and those locations are easy to find programmatically. By adding hints to such locations, Eden can notify the pager about upcoming memory accesses to customize read-ahead and memory reclamation. We show that Eden achieves 19.4–178% higher performance than Fastswap for memory-intensive applications including DataFrame and memcached. Eden achieves performance comparable to AIFM with almost 100× fewer code changes.

1 Introduction

As the memory requirements of datacenter applications increase without a commensurate decrease in DRAM costs, datacenter operators are increasingly concerned by memory stranding. At the same time, host interconnects are delivering higher bandwidth and lower latency [20, 25, 27, 28, 30, 36]. As a result, there is a renewed interest in far-memory systems which enable servers to access under-utilized DRAM on other servers or pooled on dedicated devices within a rack or cluster. Recent efforts have explored a range of issues that arise including prefetching, access models, and fault tolerance [8, 13, 16, 24, 26, 39, 42, 43, 45, 50, 52, 53] but they all make sacrifices along at least one dimension: performance, developer effort, or flexibility.

Existing far-memory systems generally take one of two approaches. Paging-based systems like Infiniswap [24], Fastswap [13] and their extensions [12, 39, 50] rely on *hardware guards* to intercept accesses to remote memory: the microprocessor checks if a page is not locally present and in such case triggers a page fault to fetch it remotely [9]. This approach works with unmodified applications and incurs little overhead on local accesses, but suffers from two issues. First, page faults are expensive because they require trapping into the kernel—which takes on the order of a microsecond, making it impractical to context switch to a different task while waiting for a page fetch to complete (which takes 5–6 μs in our RDMA cluster). As a result, systems such as Fastswap busy wait on page fetches, wasting CPU cycles and limiting performance [13]. Second, their transparent page-based approach does not provide the kernel with the information needed to implement application-specific policies to improve performance (e.g., custom prefetching or eviction).

Alternatively, app-integrated systems such as AIFM [42] and Carbink [53] manage memory in user space at object granularity and rely on *software guards* to intercept remote accesses: applications use remotable pointers which, when dereferenced, check if the data is remote and if so fetch it. Their user-space implementation avoids the overheads of kernel-moderated page faults and makes use of the CPU during a remote fetch by switching to a different lightweight user-level thread. In addition, by operating at object granularity, they can expose data-structure-specific access patterns to the runtime to enable application-specific optimizations. However, app-integrated systems must pay the cost of software guards even on local accesses, and require significant developer effort to port applications to use remotable pointers. Systems like TrackFM [45] and Mira [26] employ compiler techniques to decrease the porting effort but still incur software guard overheads and require source code access to all of the library dependencies of the application, which may be infeasible in many scenarios (§2.1).

We propose Eden, a page-based system that combines software and hardware guards to expose memory access patterns without imposing a disruptive object-based interface. Eden is based on the insight that far-memory applications tend to generate their page faults at a small number of code locations, and those locations are easy to find programmatically. We provide a tool to identify fault locations, and our study of 22 applications shows that, at the median, 12 code loca-

tions cover 95% of page faults. Hence, Eden deploys software guards (code annotations) at these few performance-critical code locations to presage most remote accesses, while relying on hardware guards (page faults) to service the few remaining remote accesses that may occur elsewhere in the application.

The parsimonious use of software guards balances their expressive power against their runtime overhead. By capturing most far memory accesses with code annotations, we avoid almost all expensive kernel-mediated page fetches. Moreover, we show that Eden can effectively coalesce related far memory accesses to provide the same type of application-specific optimizations as systems that resort to object-based interfaces, but with significantly less programmer effort. Finally, by employing software guards only where accesses are likely to be remote, Eden minimizes their overhead.

Based on this insight, Eden asks the developer to instrument these code locations with memory-access hints. By using a hybrid kernel/user-space page management scheme, these hints allow Eden to tailor memory policies to the application (including fetch, eviction, and read-ahead policies) and to leverage a lightweight scheduling framework to run user-level threads during page fetches, thereby improving performance. Specifically, a hint transfers control to Eden's user-space runtime, which checks for page presence and detects an impending fault without trapping into the kernel. If a page is missing, Eden initiates the fetch from user space.

Efficient joint kernel/user-space page management is challenging to realize, as some parts of page-fault handling must occur in kernel context. For example reading the faulting address is a privileged operation on x86 processors. Similarly, mapping and unmapping pages must occur in kernel space. Finally, hints may not be perfect, so Eden's user-space runtime may not catch every fault. Thus, employing a page-based approach with today's CPUs fundamentally requires some kernel participation, whose overheads, if left unchecked, can quickly overwhelm any potential performance benefits.

Eden uses two key ideas to overcome the challenges of hybrid kernel/user-space page management. First, because the kernel retains control of page tables in Eden, syscalls are still needed for tasks such as mapping pages after they are fetched, toggling write protection, and indicating when pages can be reclaimed from a process' address space. Our design streamlines these operations by extending Linux's `userfaultfd` and `madvise()` syscalls to provide *vectorized APIs*. Using these enhanced syscalls, Eden amortizes the cost of necessary kernel traps across many pages, dramatically improving performance.

Second, Eden further uses its hints to pass additional application-specific information to guide prefetching and reclamation. As the hinted accesses cover most page faults in the application, these hints provide a great opportunity to pass data-access knowledge gained either through developer insight or offline profiling. For example, in loops or when accessing large objects, hints can provide precise read-aheads to batch page faults and amortize the page-in cost. Similarly, faulting accesses to certain data structures can be prioritized for reclaim over others by setting different priorities for different hints. We show that such *extended* hints help Eden perform on-par with app-integrated approaches that require significant developer effort. While Eden relies on developer insight to extend hints (similar to app-integrated systems), the effort is smaller and we carefully design the hint extensions to be simple enough for automatic detection and injection based on offline profiling.

We implement Eden on top of Shenango [37], a highly scalable user-level thread scheduling runtime, which Eden leverages to efficiently switch from one thread to another while a hinted fault is being resolved. We evaluate Eden across a variety of workloads both from a performance and developer-effort standpoint. We show that Eden achieves higher throughput than a recent kernel-based far memory system, Fastswap [13], by 19.4–178% on real-world applications such as DataFrame and memcached. Furthermore, Eden achieves similar throughput to an app-integrated approach, AIFM [42], on both DataFrame and synthetic Web service workloads with sufficient local memory. In terms of developer effort, we show that for several example applications, only a handful of hints (2–32) are required to handle 95% of page faults in Eden. In contrast, the AIFM authors modified 1,192 lines in the DataFrame library—108× more than Eden. Our code is openly available at https://github.com/eden-farmem/eden.

2 Background and Motivation

In this section we justify our key design decisions. First, we explain the issues with prior app-integrated systems that we address with our hybrid approach. We then document the overheads inherent in managing traditional, page-based virtual memory from user level and suggest how hints can be used to both communicate application-specific workload information and ameliorate the cost of user/kernel transitions. And finally, we show that there are typically only a small number of locations where such hints are needed so introducing them requires limited developer effort.

2.1 Issues with prior app-integrated systems

The original app-integrated approach to far memory, used in AIFM [42] and Carbink [53], entails significant developer effort and performance overhead due to its remotable pointers. To use these systems, the developer must identify every location where a remote pointer dereference might occur and insert a software guard at each. The guard checks that the data is in local memory, and if not, reads it from far memory. Adding these guards can involve significant modifications; for example, porting the DataFrame library to use AIFM required modifying 7.7% of the lines of code.[1] In addition, while these

[1]The AIFM authors reported modifying 1,192 lines of code [42] out of 15,525 lines of C, C++, and makefiles in the library [2].

guards are highly optimized, they still incur overhead, even if the check succeeds (i.e., the data is already in local memory).

For example, AIFM's guard contains 5 instructions in the best case, and can take up to 489 cycles at the 90th percentile even when data is in local memory. (In contrast, the hardware guards of paging-based approaches use highly optimized TLB hardware.) Unfortunately, for functional correctness, guards are needed throughout the application (anywhere a pointer to the heap may be dereferenced) resulting in a large number of guards. To avoid invoking guards on every access, AIFM employs programmer-assisted *dereference scopes*—small blocks in the program code—wherein the object is marked unevictable and guards are avoided using native pointers. Such optimizations, while helpful, are not always applicable (e.g., streaming workloads where objects experience a single access) and may even add to the programmer burden. For example, the DataFrame library above sees up to 30% slowdown with AIFM even after these guard optimizations. Eden does not need such optimizations because it uses hardware guards where accesses are likely to be local.

More recently, systems such as TrackFM [45] and Mira [26] use compiler techniques to programmatically insert software guards. While effective at reducing programmer burden, neither can eliminate the overhead of guards (e.g., TrackFM's guard contains 14 instructions). Both TrackFM and Mira try to reduce the number of guards required through static analysis, but such techniques have limits (e.g., when the compiler cannot statically resolve branching, function calls, or shared accesses from multiple threads). Ultimately, even with optimizations, TrackFM's overheads are significant: up to 40% slowdown in application performance (§5). Furthermore, for correctness, both systems require instrumenting most or all external library dependencies of an application, which may not be possible (e.g., when source code is not available). In addition, the instrumentation produces larger binaries, incurs higher compilation times—TrackFM reports 2.4× bigger binaries and 6× longer compilation times on average—and complicates the tasks of debugging and profiling.

2.2 Page management overheads

When we combine software and hardware guards, any piece of remote data may be fetched through either our user library (software guards) or page faults (hardware guards). Our library thus needs a way to track and manage page faults to ensure consistent operation: if the user library fetches a page, it should be marked as present in its page table entry, so that subsequent accesses do not cause a page fault; similarly, if the fault handler fetches a page, our software guard should know that the page is local.

The Linux kernel has a simple mechanism to manage page faults in user space called `userfaultfd` [4]. To use `userfaultfd`, a process first registers a region of virtual memory with the kernel and receives a file descriptor in return. Then, if the process faults on an address in that region, the kernel notifies the process by sending an event to the file descriptor. Typically, the process will dedicate a core to polling the file descriptor to detect page faults and resolve them by using the `UFFDIO_COPY` `ioctl()` to map the missing page in the process' address space and unblock the faulting thread. When pages are no longer needed, processes use the `madvise()` syscall to ask the kernel to unmap them, and processes use another `ioctl()`, `UFFDIO_WRITEPROTECT`, to write-protect dirty pages while they are written out.

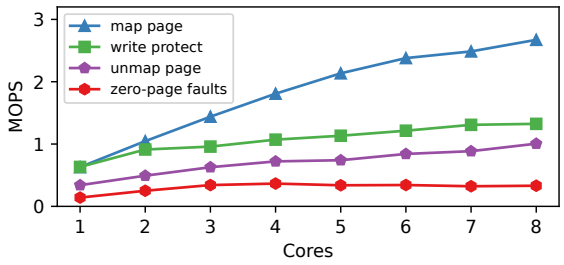


Figure 1: The maximum throughput of memory operations when performed from user space on Linux, as a function of the number of cores. Faults are serviced by four dedicated handler cores; additional cores do not improve performance.

However, we find that `userfaultfd` can entail significant overheads. Figure 1 presents a simple microbenchmark in which a variable number of application cores trigger each of the above memory-management operations as quickly as possible. (We service the zero-page faults on separate handler cores; the other three operations are performed locally at each core.) Even when faults are resolved in the simplest way possible—supplying a zeroed page with `UFFD_ZERO`—we find that the maximum rate at which (any number of) handler cores can poll for faults via a single `userfaultfd` file descriptor is 350 thousand faults per second on our hardware ("zero-page faults" plotted in red).[2] A far-memory system that handles all page faults via `userfaultfd` would only be able to handle ∼12% percent of the 3 million page transfers per second that are possible with a 100-Gbps NIC. While the other three operations are more lightweight, both unmapping (shown in purple) and write protecting pages (green) scale poorly due to the need to flush TLBs across all impacted cores.

Fortunately, our approach invokes this expensive `userfaultfd` machinery only occasionally, as hints can invoke our user-space page-management code before a page fault occurs. Moreover, we provide some simple vectorized enhancements to system calls to reduce the overheads of context switching to the kernel; with these enhancements, our library can manage multiple pages with a single system call.

2.3 Faulting locations

We validate our intuition regarding the number of different places in an application's source code that generate page faults

[2] Using additional file descriptors can improve throughput, but requires sharding far memory into one segment per `fd` (and, hence, across multiple handler cores) potentially causing load imbalance across handler cores.

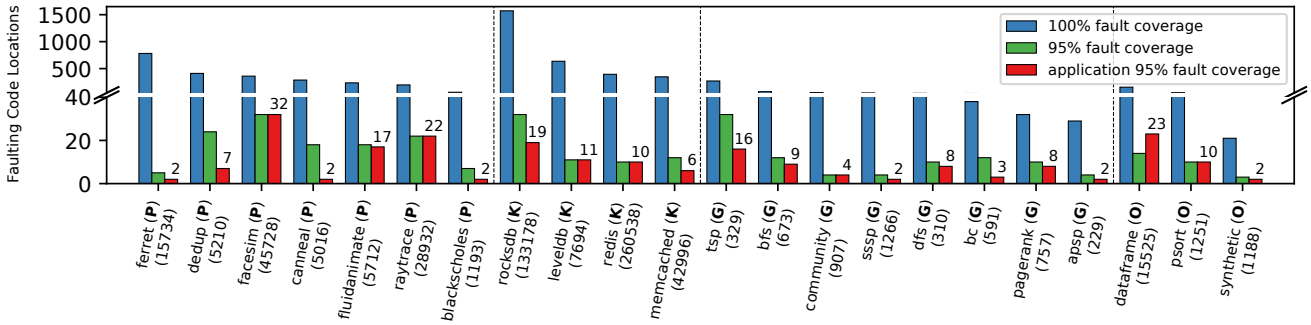


Figure 2: The number of unique lines of source code that account for 100% (blue) and 95% (green, localized to application code in red) of the page faults in 22 different applications. Local memory is configured to hold only 10% of the working set. The *x*-axis labels indicate the total number of lines of code in each application and the application's category (bolded letter).

in a far-memory context—and, therefore, require a programmer to add hints—by conducting a study across a set of 22 applications drawn from different benchmark suites to cover a range of application categories. (These results were published in a workshop paper [14]; we reproduce the high-level findings in this subsection for self-contained review.[3])

Specifically, we consider HPC applications from **P**ARSEC [15], **K**ey-value stores from LK_PROFILE [48] (e.g., LevelDB [5] and Memcached [33]), **G**raph algorithms from CRONO [11], and three **O**ther applications used in our evaluation. We run each application with its standard benchmarking workload to ensure realistic code coverage. Details about the applications and their workloads are provided as Supplementary Material (§A). Using our Eden-based fault-tracing tool (§4), we observe the behavior of each application while restricting the amount of local memory available to 10% of its maximum resident set. The tool identifies all unique fault-triggering source code locations in the application and the libraries it uses.

Figure 2 shows for each application both the total number of code locations that trigger faults and the number of code locations responsible for 95% of faults. In most cases, a small number of locations cover 95% of faults—12 locations at the median and fewer than 32 for all applications. Moreover, when we trace the faults back to the line of application code that led to the fault, even fewer unique lines of code are implicated: less than 10 in most cases. Further experimentation on these and other applications published elsewhere [14] shows that the set of locations is robust across different degrees of memory pressure. Manual analysis reveals the reason is straightforward: few code paths dominate runtime in these applications. At even 10% of the maximum resident set size, local memory is sufficiently large that each page brought in on a code path tends to remain available for later page accesses in that path. As a result, only the initial reference on a given path is likely to cause a fault.

[3]We emphasize that the workshop paper studies faulting locations in far-memory applications; neither Eden nor its evaluation have been published.

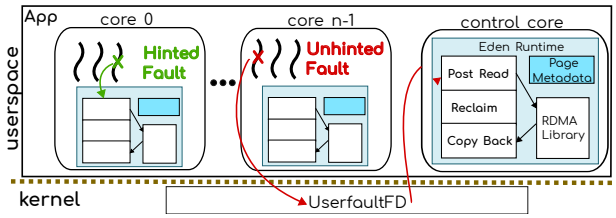


Figure 3: Eden's runtime runs on each application core and a dedicated control core. Hinted faults are handled directly by the runtime on the faulting core while unhinted faults are caught by the kernel and steered to the control core.

3 Eden Design

Eden is a far memory system that aims to achieve three goals:

1. **Low developer effort:** developers should not have to extensively modify their application to use Eden.
2. **Flexible policies:** applications should be able to specify prefetch and eviction policies, and to leverage application-specific information in these policies.
3. **High throughput:** Eden should minimize its overheads as much as possible to deliver high throughput.

Eden's key idea is to manage remote memory accesses with software guards in the common case while avoiding these guards for most local accesses. It achieves this by having the developer add (a small number of) hints to their program about locations where remote accesses are most likely to occur. Eden's runtime can then efficiently check in user space whether the needed page is present, and initiate its fetch if not. In addition, Eden's hints provide an opportunity for applications to convey additional application-specific information about their memory-access patterns to customize prefetch and reclamation policies. Finally, for operations like write-protecting and unmapping pages which require kernel privileges, Eden provides low-overhead vectorized syscalls.

Figure 3 shows Eden's overall design. Application cores run applications in lightweight, user-level threads. Eden's runtime runs on each core and provides general-purpose functionality such as threading and networking. In addition, the

```
hint_fault(
  /* basic hint */
  address, /* faulting address or page */
  mut = false, /* map read-only or writable */
  size = PAGE_SIZE, /* size of the region */
  /* extended hint */
  rdahead = 0, /* positive or negative read-ahead */
  ev_prio = 0, /* eviction priority */
  seq = false, /* sequential access */ );
```

Figure 4: Eden's hint API. Basic hints suffice to signal potential impending page faults; developers wishing to convey additional workload information can use extended hints.

runtime implements memory-management functionality such as logic for handling hints, fetching and evicting pages, and prefetch and reclaim policies. In addition to the application cores, Eden's design includes a dedicated *control core* which is responsible for proactive memory reclamation and handling any "unhinted faults", i.e., those remote accesses which are not presaged by Eden's hints and therefore trigger a hardware guard (i.e., page fault). The kernel steers these (relatively rare) faults to the control core using `userfaultfd`.

3.1 Hint API

Hints are annotations that a developer adds to their program to notify the runtime of a potential page fault and to convey application-specific information about memory access patterns. The runtime uses the information provided by hints to fetch missing pages and avoid almost all page faults, as well as to improve its policies for prefetch and memory reclaim.

3.1.1 Basic hints

Figure 4 shows Eden's hint API. It includes two types of hints: basic hints and extended hints. Eden expects applications to provide at least the basic hints; otherwise all page faults will be handled with high overhead via traditional hardware guards and `userfaultfd`. The key piece of information that each hint conveys is the upcoming memory address (or region) that, if unhinted, might trigger page fault(s), and whether the faulting pages should be mapped read-only or writable. These addresses need not be page aligned. Eden strives to make placing hints as easy as possible, both in terms of lines of code modified and time spent to identify hint locations. We provide a tool to assist in this process (§4).

3.1.2 Extended hints

Eden's extended hints allow motivated developers to pass application-specific knowledge to the runtime. For example, an application can trigger prefetching by using `rdahead` to specify a number of batched pages to read either immediately before or after the hinted address. When the `size` of the hinted region is large, read-ahead can be extremely useful as the entire region can be fetched in a few `rdahead`-size chunks, amortizing the cost of page faults. Note that unlike `size`, `rdahead` is only suggestive and does not necessitate fetching all the pages in the window (e.g., when necessary page locks cannot be obtained). Eden also lets the application specify a page's eviction priority with `ev_prio`, which enables cache separation similar to AIFM's non-temporal hints. In addition, developers can use the `seq` flag to reduce the overhead of hints for workloads that scan data sequentially (e.g., array lookup in a `for` loop). When the `seq` flag is set, the runtime saves the last page accessed by a hint and skips the hint-handling process if the next invocation of that hint points to the same page.

Extended hints enable Eden to convey far more information than what is available to existing prefetch and eviction policies that run inside the kernel. Existing prefetching policies in Linux tend to rely on detecting sequential accesses or using more sophisticated algorithms to detect more complex access patterns [12]. In contrast, when an application knows (even parts of) its memory-access pattern, it can use extended hints to directly convey this information to Eden. The API could be extended to convey access patterns as well (for example to specify non-contiguous pages to prefetch) but we defer a full exploration to future work, focusing instead on leveraging deliberately limited annotation.

3.2 Handling page faults

In Eden, there are two paths for handling page faults, depending on whether the remote access is detected by a software or hardware guard. The *hinted path* corresponds to instances where the address is signaled by a hint before the program attempts to access it—thus, the fault is detected by Eden's software guards and no trap to the kernel is required. The *unhinted path* is followed if the application actually traps to the kernel due to a hardware guard, i.e., traditional page fault.

In both cases, if the page is not present, Eden's runtime fetches the page—or pages, in the case of extended hints—from far memory via RDMA and uses the `UFFDIO_COPY ioctl` to allocate a physical page, copy the fetched data in, and map it in the process' page tables. If the fault is due to missing write privileges, the runtime uses a `UFFDIO_WRITEPROTECT ioctl` to remove write-protection on the page. (The expensive `ioctl` calls are necessary in both hinted and unhinted paths because, unlike pure app-integrated systems, Eden needs to support hardware guards as well. While Eden is in control of what pages are mapped locally, the kernel still controls the page tables and translation hardware that support the hardware guards. Hence, when evicting pages, Eden must unmap them from the page tables.) However, the two paths differ in several aspects including which core handles the fault and whether detecting the fault involves context switching to the kernel.

Hinted path. Eden expects to handle the majority of page faults via the hinted path. When an application invokes Eden's hint API (i.e., `hint_fault(...)`), it calls into Eden's runtime, which implements a software guard. The runtime computes the page-aligned address and looks up the page in its own internal data structure of page metadata. If the page re-

quires that write-protection be removed, the runtime issues a synchronous `ioctl()`. Otherwise, if the page is present, the runtime returns to the application immediately. This process completes entirely on the same core as the application.

If the page is missing, the runtime fetches it from far memory and runs another user-level thread while the page is being fetched (unless the hint specifies otherwise). First, the runtime marks the current user-level thread as blocked. Then it initiates the RDMA read for the page. While the page is being fetched, Eden's runtime can switch to a different user-level thread and run it. This enables Eden to effectively utilize CPU cycles while a page fetch is outstanding, in contrast with existing systems such as Fastswap that busy wait [13]. When control returns to the runtime (e.g., the thread yields, exits, or issues another hint), it polls for any completed RDMA operations on its local queue. On a page-fetch completion, the runtime `UFFDIO_COPY`s the page and resumes the original user-level thread. While issuing the `ioctl()` still requires a context switch to the kernel, hinted faults only require one such context switch, whereas unhinted faults (described below) require two. In addition, the copy happens on the same core, warming up the cache for the application thread, unlike in the unhinted path.

Unhinted path. Hints may not catch all page faults in Eden, so regular page faults can still occur. This can happen if a developer instruments most of—but not quite all of—the potential fault locations, or the system experiences unexpected memory pressure. Eden handles these faults in the control core. In Eden, each application's memory is registered with a single `userfaultfd` file descriptor (§4). When an unhinted fault occurs, traditional hardware guards detect the fault and trap to the kernel, which then writes an event to the `userfaultfd` file descriptor. The control core polls this descriptor and receives the event. It resolves the fault in manner similar to the hinted path with a key difference: Because the control core does not run application threads, it handles other faults or memory reclaim (§3.3) while a page fetch is outstanding. When unhinted faults are rare, the control core is not a bottleneck.

Concurrent faults. It is possible that while a fault (either hinted or unhinted) is being resolved, another user-level thread faults on the same page. The runtime must not re-execute the fault-handling logic for the second fault, because this would trigger an extra page fetch and could potentially overwrite the page of memory after the first thread had already resumed. Eden handles this by including a lock in its page metadata, which the runtime acquires when it starts working on a page fault to ensure concurrent faults are resolved exactly once. When the runtime encounters a concurrent fault, it places the fault on a local waiting queue and proceeds to do other work. When the first fault finishes, the runtime sets the page status bits in global page metadata. Whenever a core returns to the runtime, it checks for completed faults and releases the blocked threads.

Fault stealing. In the common case, Eden resolves hinted faults on the core that triggered them. However, when load is imbalanced or deadlocks occur, Eden can also resolve faults on a different core via work stealing. Load can become imbalanced if an application thread runs for a long time without yielding (Eden inherits Shenango's non-preemptive cooperative threading model [37]) while another core sits idle. Deadlocks can arise due to asynchronous page fault handling where application cores block while waiting for locked pages to be handled by other cores that in turn wait on the former cores. Eden addresses both of these challenges with fault stealing. Faults can be stolen (served) not just by other application cores, but by the runtime on the control core. This is necessary for correctness as all the application cores may block. We ensure progress because the control core never blocks, so any faults waiting past a timeout get handled.

3.3 Memory reclamation

In Eden, both the control core and application cores participate in memory reclaim; this is similar to Fastswap [13], which has a dedicated "reclaimer core". Reclaim involves evicting enough cold pages to maintain the fraction of empty page frames at a configurable threshold. (We use 1% by default.) Most memory reclamation in Eden happens on the control core, since only a small portion of its time is consumed with handling unhinted faults. In some instances, however, the control core may become overwhelmed. In this case (i.e., when there are no empty frames), a faulting core will perform its own reclamation (i.e., instead of running another user-level thread while a page fetch is outstanding).

Regardless of which core performs reclamation, the runtime first must identify which page(s) to reclaim by running a reclaim policy (§4). Once the runtime has selected a page to evict, it first locks the page to check whether it is dirty. (Eden initially write protects pages pulled in by read faults and sets the dirty bit on a write hint or a `userfaultfd` write notification.) If the page is dirty, Eden write protects it using `UFFDIO_WRITEPROTECT` so that it cannot be modified during eviction. Then it copies the page to a buffer and writes it out to far memory using RDMA. Once this is complete—or if the original page was clean to begin with—the runtime removes the page from process memory with the `madvise(MADV_DONTNEED)` syscall.

3.4 Vectorized syscalls

The Userfaultfd syscalls that Eden relies on for memory reclamation (specifically, for write-protecting and unmapping pages), scale poorly due to the need to flush TLBs across cores (§2.2). These operations limit the rate at which Eden can possibly evict pages, and therefore the rate at which Eden can fetch pages as well. As currently implemented in Linux, these operations support batching of contiguous ranges of pages, thereby amortizing the overheads of context switching in and out of the kernel and flushing TLBs. However this is not particularly helpful in our context because the set of pages to be

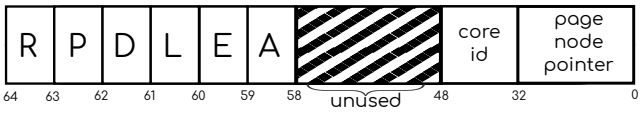


Figure 5: Eden's page metadata format. Each page has six bitflags (Registered, Present, Dirty, Locked, Evicting, Accessed), the ID of the core managing any outstanding faults, and a pointer to the page node for pages present locally.

reclaimed simultaneously is rarely contiguous. Thus Eden introduces vectorized variants of `UFFDIO_WRITEPROTECT` and `madvise(MADV_DONTNEED)`, which enable Eden to reclaim multiple non-contiguous pages in a single syscall and a TLB flush. This significantly improves the scalability of these operations, as shown in Figure 8(a) of our evaluation.

4 Implementation

Our prototype builds on Shenango [46], a user-level thread scheduler written in C/C++. The Eden runtime extends Shenango in several ways. It adds 6,296 lines of code that implement: an `LD_PRELOAD` library to provide transparent support for the standard memory allocation API; fault handling, work stealing and memory reclamation; and page management. Note that while we based Eden on Shenango, Eden can be easily extended to run with other "two-level" thread schedulers like Arachne [40] or GoLang [22]. In addition, Eden includes an RDMA network stack (605 lines) and a far-memory RDMA server (1,304 lines). Finally, we implement a stand-alone tool to identify page fault locations [14] and a patch for the Linux kernel to support vectorized syscalls.

Memory management. Eden intercepts memory allocation functions and forwards them to to `jemalloc` [18], which generates easier-to-manage batched page-sized allocations. Eden supports mapping multiple far memory segments from one or more memory servers as separate memory regions that are registered with `userfaultfd`. Eden maintains a single metadata array per segment with 64-bit per-page entries (shown in Figure 5) that hold page flags, an index to the page node in the reclaim page lists (that is valid only if the page exists locally), and a few bits to save the current application core ID when a fault is in progress. The core ID allows the control core to perform targeted stealing and resolve deadlocks (§3.2). In terms of memory overhead, Eden requires both a 64-bit entry for every mapped page and an additional 24-B page node for resident pages; e.g., supporting 256 GB of far memory with a 64-GB local cache requires 750 MB (∼1%) for Eden's bookkeeping using standard 4-KB pages.

Reclaim policies. Even Eden's basic hints enable more sophisticated reclaim policies than existing kernel-based systems. We implemented four reclaim policies; applications are free to choose the policy that provides the best trade-off for them. With *default*, Eden does not use hints for page hotness and just maintains one big page list that it pops off candidates from. *Second chance* is similar to the Linux kernel's default policy, where hints set a hot bit and Eden maintains two lists and bumps or evicts a page depending on the hot bit. With *LRU*, hints update a timestamp on the page on each access which Eden uses to either evict or bump the page to any of the top $(N-1)$ lists based on how recent the last access was, where N is the configurable number of lists. Finally, with *priority*, Eden lets applications set different priorities for pages faulted in through hints (via the `ev_prio` argument). Eden maintains a configurable number of priority levels for each eviction page list, which are appended to based on the priority set by hints (default is zero for low priority). When evicting, Eden starts with the higher priority levels. This policy generalizes AIFM's non-temporal dereferences, which reclaims an object immediately after its `DerefScope` ends [42].

Fault tracing tool. To find which source code lines should be hinted, we provide a simple tool (also used in [14]) that outputs faulting code locations and the fraction of total page faults they trigger. Like Eden, the tool registers application's memory with `userfaultfd` and records the call stacks on each page fault. While tools like `ftrace` can also be used to collect traces, our tool can run as a part of Eden and help the developer iteratively with the hinting process; when running with Eden, the tool will only record the unhinted faults, and generate flame graphs that highlight only the parts of the application that are yet to be hinted. In future work, we could extend the tool to help detect misplaced hints (e.g., by checking to see if each hint results in page fetches) and automatically suggest hint extensions like `rdahead` and `seq` based on profiled access patterns at the hinted faulting locations. The tool is openly available at `https://github.com/eden-farmem/fltrace`.

5 Evaluation

Our evaluation seeks to address the following questions:

1. How much developer effort does Eden require? (§5.2)
2. How does Eden perform for different applications, and how does it compare to Fastswap, AIFM, and TrackFM? (§5.3)
3. How much do each of Eden's individual elements contribute to its performance? (§5.4)

We answer these questions using four C/C++ far-memory applications: The synthetic Web service and DataFrame library [2] used in the AIFM paper [42], Memcached [33], and an implementation of parallel sort [44]. We use the first two to compare Eden's mechanisms for leveraging application-specific information against those of AIFM (and TrackFM in the case of DataFrame). The latter two represent common real-world workloads. (For brevity, we discuss parallel sort in the Supplementary Material (§A.4), as the results are qualitatively similar to the DataFrame benchmark.)

5.1 Experimental setup

We evaluate Eden and Fastswap on a testbed of three Linux servers, each equipped with a single 100-Gbps Mellanox

App	LoC	# Hints	Coverage	Extensions	Reclaim Policy	Max gain%	Benefits from
DataFrame [2]	15k	11	97.3%	`rdahead,seq`	Default	37%	Read ahead
Syn Web Service [42]	1.1k	4	> 99%	`ev_prio`	Priority	178%	Priority reclaim
Memcached [33]	43k	2	> 99%	-	Second-chance	104%	Latency hiding
Parallel Sort [44]	1.2k	6	> 99%	`rdahead,seq`	Default	19.4%	Read ahead

Table 1: For each application, we show the total lines of code (LoC), the number of hints we add (# Hints), the percentage of faults covered by the hints, any hint extensions employed, the reclaim policy (§4) employed, the maximum relative performance gain over Fastswap at any local memory, and the main reason for that speedup.

ConnectX-5 NIC and two 14-core Intel Xeon Gold 5120 CPUs clocked at 2.20 GHz and provisioned with 94 GB of DRAM per socket. We run on a single NUMA node and disable hyperthreading, TurboBoost, C-states, and CPU frequency scaling. One server plays the role of both the far-memory server and, for certain applications, the client traffic generator. The applications under test run on one of the other two servers: one is configured for Eden and runs Linux 5.15, the other for Fastswap which requires Linux 4.11. We disable page-table isolation (PTI) because it is not available until 4.15, although Eden's performance is similar with PTI enabled.

Both Eden and Fastswap use a dedicated core for memory reclamation, placing them on equal footing. For Eden we use the reclaim policy that worked best for each benchmark (listed in Table 1) while Fastswap uses the Linux default. We employ eviction batching (32 pages) and set the eviction threshold (the local memory usage at which the control core begins to reclaim memory) to 99% of local memory. For Fastswap, the analogous eviction batching parameter is hardwired in the kernel with the reclaim core activating when the memory is above the limit, and other cores joining in if the excess memory use exceeds 2,048 pages.

We run AIFM and TrackFM in the same configurations as their published artifacts on Cloudlab [1, 3]. AIFM requires manual porting and we use only benchmarks ported by the AIFM authors. To facilitate comparisons across all systems, we normalize the results and report the throughput (or runtime) of each system as a fraction of its performance in a fully local baseline configuration, i.e, when no page faults are required. For Eden we measure baseline performance before adding hints (so the normalized numbers incorporate any hinting overhead). In the case of AIFM and TrackFM, we normalize to the performance of versions of the application programs that do not link these libraries to account for their dereference overheads.

5.2 Developer effort

Our analysis in Section §2.3 shows that, for many applications, we can achieve good coverage of potential page faults with a small number of faulting locations. Table 1 summarizes the hinting effort for the applications we evaluate by showing the number of (single-line) hints added, the resulting page fault coverage, and the way hints are extended to benefit each application. We find that we are able to hint faults in Eden for all four of these applications with a handful of hints (at most 11). For DataFrame, the percentage of unhinted fault occurrences is at most 2.7% and for the rest of the applications it remains less than 1%. In the rest of this section, we detail the hinting process for DataFrame as it is the most complex application we consider and allows direct comparison to the effort required for an AIFM port. The process was much more straightforward for the other applications which we omit due to space constraints. (We summarize the analogous process for Memcached in the Supplementary Material (§A.2)).

DataFrame and AIFM porting. DataFrame [2] provides a Pandas-like C++ library for common data-analysis tasks like slicing, grouping, and aggregating table-based in-memory data structures. Tables are stored in columnar format with STL vectors for each column, and, as such, most data accesses occur through vector indexing. The AIFM authors report changing 1,192 lines of code, or 7.7% of the total codebase to port the library to far memory; most of these changes replace the STL vectors used to implement table columns with custom, far-memory-friendly vectors provided by AIFM's customized STL library. Instead of transforming the data structures, TrackFM [45] (like Eden) interposes on all memory allocations using a pre-defined object size for the entire program. It then uses LLVM-based compiler passes to transforms all pointers, while applying a loop-chunking optimization to reduce dereference overheads which is necessary for this heavy array-scanning application. Because NOELLE [32], TrackFM's LLVM C++ frontend, does not currently support certain C++ semantics in its loop analysis, the authors first port the application to C. (Mira's authors [26] reported a similar effort to simplify DataFrame's C++ constructor and list initializations for which their frontend [34] had limited support.)

Hinting DataFrame for Eden. For Eden, we took the original, unmodified DataFrame library [2] and traced faults when running with AIFM's analytics benchmark based on NYC Taxi trip data. 23 distinct code locations are responsible for 95% of DataFrame's faults (Figure 2) but we are able to presage two thirds of the faults by adding hints for just the top-three faulting locations. These locations are highlighted by our tool in a list of top faulting locations ordered by fault density. (Figure 10 in the Supplementary Material shows an example fault-density flame graph for DataFrame). Out of the 23 locations, 13 raise allocation faults for newly created column vectors. Because Eden controls memory allocations, we optionally support pre-faulting small allocations, i.e., those less than 100 MB, at allocation time to avoid the need to add

```
679 new_col.reserve(std::min(sel_indices.size(), vec_size));
680 hint_fault(new_col.data(),new_col.size(),mut=True,rdahead=MAX);
681 for (const auto citer : sel_indices)
682   hint_fault(&(*citer), seq=True, rdahead=MAX);
683   const size_type index =
684       citer >= 0 ? citer : static_cast<IT>(indices_size) + citer;
685   const auto citer = sel_indices[i];
686   const size_type index =
687       citer >= 0 ? citer : static_cast<IT>(indices_size) + citer;
688   if (index < vec_size) {
689       hint_fault(&vec[index], seq=True, rdahead=MAX);
690       new_col.push_back(vec[index]);
```

Listing 1: Lines 680, 682 and 689 are the hints we add to cover the two top faulting locations in DataFrame. Line 680 hints the write faults for the `new_col` vector in 690, whereas 682 and 689 hint the read faults for the lines that immediately follow them. The latter two hints are extended with `rdahead` and `seq` as they access a contiguous vector in a loop.

explicit hints. With DataFrame, this feature obviates the need to explicitly hint all but one allocation location, leaving only 11 locations to hint.

The top-three faulting locations fall in the selection operator where new columns are generated out of existing ones, and involves a lot of data copying; Listing 1 (without the highlighted green lines) shows the portion of original code where a new column is written to from an existing one (Line 690 sees two faults) based on an index vector (accessed in Line 683 that sees another fault). Unsurprisingly, this is one—and by far the most impactful (c.f. [42, Fig. 8])—of the operations that AIFM offloads to the memory server. We add two hints to read in the index and data vectors, and a third to allocate the column vector in one go.

Extended hints. Each of these hints are inside C++ vector scans, prompting us to add `rdahead` to bring in more pages and `seq` to reduce hinting overhead in each instance. (As we show in Section 5.3.1, this enables Eden to perform on par with AIFM.) Overall, our changes to the application include 11 hints, all similar to those above, each incurring an extra line of source code. With the assistance of our tracing tool, it took the first author only a couple of hours to insert the hints and no time was spent in debugging.

5.3 Performance benefits

We compare the performance of Eden to AIFM, TrackFM, and Fastswap [13], a state-of-the-art kernel-based far-memory system. We evaluate each of our four applications with Fastswap, two with AIFM, and one with TrackFM. Eden consistently outperforms Fastswap, due to the benefits enabled by its hints; for each app, a single mechanism (listed in the last column of Table 1) predominantly drives the performance improvement. Eden's hints can capture most of the benefits of AIFM with many fewer code changes. Specifically, for DataFrame, Eden's hint-level read-ahead was enough to match AIFM's data structure-specific prefetching. In AIFM's Synthetic benchmark, Eden's hint-level eviction priority matched AIFM's data structure-level cache separation. We only evaluate the Synthetic and DataFrame applications on AIFM because the effort of porting other applications to AIFM is high and out of scope given our limited expertise with AIFM. Eden bests TrackFM's DataFrame performance due to the latter's need to insert software guards around every potentially remote access as opposed to Eden's few hint locations.

5.3.1 DataFrame

We start with the performance of the DataFrame application described in §5.2. We use the benchmark provided by the AIFM authors [1], varying the local memory between 3 GB and 31 GB, the workload's full working set size. The leftmost graph of Figure 6 shows the overhead of running DataFrame on Fastswap, AIFM, TrackFM, and Eden in fully local configurations. At 100% local memory, AIFM incurs a slowdown of 30% (85 seconds normalized to the 65-second baseline on AIFM's testbed) because of overheads on pointer dereferences, while TrackFM's overhead is even higher (more so than reported in their paper [45]) because TrackFM runs a C-ported version of the application while all the other systems run the original C++ version. Eden incurs a 12% (82 seconds normalized to the 73-second baseline on Eden's testbed) overhead due to its localized hints. Fastswap incurs essentially no overhead due to its exclusive reliance on hardware guards.

The second graph of Figure 6 shows the normalized runtime of DataFrame on each system as a function of available local memory.[4] Fastswap's performance degrades significantly due to page-fault overheads whereas Eden, TrackFM, and AIFM degrade more gradually. As local memory becomes extremely constrained, Eden's performance deteriorates slightly due to an increased prevalence of unhinted faults (rising to 2.7% at 9% local memory); yet even then Eden's performance remains close to that of AIFM—achieving a normalized runtime 37% faster than Fastswap—because Eden is able to amortize page-fault costs with targeted read ahead and eviction batching. For example, with only basic hints Eden's performance at 22% (5 GB) local memory is 4× (296 secs) the baseline (not shown), compared to Fastswap's 2.7×. With eviction batching Eden improves to 2.9× (214 secs)—comparable to Fastswap—and adding targeted read ahead drops Eden's runtime to the final 1.75× (127 secs), competitive with AIFM's 1.67×.

5.3.2 Synthetic Web service front end

We demonstrate Eden's ability to perform application-informed reclaim using the Web service described in the AIFM paper [42]. This application services each client request with a series of lookups: it retrieves a 4-byte user ID by doing several lookups (32 in AIFM's benchmark) into a hash table and then an additional lookup retrieves a large (8-KB) object from a user data array. Both objects are compressed with Snappy [21] and encrypted before being sent back to the client. The workload is designed to stress the hash-table data, which is prone to cache evictions by the much larger user data

[4] This is the experiment depicted in Figure 7 of the AIFM paper [42], but we (like [45]) plot normalized runtime because the benchmark does not specify how to compute normalized throughput.

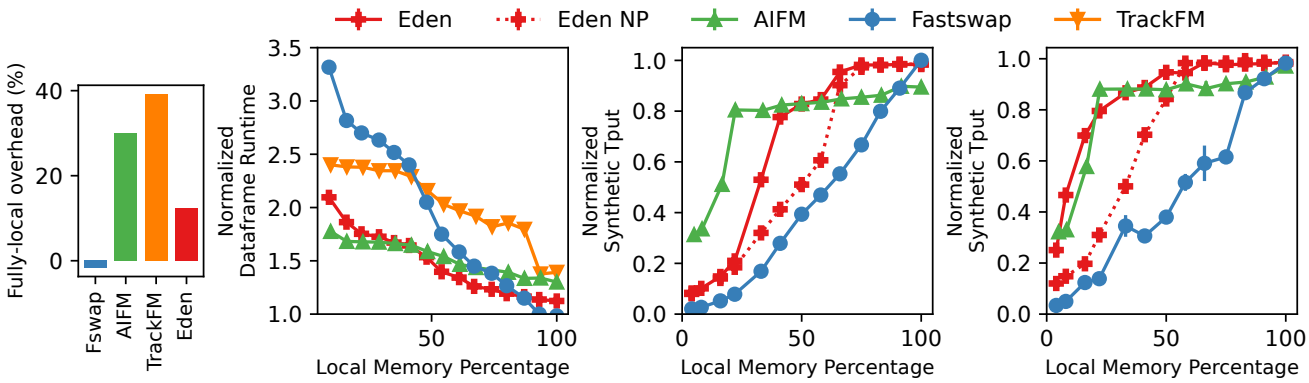


Figure 6: Performance comparison with two AIFM-ported applications. From the left, the first graph shows the performance overhead of DataFrame running on Eden, TrackFM, AIFM, and Fastswap in fully-local configurations compared to its native performance, highlighting the much higher overheads of object-based approaches. The second graph shows the runtime of DataFrame as a function of local memory normalized to the native performance. The third graph similarly shows the normalized throughput of AIFM's synthetic Web server on Eden with and without (Eden NP) priority hints, AIFM non-temporal, and Fastswap. The rightmost graph shows the same experiment with 1,600-byte hash entires.

objects, thus hurting performance under oblivious reclaim policies. AIFM's "non-temporal" feature isolates hash-table data from the user data to avoid cache evictions of the former. Eden deliver similar benefits with priority-reclaim hints (§4).

To facilitate comparison to Eden and Fastswap, we "backport" the application to transparent page-based far memory by replacing AIFM's custom far-memory data structures with standard variants, including the "local-only" hopscotch hashtable available in the AIFM repo [1]. Running the backported application through our fault-tracing tool with AIFM's workload identifies two faulting locations in the hash table and one location for the user-data array access. As a baseline (plotted as "Eden NP"), we add 2 basic hints to the Web service, one before accessing the bucket structure in the hopscotch-hash lookup, and a second ahead of accessing the object itself (the key-value objects are stored separately from the buckets). We consider the workload from the AIFM paper: 128 M hash-table entries and 2 M user objects, resulting in a ∼26-GB memory footprint. Client requests are Zipf(0.8) distributed.

The third graph of Figure 6 plots the normalized performance of Eden, Fastswap, and AIFM, each with 10 cores. At 100% local memory, both Eden and Fastswap achieve normalized throughput close to 1.0 (520 KOPS in absolute terms). However, AIFM incurs overhead due to dereferencing its smart pointers; in this workload this occurs much more frequently than hints in Eden, yielding 10.6% lower throughput for AIFM (385 KOPS normalized to 430 KOPS vanilla performance on AIFM testbed) in fully local configuration.[5] At lower local memory where paging is required, Fastswap degrades roughly linearly, down to a normalized throughput of 0.02 (10.7 KOPS) at 4% local memory. Eden's throughput degrades as well, but remains higher than Fastswap (43 KOPS). Unsurprisingly, AIFM degrades more gracefully than either page-based system, maintaining roughly stable performance until about 20% local memory, at which point almost all local memory is required for the hash table.[6]

To mimic AIFM's non-temporal feature, we prioritize reclaim of array data in the array-access hint using Eden's `ev_prio` setting (plotted as "Eden"). Under this configuration, Eden's performance remains competitive with that of AIFM down to about 40% local memory, as the cache miss rate is low (< 1 miss per request for both) and there is no I/O amplification as all the misses are for page-sized array items—hence, the network read I/O (not shown) for both systems is similar at 15 Gbps. Eden also sees the highest normalized performance gain of 0.49 (or 178%) over Fastswap at this setting. As local memory becomes further constrained, both systems begin evicting (4-B) hash-table entries, causing I/O amplification for Eden. Specifically, AIFM is able to reference far memory on a per-object basis, while Eden must pull in—and evict—entire pages at a time, resulting in less efficient local memory utilization. For example, at 8% local memory, AIFM sees an average of 13 misses per request and generates a network I/O rate of 6 Gbps. Eden sees a much higher I/O rate with 1.2 million page faults at 40 Gbps (which saturates our NIC, explaining the performance bottleneck) due to both a higher miss rate (23 per request) and page-sized I/O (i.e., amplification). Recall that each request in this benchmark can see up to 33 far memory lookups.

To isolate this effect, we re-run the experiment with larger hash-table entries. Specifically, instead of a 4-byte entry as before (for which the memory allocator uses a 32-B slab), we use a 1,600-byte entry (requiring a 2-KB slab) while maintain-

[5] Note that the original AIFM evaluation uses the far-memory-capable variant as a baseline, leading to higher normalized throughput at all local memory percentages presented in their paper [41].

[6] AIFM employs hand-crafted look-aside tables in the smallest (<20%) memory configurations to avoid spilling the hash table to far memory.

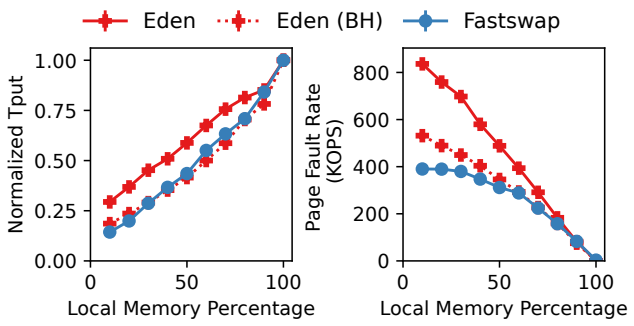


Figure 7: Memcached performance as a function of the fraction of the working set that can fit in local memory. The left-hand graph shows the maximum achievable throughput on both Eden (red; blocking hints in dashed red) and Fastswap (blue) normalized to their respective fully local configurations. The right graph shows the corresponding page fault rate for the same experiment.

ing total hash-table size by decreasing the number of entries to 2 M. In this configuration (shown in the rightmost graph of Figure 6), the normalized performance of Eden and AIFM is similar. Hence, we conclude that Eden's poor performance (relative to AIFM) on the original synthetic workload is due to the interaction of Eden's page-based granularity and the workload's lack of (spatial) locality.

5.3.3 Memcached

Next we consider an application that benefits from our user-level scheduler that hides the latencies more effectively than the kernel: memcached. We add only 4 hints using our tracing tool, 3 to cover item search in the hash table in the GET path and an extra write-fault hint in the PUT path. (More details are in the Supplementary Material (§A.2).) To maximize throughput we use a read-only workload and preload memcached with 30M (24-byte, 512-byte) key-value items, resulting in a ∼8-GB memory footprint. We issue client requests based on a Zipf(1.0) distribution.

Figure 7 shows the performance on 5 cores in comparison to Fastswap. We plot the average across multiple runs; the standard deviation is below 2% in all cases, and usually well below 1%. The leftmost graph plots the throughput of each system normalized to the performance of its fully local baseline (4.45 MOPS for Eden and 3.74 for Fastswap). Because memcached requires almost no computation to service each request, its throughput drops dramatically as memory pressure increases. Eden degrades more gracefully than Fastswap, maintaining 29.4% of its baseline performance with ony 10% local memory as compared to 14.4% for Fastswap—an improvement of 104%. (In absolute terms, the gap is even larger: Eden delivers 1.31 vs. Fastswap's 0.54 MOPS.) For this workload, Eden extracts significant additional throughput from the increased concurrency of its lightweight threads (memcached services each request with a separate thread), allowing it to process client requests in parallel on each core, resulting in a dramatically higher page-fault throughput (836 KOPS) than Fastswap (312 KOPS), as shown in the center graph. To isolate this effect from Eden's other features we also measure the performance when Eden spins during fetches ("Eden (BH)") rather than scheduling another thread.

5.4 Microbenchmarks

In this section we isolate the performance benefits of different components of Eden's design through microbenchmarks.

Vectorized operations First we evaluate the performance of Eden's vectorized operations by repeating the experiment from Section §2.2, for the write protect (`UFFDIO_WRITEPROTECT ioctl()`) and unmap (`madvise(MADV_DONTNEED)`) operations. The leftmost portion of Figure 8 shows that with a batch size of 16, Eden's vectorized operations achieve 5.4–6.6× and 3.7–5.7× as much throughput, respectively, as Linux's standard versions. This is because batching enables Eden to amortize the overhead of context switching in and out of the kernel and batch TLB flushes. Eden's vectorized operations are sufficient to sustain line rate of page operations with 100-Gbps NICs, with 1 core for write protecting and 6 for unmapping pages.

Page fetch rate Next we evaluate the maximum rate at which Eden can fetch pages, with eviction disabled. We run a simple multi-threaded benchmark that first registers a unique, large memory region on each core, then touches every page and evicts it to far memory. Next the benchmark accesses the evicted pages sequentially such that each access incurs a major page fault (but local memory is sufficient to never trigger reclaim). We evaluate three systems: Eden, Eden without hints, and Fastswap.

The second graph in Figure 8 shows that, without hints, Eden is bottlenecked by the control core (which must handle every fault), and cannot sustain high throughput. However, with hints, Eden approaches 2 MOPS, achieving 38–88% more throughput than Fastswap, with the same number of cores. This demonstrates the benefit of non-blocking hints, which allow Eden to continue the microbenchmark and initiate additional asynchronous page fetches.

Fetching and evicting Now we adapt the sequential benchmark above by restricting the amount of local memory available so that every page access triggers both a page fetch and an eviction. We evaluate the performance of two reclaim configurations: non-dirty reclaim and dirty reclaim (with write-back) by touching pages with reads and writes respectively, on initialization. Eden uses a batch size of 16. The third graph in Figure 8 shows that throughput with reclaim decreases significantly for both Eden and Fastswap compared to without reclaim (second graph), because reclaim adds more work to every page fault. Though Eden incurs additional kernel crossings during reclaim for write protecting (when the page is dirty) and unmapping, its vectorized APIs enable it to amortize these costs across several pages. In addition, Eden benefits from overlapping the eviction work with page fetches, allowing it to achieve higher throughput than Fastswap.

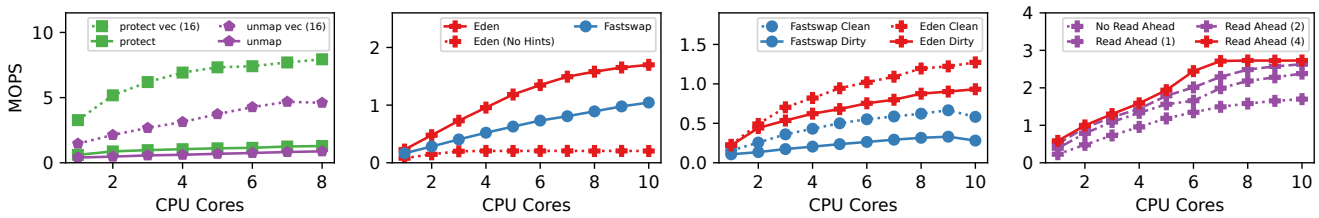


Figure 8: The performance of Eden's vectorized write protect and unmap implementations compared to default Linux (c.f. Figure 1) (a). On a sequential microbenchmark (b,c,d), Eden achieves higher throughput than Fastswap when just fetching pages (b) and both fetching and reclaiming pages (c). In addition, Eden benefits significantly from hints (b) and read ahead (d).

Read-ahead performance Finally, we evaluate the benefits of leveraging application-specific knowledge with Eden's read-ahead feature in its extended hints. We repeat the microbenchmark shown in the third graph ("Eden Clean") with varying amounts of read-ahead, so that Eden can fetch multiple pages in a single RDMA read. The right graph in Figure 8 demonstrates that, with this sequential workload, read ahead improves throughput significantly, saturating the 100-Gbps NIC at high core counts.

6 Related Work

Far memory systems vary widely in terms of programmer effort. Solutions based entirely on transparent hardware guards have been implemented at both the hardware [16, 23, 31] and kernel [9, 10, 12, 13, 19, 24, 29, 39, 43, 50, 51] level, but leave potential application and workload-specific optimizations on the table. In contrast, some app-integrated systems [8, 42, 53] eke out those gains through software guards at the cost of heavy application modifications. Other recent app-integrated systems like TrackFM [45] and Mira [26] use compiler help to automate application changes however they still suffer from significant overhead of their software guards and bring in some practicality concerns like requiring source code for all external dependencies.

Like Eden, DiLOS [52] leverages hardware guards while moving page handling into the application to enable specialized swap caching policies and app-specific optimizations; however, DiLOS employs a LibOS approach to handle hardware guards in the application and cannot run applications natively on Linux. Moreover, while DiLOS's design obviates basic hints, it can still benefit from Eden's extended hints.

Hints, whether specified by the user or a compiler, have long been used to supply prefetching information [35, 38, 47, 49]. Eden's hints go further by both preventing page faults and specifying reclaim behavior. In that way, Eden's hints are semantically analogous to some `madvise()` options. For example, Eden's read-ahead hints are similar to `madvise()` with `MADV_POPULATE` and reclaim priority hints are similar to `MADV_COLD` but with more levels. Critically, Eden's hints are much lower overhead than `madvise()` calls, allowing them to be used more generously in the code. Moreover, `madvise()` guidance is input to the kernel's fixed swap implementation whereas Eden's policies are much more flexible.

7 Conclusion

Eden represents a new point in the design space of far-memory systems: it combines software and hardware guards to intercept accesses to remote memory. Eden uses software guards at the few code locations where most of the remote accesses tend to occur, while relying on hardware guards elsewhere in the application. This approach provides good performance—better than paging-based approaches (pure hardware guards) and comparable and sometimes better than prior app-integrated systems (pure software guards).

It does not, however, tackle a few remaining limitations with page-based accesses. First, for applications that work with small objects and poor locality, managing memory at page granularity can result in overhead due to cache and I/O amplification (as we saw in § 5.3.2). While object-based designs address this issue, such benefits may be easily offset by their high guard and bookkeeping overheads (e.g., object tracking for eviction), which also grow with smaller objects [17]. Second, Eden's far memory handling (miss path) is slower than pure app-integrated systems, especially when Eden triggers a hardware guard and traps into the kernel; in fact, here Eden is even slower than other existing paging-based systems due to its use of `userfaultfd`.

On balance, we believe Eden strikes a useful middle ground between completely transparent page-based systems and existing app-integrated designs. Our approach reduces programmer effort and the overhead of software guards while still enabling application-specific optimizations, which we find more than compensate for the remaining overheads of our page-based design.

Acknowledgements

We thank our shepherd, Sam Kumar, and the anonymous reviewers for their valuable feedback. We also thank Tanvir Ahmed Khan and Baris Kasikci for sharing the dataset [48] we used in our analysis, and Zhenyuan Ruan and Brian Tauro for their help with reproducing the AIFM and TrackFM numbers respectively. This work was conducted under a sponsored research agreement between UC San Diego and VMware Research.

References

[1] AIFM source code and benchmarks. https://github.com/AIFM-sys/AIFM.

[2] C++ DataFrame for statistical, Financial, and ML analysis. https://github.com/hosseinmoein/DataFrame.

[3] TrackFM source code and benchmarks. https://github.com/compiler-disagg/TrackFM.

[4] Linux Userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html, 2022.

[5] Leveldb - an open-source on-disk key-value store. https://en.wikipedia.org/wiki/LevelDB, 2023.

[6] Redis - an open source, in-memory data store. https://redis.io/, 2023.

[7] Rocksdb - a persistent key-value store for fast storage environments. http://rocksdb.org/, 2023.

[8] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: A simple abstraction for remote memory. In *Proceedings of the USENIX Annual Technical Conference*, pages 775–787, July 2018.

[9] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the Symposium on Cloud Computing*, pages 121–127, 2017.

[10] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019*, pages 120–126, 2019.

[11] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 44–55, 2015.

[12] Hassan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *Proceedings of the USENIX Annual Technical Conference*, July 2020.

[13] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020*, page 16. ACM, 2020.

[14] Anonymous. Workshop paper. Please contact PC chairs for a copy.

[15] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[16] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[17] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. A tale of two paths: Toward a hybrid data plane for efficient Far-Memory applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 77–95, Santa Clara, CA, July 2024. USENIX Association.

[18] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDcan Conference*, April 2006.

[19] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, Savannah, GA, November 2016. USENIX Association.

[20] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, Renton, WA, April 2022. USENIX Association.

[21] Google. Google's fast compressor/decompressor. https://github.com/google/snappy.

[22] Google. The Go Programming Language. https://golang.org/doc/, 2021.

[23] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022*

USENIX Annual Technical Conference (USENIX ATC 22), pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[24] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[25] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

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

[28] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 317–330, 2019.

[30] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 267–278, New York, NY, USA, 2009. Association for Computing Machinery.

[32] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, Tommy McMichen, David I. August, and Simone Campanoni. Noelle offers empowering llvm extensions. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 179–192, 2022.

[33] Memcached. memcached: a Distributed Memory Object Caching System. http://www.memcached.org/.

[34] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising c to polyhedral mlir. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, New York, NY, USA, 2021. Association for Computing Machinery.

[35] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *OSDI*. USENIX, 1996.

[36] NVIDIA. Mellanox CX-6 100 Gbps NICs. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[37] Amy Ousterhout, Joshua Fried, Jonathan Behrens, and Adam Belay. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 361—-377, 2019.

[38] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed caching and prefetching. In *SOSP*. ACM, 1995.

[39] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, Boston, MA, April 2023. USENIX Association.

[40] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne:{Core-Aware} thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[41] Zhenyuan Ruan. Private communication, 2023.

[42] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, application-integrated far memory. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*, pages 315–332, 2020.

[43] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, pages 69–87, 2007.

[44] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[45] Brian R. Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C. Hale. Trackfm: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 401–419, New York, NY, USA, 2024. Association for Computing Machinery.

[46] The Shenango Authors. Shenango's open-source release. https://github.com/shenango/shenango.

[47] A. Tomkins, R. H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Sigmetrics*. ACM, 1997.

[48] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. One profile fits all: Profile-guided linux kernel optimizations for data center applications. *SIGOPS Oper. Syst. Rev.*, 56(1):26–33, June 2023.

[49] S. VanDeBogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *ATC*. USENIX, 2009.

[50] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*, April 2023.

[51] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 266–282, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

Category	Count	LoC	Max RSS
Parsec [15]	7	10–32K	110–800 MB
Key-value stores [48]	4	7–260K	0.5–1.3 GB
Graph [11]	10	0.2–1.2K	0.1–300 MB
Other	3	1K-15K	>20 GB

Table 2: Application suite for fault location analysis.

A Supplementary Material

We provide additional information about the applications used in our fault analysis (§A.1) and the hinting process for two of our evaluated applications (§A.2 and §A.3).

A.1 Applications

Table 2 lists the number of applications that we evaluate per category and the lines of code and maximum resident set size (RSS) for each.

PARSEC: PARSEC [15] is a benchmarking suite of compute-intensive applications designed to stress parallel performance on multi-core CPUs. We include a subset of these applications which are more HPC focused. We run each with 8 threads and its *native* dataset.

KVS: We evaluate four key-value store applications: two in-memory caches (Redis [6] and Memcached [33]) and two persistent data stores (Rocksdb [7] and LevelDB [5] with memory caches). We use the workloads from LK_PROFILE but increase the number of operations to at least a million, which yields a proportionally larger memory footprint.

Graph: We consider the CRONO graph benchmark [11], which consists of 10 independent multi-threaded graph algorithms. We use the inputs provided with the benchmark suite and run each program with 4 threads.

Other: Contains applications from our evaluation described in Section 5.3. Synthetic and DataFrame are ported from AIFM's evaluation [42], and psort is our own multi-threaded quicksort implementation.

A.2 Memcached hinting

Here we describe the hinting process for Memcached, a widely used in-memory key-value store. Key-value items are sorted into buckets indexed by the hash of the key, and each bucket is a linked list of items storing a key, its associated value, and other metadata like reference counts, locks, and list pointers. To look up a key, Memcached finds the bucket and walks the list until the key is found. We use a port by the authors of Shenango [46] where each request is served by a separate lightweight thread.

We run the executable on our selected workload (see Section 5.3) linked against our tracing tool. The tool produces a flame graph that visualizes the frequency and call stacks for a configurable fraction of faulting locations. Figure 9 shows the locations of 95% of all faults. The bottom-most layer corresponds to `main` and includes all faults, while the top-most layer shows the functions that include the faulting locations.

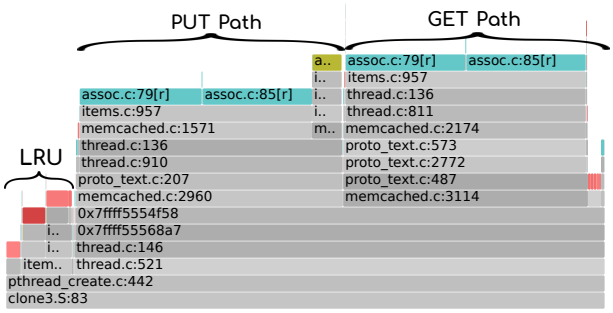


Figure 9: Output of our fault-tracing tool on Memcached.

```
82  item *ret = NULL;
83  int depth = 0;
84  while (it) {
85    hint_fault(&it->nkey);
86    if ((nkey == it->nkey) && (memcmp(key, ITEM_key(it), nkey) ==
          0)) {
87      ret = it;
88      hint_fault(&ITEM\_data(it), it->nbytes);
89      break;
90    }
91    it = it->h_next;
```

Listing 2: In Memcached we add a hint on line 85 to prevent the fault on line 86 and another hint on line 88 to indicate size of the data and ensure that all pages spanned by the item data are faulted in.

Read faults are colored blue, write faults are shown in red, and the protection faults (i.e., the page is present as read only) are greenish yellow. Based on this graph we see that there are two major faulting locations (`assoc.c:79` and `assoc.c:85`) which occur in both the PUT and GET code paths triggered by our 50:50 workload. These correspond to accesses for the bucket and the items. An additional protection fault location appears in the PUT path.

After identifying the faulting locations, Listing 2 shows how we add hints to the item access location, `assoc.c:85`. The fault occurs inside a `while` loop that traverses a linked list in a hash-table bucket: The value of the key being looked up is compared against the contents of the bucket by dereferencing a pointer (`it->nkey`); this dereference frequently causes a fault. We add a basic hint immediately beforehand to prevent this fault. The hint added for the second faulting location (`assoc.c:79`) is similar.

While these two locations cover the majority of faults, there are others. The (write-fault) locations on the far left correspond to accesses by the LRU-maintainer thread. Memcached tracks per-item access information to maintain LRU lists and evicts cold items under memory pressure, thereby dirtying memory even when serving reads. Additionally, it performs "item shuffling" between these lists and updates item pointers in the background, further dirtying memory. As a result, read-only workloads dirty pages at a rate similar to write-heavy workloads. Rather than insert (four additional) hints at these locations, we instead disable both of these unnecessary features in our tests—dramatically improving performance for both Eden and Fastswap—as the underlying far-memory sys-

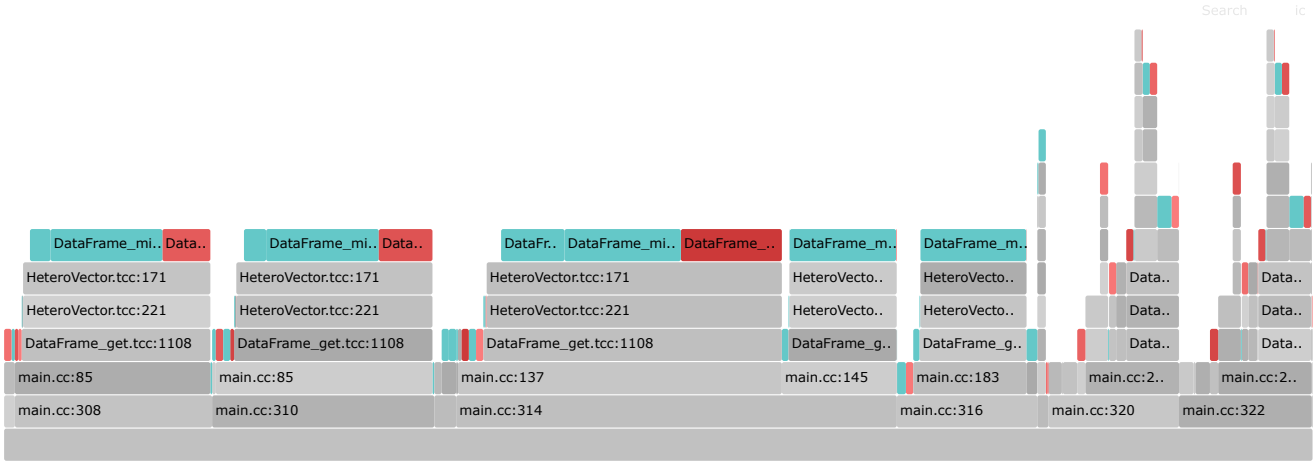


Figure 10: A flame chart for faulting code locations generated from DataFrame's entire execution.

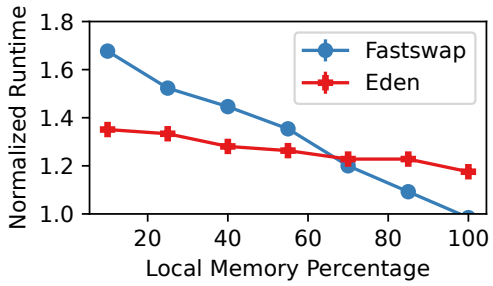


Figure 11: Performance of parallel-sort application with optimal read-ahead settings. The plot shows the run time (lower is better) normalized to the (unhinted) fully local configuration.

tem already takes care of access tracking and moving pages to and from far memory—obviating the need for such internal LRU lists. A number of other rare locations on the right happen when the item found falls across two pages and the page hint brings in only one of them, so we add an extra hint (`assoc.c:88`) for the matched item indicating the size of the item to bring in both pages in such cases.

A.3 DataFrame hinting

Figure 10 presents the fault-density flame graph for DataFrame generated by our tracing tool when running the analytics benchmark. The benchmark runs nine different analytics queries on the data, with seven segments in the flame graph showing the most time-consuming ones.

A.4 Parallel-sort evaluation

Here, we describe the evaluation of a sorting benchmark, in which, unlike the other applications we evaluated, each thread has its own distinct working set. Specifically, we implement parallel sorting with regular sampling [44]. This application first shards an input buffer to individual threads which each quicksort their shard. Then it merges the sorted shards together into an output buffer and copies it back to the input buffer. Because of the overhead of merging results, best performance is typically achieved by matching the number of threads to the number of CPU cores.

Our benchmark sorts 3B 4-byte integers (resulting in a 21-GB memory footprint). We implement two versions: one using Linux pthreads for Fastswap and another using Shenango threads for Eden. We add 6 hints (2 in quicksort, 3 in merge and 1 in copy-back stages) to the baseline code for Eden. The rightmost graph of Figure 11 presents the normalized runtime of both systems running on 10 cores, with Fastswap normalized to the fully local pthreads version and Eden normalized to the (unhinted) Shenango version. (The total runtime does not differ substantively between fully local pthreads and Shenango versions.) Eden experiences a 17.5% slowdown when fully local, most of it coming from the (unnecessary, in that configuration) hints in the partition function of quicksort. However, the (64-page) read ahead signaled through these hints starts helping as memory pressure increases. We plot Fastswap's performance with an optimal *system-wide* configuration of 7-page read ahead. Fastswap extracts less benefit from read ahead because prefetched pages are not mapped until accessed, triggering minor page faults for pages that are already present. As a result, Eden's runtime is 19.4% less than Fastswap's in the 10% local-memory case. Note that the hinting effort, hint extensions and the performance benefits are similar to those of DataFrame as both applications have similar array scanning patterns.

Achieving Wire-Latency Storage Systems by Exploiting Hardware ACKs

Qing Wang, Jiwu Shu*, Jing Wang, and Yuhao Zhang

Tsinghua University

Abstract

We present JUNEBERRY, a low-latency communication framework for storage systems. Different from existing RPC frameworks, JUNEBERRY provides a fast path for storage requests: they can be committed with *a single round trip* and *server CPU bypass*, thus delivering extremely low latency; the execution of these requests is performed asynchronously on the server CPU. JUNEBERRY achieves it by relying on our proposed Ordered Queue abstraction, which exploits NICs' hardware ACKs as commit signals of requests while ensuring linearizability of the whole system. JUNEBERRY also supports durability by placing requests in persistent memory (PM). We implement JUNEBERRY using commodity RDMA NICs and integrate it into two storage systems: Memcached (a widely used in-memory caching system) and PMemKV (a PM-based persistent key-value store). Evaluation shows that compared with RPC, JUNEBERRY can significantly lower their latency under write-intensive workloads.

1 Introduction

Fast Remote Direct Memory Access (RDMA) networks are driving us to build low-latency storage systems [1, 10, 18, 19, 42, 59]. First, RDMA NICs (RNICs) offload network protocol tasks, such as segmentation and checksum, freeing CPU cycles and thus reducing software latency. Second, the RDMA software stack bypasses the OS kernel, thus eschewing latency overhead from context switch and data copying. Finally, the latency can be further reduced by leveraging RDMA one-sided verbs (i.e., RDMA `WRITE`/`READ`) to access remote memory directly; it bypasses the remote CPU, thus detaching the remote CPU's software overhead from the critical path.

Protocol offloads and kernel bypass can bring latency reduction for *every* storage request. However, this is not the case for remote CPU bypass: it is non-trivial to commit a request using one-sided verbs *without increasing the number of round trips*. For simplicity, we term the latency of a single round trip with remote CPU bypass as **wire latency**. Currently, only two use cases can achieve wire latency: simple data access and data replication. In the first case, clients use one-sided verbs to read/write a data block whose remote address is known [18]. However, due to the limited semantics of one-sided verbs, this approach cannot be extended to complex scenarios, such as those containing index traversal and concurrency control, since it will induce excessive round trips [53, 55], offsetting the latency benefits of bypassing remote CPUs. In the data replication case, the leader appends replication requests to followers' replication logs via RDMA `WRITE`, and commits upon the completion of RDMA `WRITE` [9, 18, 58]. Followers execute replication requests lazily in the background. By doing so, the software overhead of followers is removed from the replication critical path.

In this paper, we explore how to achieve wire latency for more general storage requests (beyond simple data access and data replication mentioned before). We begin by identifying what types of storage requests semantically support wire latency. By analyzing, we find that *nil-externalizing* (nilext) requests defined by recent work [21] can meet the requirement of wire latency. Nilext requests do not return execution results or execution errors, i.e., they do not externalize its effects or system state. Hence, the execution of nilext requests can be deferred, making it possible to commit them with server CPU bypass (one requirement of wire latency). Nilext requests are common: for example, `set` requests in Memcached, and `put`/`delete`/`merge` requests in RocksDB [21]. So our goal becomes to make nilext requests wire-latency.

The challenge of achieving wire-latency nilext requests is *maintaining linearizability in the presence of multiple clients*. Specifically, we need to order (nilext and non-nilext) requests issued by different clients for linearizability, and ensure that nilext requests are committed with server CPU bypass. However, RDMA one-sided verbs lack the ability of inter-client coordination, since clients must specify the destination addresses of sent data. This restricts RDMA `WRITE` to achieve wire latency only for single-client scenarios (e.g., replication case where only one leader writes log entries to followers).

We propose Ordered Queue (OQ) abstraction to address the challenge. It is based on our new insight: *the property of remote CPU bypass results from the ability of RNICs to generate hardware ACKs, independent of the type of RDMA verbs*. Specifically, as long as the RNIC supports reliable data transfer, it will generate hardware ACKs, which we can use as the commit signals of nilext requests, to achieve remote CPU bypass. Inspired by the insight, OQ abstraction leverages receive queue, a key structure of RDMA two-sided verbs, to achieve wire-latency nilext requests. It specifies how the CPU and RNIC separately interact with the receive queue using two rules : 1) NIC rule: the server-side RNIC allocates buffers from the receive queue *in order*, accommodating incoming requests, as well as generates hardware ACKs; 2) CPU rule: the server CPU executes requests in the receive queue *in order*

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn)

and returns software responses for non-nilext requests. By using the OQ abstraction, storage systems will generate a linearizable history that follows the positions of buffers in the receive queue, while enabling wire-latency nilext requests.

We build JUNEBERRY, a communication framework that implements the OQ abstraction using commodity RDMA NICs. In JUNEBERRY, clients can mark a request as nilext and use the hardware ACK from server-side RNIC as its commit signal, to make nilext requests wire-latency. The OQ abstraction in JUNEBERRY is centered on an RDMA hardware component called *shared receive queue* (SRQ) [12]. SRQ inherently enforces the NIC rule: RNICs place requests from different clients in an SRQ in order [12]. To enforce the CPU rule and support multiple CPU cores, JUNEBERRY adopts a bitmap-assisted, in-order execution mechanism, and partitions requests into multiple SRQs according to data affinity.

The implementation of the OQ in JUNEBERRY brings two performance issues. First, work queue elements or WQEs (i.e., descriptors) in an SRQ are connected using a link list, not as an array as in a standard receive queue. As a result, the RNIC must fetch them in a pointer-chasing manner, inducing excessive DMA operations. Second, the JUNEBERRY server creates a large number of connections and the RNIC cannot cache their states due to its size-limited SRAM, thus leading to performance degradation. To address the first issue, JUNEBERRY allows a single buffer to accommodate multiple requests by leveraging the multi-packet features in RNICs [56]. Hence, a WQE fetched by one DMA operation can serve multiple requests. To address the second issue, we design a client-side delegation mechanism that supports connection sharing between client threads, reducing the number of connections.

To enable JUNEBERRY to be integrated into storage systems that are crash-consistent, we further add durability support for JUNEBERRY. This is achieved by leveraging emerging persistent memory (PM) technologies. Specifically, JUNEBERRY places SRQ buffers in PM and makes the hardware ACK have persistence guarantees. Moreover, we design a recovery procedure that scans PM-resident SRQs to identify valid requests and then restores storage systems to a consistent state.

We augment two storage systems with JUNEBERRY: Memcached [6], a widely used in-memory caching system, and PMemKV [7], a PM-based persistent key-value store. We evaluate them in a cluster of 9 machines. For Memcached, we run representative traces from Twitter's workloads [63]. Compared with RPC, JUNEBERRY reduces median latency and P99 latency by up to 90.70% and 45.65%, respectively. We also compare JUNEBERRY with a software approach that returns responses of nilext requests before execution: JUNEBERRY still reduces median/P99 latency by 83.77%/42.86%. This is because by leveraging hardware ACKs, JUNEBERRY eliminates any server-side software delay from the critical path of nilext requests, including software queueing. For PMemKV, we run YCSB-like workloads. Under write-intensive workloads (i.e., 50% `put`), JUNEBERRY reduces median latency by 40.83%. Under read-intensive workloads (i.e., 5% `put`), JUNEBERRY has 1.19× (1.6μs) higher median latency, which results from the overhead of supporting durability, including disabled DDIO and PM-resident SRQs (Optane PM [5] in our evaluation).

In summary, we make the following contributions.

- Ordered Queue (OQ) abstraction, which leverages RNIC hardware ACKs to enable wire-latency nilext requests while maintaining linearizability.
- JUNEBERRY, a communication framework that implements the OQ abstraction with commodity RNICs, optimizes performance using hardware/software techniques, and supports durability with persistent memory.
- A set of evaluations on Memcached and PMemKV that demonstrate the performance benefits of JUNEBERRY.

2 Background

In this section, we first provide the background of RDMA (§2.1). Then, we show how existing work utilizes RDMA one-sided verbs to achieve extremely low latency (i.e., wire latency) for storage systems (§2.2).

2.1 RDMA

As the CPU performance stagnates but the network bandwidth continues to grow, it is prevalent to offload network-related processing tasks to NICs [11, 22, 46, 48, 50], freeing valuable CPU cycles. RDMA is a typical example, which is widely deployed in data centers due to high performance and high CPU efficiency [1, 22, 23].

RDMA NICs (RNICs) execute RDMA protocol in hardware and provide queue pair (QP) abstraction to applications. A QP consists of a send queue (SQ) and a receive queue (RQ), each being associated with a completion queue (CQ). A sender submits RDMA commands (verbs), i.e., `WRITE`/`READ`/`SEND`, to the SQ. `WRITE` and `READ` are one-sided, which means that the sender can directly access remote memory without involving the receiver's CPUs. In contrast, `SEND` is two-sided, since the receiver needs to push `RECV` commands to the RQ to prepare receive buffers, which accommodate incoming `SEND` messages. Upon the completion of RDMA commands, the RNIC generates completion entries (CEs) to the associated CQs. RDMA offers several transport modes, of which the *reliable connection (RC)* mode is the most commonly used. RC mode supports 1) reliable data transfer, 2) one-sided `READ`/`WRITE`, and 3) larger-than-MTU message delivery, but it needs one-to-one QP connections.

2.2 Achieving Wire-latency via One-sided Verbs

RDMA one-sided verbs (i.e., `READ`/`WRITE`) are capable of accessing remote memory with remote CPU bypass. This enables extremely low-latency data transfer by avoiding the receiver's CPU software overhead. For simplicity, we term the latency of a single one-sided `READ`/`WRITE` (i.e., *a single round trip with remote CPU bypass*) as **wire latency**.

Researchers have leveraged one-sided verbs to achieve wire-latency replication for storage systems [9, 18, 19, 58].

For example, Mu [9] redesigns Paxos-based state machine replication (SMR): the leader appends a log entry to followers' replication logs via one-sided `WRITE` commands; once the majority of `WRITE` commands have been completed, the leader can ensure the log entry is committed. Followers execute log entries asynchronously in the background. As with the traditional RPC-based approach, Mu completes replication in a single round trip. The difference is that Mu removes the CPU overhead of followers from the replication critical path. Therefore, Mu achieves wire-latency for replication requests (i.e., 1.3μs for leader replicating small data [9]).

3 Motivation

Motivated by the one-sided replication case, we explore a more fundamental question in this work: *Can we achieve wire latency for general storage requests (not just replication requests)?* In this section, we analyze the feasibility and locate the associated technique challenges.

What types of storage requests semantically support wire latency? If a storage request can be realized in a wire-latency manner, it must meet certain requirements, such as not returning an execution result. This is because a wire-latency request is committed without waiting for the server CPU to execute. We find the *nil-externality* property defined by a recent work [21] perfectly describes the requirements.

Formally, a storage interface is nil-externalizing (nilext) if it does not externalize its effects or system state immediately [21]: it just returns an acknowledgment indicating committed, rather than returning an execution result or error. Nilext interfaces are common in storage systems: for example, in Memcached, `set` is nilext; in RocksDB, `put`, `delete`, and `merge` are nilext [21]. By exploiting nil-externality, the execution can be performed *asynchronously* on the server CPU, which makes wire-latency nilext requests possible.

What are the benefits of wire-latency nilext requests? Accelerating nilext requests will notably reduce overall latency (including tail and median) that clients perceive. First, the nilext requests are typically writes and read-modify-writes; executing them is more complicated and thus slower than executing read requests. For example, Memcached's `set` involves memory allocation and object eviction; LevelDB/RocksDB's `put` appends log entries to a write-ahead-log (WAL) file. Making them wire-latency will lower the tail in latency distribution. Second, according to recent studies, many workloads have a non-negligible write ratio [15, 54, 63]. For example, in Twitter, more than 35% cache clusters generate more than 30% `set` requests (nilext) [63]; in Meta's large-scale AI/ML services UP2X, which uses RocksDB for storage backend, about 92.53% of request are RocksDB's `merge` (nilext) [15]. Under these write-heavy workloads, wire-latency nilext requests will reduce the median latency.

Can wire-latency nilext requests be achieved via RDMA `WRITE`? Unfortunately, we give a negative answer by analyzing the limited semantics of RDMA `WRITE`. To guarantee linearizability [24] of storage systems (e.g., after a `set(k,v1)` is committed, the subsequent `get(k)` can return `v1`), we need to produce a linearizable order for requests from *different clients*. However, RDMA `WRITE` cannot achieve coordination between clients: a client must specify the destination address of the server when issuing an RDMA `WRITE`. If we use an external component like sequencer [13, 16, 38] for inter-client coordination, i.e., obtaining an increasing address from the sequencer before issuing `WRITE` and using addresses as the linearizable order, it will induce an extra round trip [13], violating the definition of wire latency, or require programmable switches [16, 38], hindering wide deployment. We also cannot use the server CPU to order requests, since wire-latency requests need to be committed with server CPU bypass. Now, we revisit why RDMA `WRITE` can achieve wire latency for replication. The reason behind it is simple: in the replication case, the server (i.e., a follower) has only a *single* client (i.e., the leader), so the leader can easily specify the order for replication requests by generating increasing destination addresses *locally*; but it does not work for the multi-sender scenario.

In summary, due to interface semantics of nilext requests, making them wire-latency is possible. If realized, we can obtain latency benefits for the upper storage systems. Yet, the challenge is how to guarantee linearizability in the presence of multiple clients, which RDMA `WRITE` cannot tackle.

4 Ordered Queue Abstraction

According to the analysis in §3, RDMA one-sided verbs are incapable of realizing wire-latency nilext requests, since they do not support inter-client coordination inherently. It seems to be in a predicament: the common view is that only one-sided verbs can transfer data in a remote-CPU-bypass manner, the requirement of wire latency.

We break the common view by reexamining what it means to bypass a remote CPU. For an RDMA `WRITE`, when the client gets a completion entry (CE) by polling the CQ, it can guarantee that the sent data will reliably reach the server memory; the entire process does not involve the server CPU. How does the client obtain the guarantee in a remote-CPU-bypass manner? By further looking at the hardware mechanism of RNICs, the answer is explicit: RNICs can generate hardware ACKs due to offloading reliability. Specifically, to support reliable data transfer (RC mode), upon receiving a packet, the receiver-side RNIC generates a hardware ACK to confirm safe delivery, or a negative ACK (NACK) to indicate out-of-order delivery[1]. The sender-side RNIC handles ACKs by generating CEs to the associated CQs; it handles NACKs by performing go-back-N retransmission or Selective Repeat (SR) retransmission (e.g., ConnectX-6 RNIC). This reveals a new insight: *the property of bypassing the remote CPU results from the ability of RNICs to generate hardware ACKs, independent of the type of RDMA verbs*.

[1] If a hardware ACK is lost, the RNIC will resend it.

Inspired by the new insight, we find that it has the potential to achieve wire-latency nilext requests by leveraging two-sided verbs (i.e., RDMA `SEND`). First, RNICs also generate hardware ACKs for RDMA `SEND`, which we can use as the commit signals of nilext requests, ensuring the property of bypassing the remote CPU. Second, unlike RDMA `WRITE`, clients do not need to specify destination addresses for RDMA `SEND`: the server-side RNIC writes the data into buffers in the receive queue (RQ); this gives the server-side RNIC a chance to order requests from different clients.

We therefore propose Ordered Queue (OQ) abstraction, which enables wire-latency nilext requests while maintaining linearizability of the whole storage system. In principle, in OQ abstraction, the server-side RNIC produces a linearizable order for requests from clients, and the server CPU executes requests according to the order. Concretely, OQ abstraction consists of two rules that define how the server-side RNIC and server CPU interact with the RQ, respectively:

Rule 1 *(NIC rule)*. The RNIC allocates receive buffers *in order*: ① the RNIC always allocates receive buffers from the head of RQ, and ② returns hardware ACKs *after* allocation.

Rule 2 *(CPU rule)*. The CPU executes requests *in order*. In other words, the CPU always fetches the first receive buffer not yet executed in the RQ, executes the associated request, and finally returns software responses for non-nilext requests.

A request is committed when the client receives hardware ACKs (for nilext requests) or software responses (for non-nilext requests). Using the above two rules, storage systems will generate a linearizable order that follows the positions of receive buffers in the RQ, even using hardware ACKs as responses of nilext requests. Here, we prove that a request (R_2) sees the effects of any request (R_1) that is committed before it invokes. There are three cases:

Case 1: R_2 is nilext. In this case, according to the definition of nilext requests, R_2 does not need to return any state of storage systems, including effects of committed R_1.

Case 2: R_1 and R_2 are both non-nilext. In this case, R_2 is invoked after the client receives the software response of R_1. According to **Rule 2**, server CPU executes R_1 before returning the software response. Thus, R_2 will reflect the result of R_1.

Case 3: R_1 is nilext but R_2 is non-nilext. In this case, R_2 is invoked after the client receives the hardware ACK of R_1. According to **Rule 1**, at the server side, R_1 has a more advanced position in RQ than R_2. Further, according to **Rule 2**, the server CPU executes R_1 before R_2, so the software response of R_2 will see the effects of R_1.

Here, we give an example of OQ using Figure 1. Suppose the back-end storage system is Memcached. **(a)** Initially, at the server side, the RQ is empty and the Memcached instance contains a KV pair ⟨k, v0⟩. **(b)** Then, a nilext request `set(k,v1)` arrives at the first buffer on the RQ; and the RNIC returns a hardware ACK indicating committed, achieving wire latency. **(c)** After that, a nilext request `set(k,v2)` experiences the same process with the second receive buffer; at the same time, the CPU asynchronously executes requests in RQ in order (i.e., executing `set(k,v1)` first). **(d)** Then, the RNIC receives a non-nilext request `get(k)` and places it in the third buffer in RQ. **(e)** After the CPU has finished executing the first two requests, it executes the `get(k)` and returns a software response containing `v2`. Receive buffers consumed by the CPU can be pushed into the RQ again.

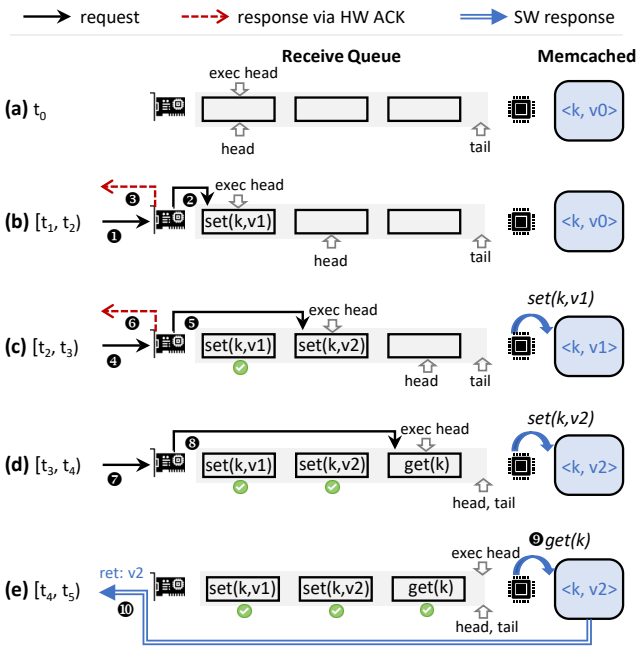


Figure 1: Memcached example with OQ. *This figure shows components at the server side, including RNIC, receive queue, CPU, and a Memcached instance.* `head` *points to the first buffer with no request;* `exec head` *points to the first buffer that is not executed by the CPU.*

Comparison with the pure software solution. One may wonder whether we can achieve the same low latency using a pure software solution that adopts deferred execution: when the server CPU meets a nilext request, it first returns a software ACK to commit, and then executes the request. However, we argue that the OQ abstraction (which achieves wire-latency nilext requests) has remarkable advantages over the pure software approach. For nilext requests, by leveraging the hardware ACKs, the OQ abstraction eliminates the server-side software delay from the critical path, not only for request execution *but also for request responses and software queuing*. In contrast, in the software approach, when the CPU is busy executing requests, the subsequent nilext requests in the RQ cannot be committed immediately and suffer from latency caused by the software queuing. We will conduct the detailed experimental comparison in §6.

Analysis of pathological access patterns. There is a pathological access pattern that makes OQ perform poorly: a client thread sends a non-nilext request *immediately after* receiving the hardware ACK of a nilext request, and both requests reach

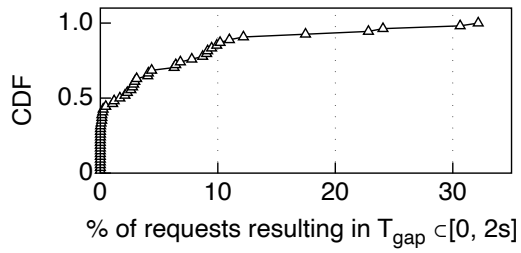


Figure 2: Analysis of 54 clusters in Twitter workloads. *For example, for a point (10, 0.852) in the figure, it means that 85.2% clusters out of 54 have less than 10% requests resulting in $T_{gap} \subset [0, 2s]$.*

the *same* RQ. In this case, the server has no chance of hiding the nilext request's execution latency to the client thread, making the non-nilext request suffer high latency.

For example, suppose the three requests in Figure 1 come from the same client thread, and the thread *immediately* sends a request when receiving the response of the previous one. We assume that the network RTT is 2μs (i.e., 1μs one-way delay), executing `set` needs 10μs, and executing `get` needs 5μs. Without the OQ, the latency of three requests are 12μs, 12μs, and 7μs, respectively. With the OQ, the latency are 2μs, 2μs, and 23μs, respectively: although the two `set` requests are fast, the `get` request has a much higher latency.

However, we argue that the above pathological access pattern is uncommon in real-world workloads. First, there are many storage servers (and each server may have multiple RQs; see §5.1), so two adjacent requests from a client often have different destinations. Second, client threads need to do computational tasks, e.g., webpage rendering (instead of sending storage requests all the time), creating time intervals between requests. As a result, client threads can easily hide the latency of executing nilext requests. Further, we analyze production traces from in-memory caching workloads of Twitter [63]. These traces contain unsampled requests that are sent to two instances (i.e., servers). We chose 1 million consecutive requests to measure the time intervals between a nilext request and the next non-nilext request issued by the same clients (called T_{gap}). Since the traces record timestamps in seconds, we can only coarsely count how many requests result in $T_{gap} \subset [0, 2s]$. Figure 2 shows the cumulative distribution across 54 clusters. In 85.2% (46) clusters, requests leading to $T_{gap} \subset [0, 2s]$ are less than 10%, which gives the OQ abstraction a great opportunity to reduce system latency.

5 Juneberry Design

Juneberry is a communication framework that implements the OQ abstraction using commodity RNICs. Juneberry provides a similar usage as RPC frameworks: The server registers request handler functions, each having a unique request type; Clients issue a request with a specified request type and parameters, to call the associated function.

Juneberry is unique in that it supports wire latency for nilext requests. Clients can mark a request as nilext. For such a request, the server CPU does not return a software response, and the client treats the hardware ACK generated by the server-side RNIC as the commit signal. In this way, the nilext request is committed with extremely low latency, removing the server CPU's overhead from the critical path.

However, when designing Juneberry, there are two obstacles that we must overcome:

- **Correct semantic with high performance**. Despite the simplicity of OQ abstraction, mapping it to the RNIC is not so straightforward. We need to carefully select hardware primitives and scrutinize them for correctness (§5.1). Moreover, while ensuring correctness, we also need to achieve high performance (§5.2).
- **Durability support with crash consistency**. The OQ abstraction lacks a description of how to handle server crashes, which is indispensable for storage systems that contain persistent states (e.g., RocksDB). Therefore, we need to augment the OQ abstraction to support durability (§5.3), including 1) making the hardware ACKs have the durability guarantee, and 2) ensuring that storage systems can recover correctly upon server crashes.

5.1 Implementing OQ Abstraction

Juneberry server leverages shared receive queues (SRQs) to implement the OQ abstraction. SRQ is a standard RDMA component supported by almost all RNICs [12]: it is proposed to reduce memory footprint by supporting multiple QPs sharing a single RQ. Juneberry exploits SRQs for another purpose: by binding multiple QPs to an SRQ, requests from different clients (i.e., QP connections) can be placed in the same queue structure, which is a basic requirement of the OQ abstraction. Juneberry runs in reliable connection (RC) mode, to enable the generation of hardware ACKs. The server pushes receive buffers to the SRQ via RDMA `RECV` commands; client threads send requests to the server's SRQ by issuing RDMA `SEND` commands.

Next, we describe how Juneberry follows the two rules of OQ (§4). In brief, SRQ naturally follows Rule 1 (i.e., NIC rule), but Juneberry relies on some software designs to follow Rule 2 (i.e., CPU rule).

According to RDMA specification [12], the receiver-side RNIC always allocates receive buffers from the head of SRQ[2]. However, if the RNIC returns hardware ACKs *before* allocation due to some radical optimizations (e.g., delayed allocation), the Rule 1 will be violated. Fortunately, by revisiting the hardware semantics of RDMA, we determine that returning ACKs before allocation is *impossible*. Concretely, in RDMA, all memory areas (including receive buffers) that can be accessed by the RNIC are registered as *memory regions* (MRs); each MR is assigned several permissions (e.g., remotely writable). Therefore, for an incoming `SEND` request,

[2] In Section 10.2.9.1 of [12]: "*When an incoming Receive Message arrives on any QP that is associated with an SRQ, the HCA uses **the next available SRQ WQE** to receive the incoming data*".

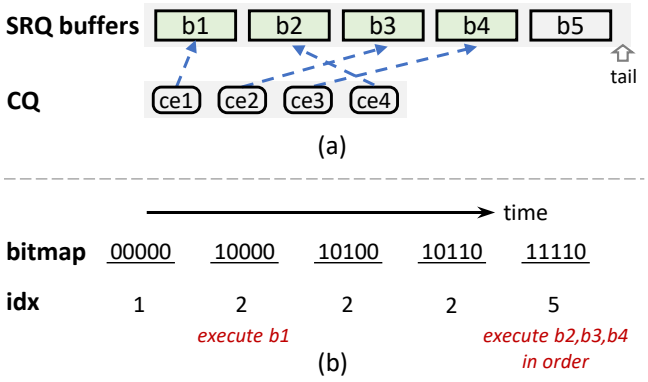


Figure 3: JUNEBERRY enforces Rule 2. ***(a)*** *The order of CEs in the CQ may not match the order of receive buffers in the SRQ.* ***(b)*** *Executing receive buffers in order with the help of the CQ and a bitmap.*

the server-side RNIC must check the permissions of the destination address before returning hardware ACKs, to enforce access control. Checking permissions indicates that the associated receive buffers have been allocated.

In existing RPC frameworks using two-sided verbs [31], the order of executing requests is the same as the order of completion entries (CEs): the server-side CPU polls the CQ to obtain CEs, then locates receive buffers using the CEs, and finally executes requests in the receive buffers. However, such a workflow (i.e., executing requests according to the order of CEs) can not enforce the Rule 2. This is because RDMA only guarantees in-order delivery for CEs generated by the *same* QP [12], but in JUNEBERRY, the SRQ's CQ accommodates CEs from different RC QPs. Hence, RDMA `RECV` commands issued to the SRQ can be completed out-of-order; in other words, when two CEs are generated by different QPs, their order in the CQ may not match the order of receive buffers (e.g., `ce3` and `ce4` in Figure 3(a)).

JUNEBERRY adopts a simple solution to enforce Rule 2, making requests in the SRQ be executed in order. As shown in Figure 3(b), JUNEBERRY maintains a bitmap and a variable `idx` for the SRQ. The bitmap tracks which receive buffers have requests; the `idx` points to the first buffer that is not executed by the CPU. CPU executes requests with the following three steps: ❶ Polls the CQ to obtain CEs and sets associated bits in the bitmap; ❷ If the `idx`-th bit is set, executes the request and advances `idx`; ❸ Repeats the step ❷ until the `idx`-th bit is clear. For example, in Figure 3(b), upon obtaining `ce4` from the CQ, the CPU executes the request in `b2` (since `idx` equals 2), and then executes the next contiguous sequence of requests in the bitmap in order (i.e., `b3` and `b4`).

Scaling to multiple CPU cores. Until now, we have assumed that there is only a single SRQ at server side. However, it is impractical when considering the multi-core architecture of servers. First, when multiple threads operate the same SRQ, the scalability issue happens. Second, when multiple threads execute requests concurrently, producing the same results as sequential execution (for Rule 2) is intricate and low-performance: it involves dependency tracking and resolution, thus inducing expensive cross-thread synchronization.

JUNEBERRY uses data partitioning to support multi-core scaling. Specifically, JUNEBERRY creates multiple SRQs at the server side, each being associated with a worker thread. Each worker thread manages an *exclusive* set of datasets. Such data partitioning is easily achieved in many storage systems [32, 33, 37, 39] such as key-value stores, since they adopt a shared-nothing architecture to improve performance. Clients in JUNEBERRY need to send requests (via RDMA `SEND`) to correct server-side worker threads according to partition scheme. For example, in the Memcached case, clients hash the target key in a `set`/`get` request, and send the request to the SRQ managed by the i-th worker thread, where i = `hash(key) % #(worker threads)`.

5.2 Optimizing Performance

JUNEBERRY materializes the OQ abstraction by leveraging SRQ and adopting software designs (§5.1). However, it poses two performance issues: 1) operating SRQs is inefficient for RNIC; 2) a large number of RC QPs harm system scalability. For each issue, this subsection first explains its causes and negative effects, and then introduces our solutions.

5.2.1 Streamlining RNIC Prefetching on SRQ

The lifecycle of work queue elements (WQEs). When creating an RQ/SRQ, the RNIC driver allocates a contiguous memory region that contains multiple WQEs, which can be accessed by RNICs (via DMA). When pushing a receive buffer to the RQ/SRQ via `RECV` commands, the CPU fills in a *free* WQE with metadata (including the address of the receive buffer), making it *valid*. A valid WQE becomes free when the CPU obtains the associated CE from CQs, which indicates that the WQE has been consumed by a `SEND` request.

The RNIC prefetches valid `RECV` WQEs via DMA and caches them in its SRAM, to accelerate the processing of incoming `SEND` requests. Such prefetching is efficient for standard RQs, since logically contiguous WQEs are also physically contiguous at memory locations, making a single prefetch operation (i.e., a DMA operation) be capable of fetching multiple valid WQEs at a time. For example, as shown in Figure 4(a), receive buffers `b1`, `b2`, `b3` are pushed into the RQ in order, and three WQEs allocated for them are memory-contiguous.

However, SRQ counteracts the benefit of RNIC prefetching due to its structure feature. Specifically, as described in §5.1, the valid WQEs in an SRQ can be completed out-of-order, thus creating fragmented free WQEs like holes. Subsequent `RECV` commands often fail to allocate WQEs in a memory-contiguous manner. As a result, valid WQEs in an SRQ are connected using a link list. For example, as shown in Figure 4(b), the SRQ organizes valid WQEs using the link list `wqe3` → `wqe1` → `wqe5`. Due to the link list structure of WQEs, the RNIC is more susceptible to cache misses (thus stalling in-

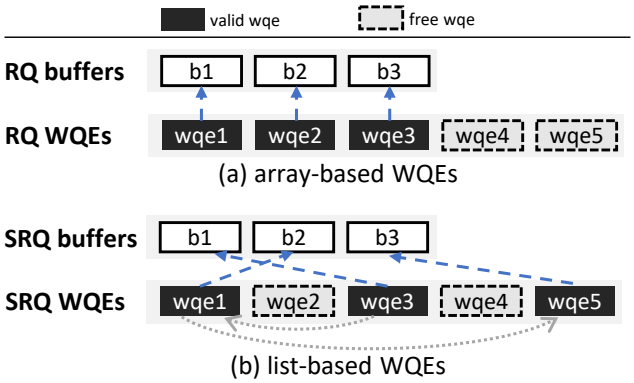


Figure 4: Array-based WQEs vs. list-based WQEs. *In these two figures, receive buffers b1, b2, and b3 are pushed into the RQ/SRQ in order, and the RNIC driver allocates WQEs for them.*

coming traffic), and needs more DMA operations to prefetch WQEs with the pointer-chasing access pattern, both degrading the network performance. We validate it using a simple experiment, where a client launches 24 threads to send 64B messages to a remote server. The remote server uses one CPU core to receive messages and respond with 64B messages; each client thread allows 4 outstanding requests. The server can deliver 2.91Mops/s when using RQ, but the performance drops to 2.24Mops/s when using SRQ. A recent work observes a similar performance degradation on Ethernet NIC that adopts list-based receive (Rx) rings [49].

JUNEBERRY improves the performance of SRQ by allowing a single WQE to accommodate multiple messages, so that a prefetch operation can serve multiple `SEND` requests. This is achieved by exploiting the multi-packet (MP) feature in advanced RNICs [20,56]. For an MP RQ/SRQ, multiple `SEND` requests can be sequentially written to the same receive buffer, and the address of each request is aligned to a k (e.g., 64) byte offset, where k is pre-defined[3]. The MP feature is proposed to reduce memory footprint, but JUNEBERRY uses it to streamline RNIC prefetching and mitigate network traffic stall. In addition, with MP SRQ, JUNEBERRY server can post a large receive buffer (e.g., 64KB) to amortize the overhead of filling in WQEs and ringing doorbells, saving CPU cycles.

5.2.2 Reducing the Number of QPs

Recall that in JUNEBERRY, a client thread can send requests to the MP SRQ of any worker thread according to partition scheme (§5.1), which indicates that there are RC connections between every client thread and every worker thread (RC QPs only support one-to-one connections). Concretely, the server maintains $C{\times}Ct{\times}St$ RC QPs in total, where `C` is the number of client machines, `Ct` is the number of client threads per client machine, and `St` is the number of worker threads in the server. However, it is well-known that RNIC cannot cache states for a large number of QPs due to its size-limited SRAM [18,31,45,57]. Upon cache misses, RNIC issues DMA operations to fetch QP states from host memory, stalling the incoming network traffic and thus degrading performance.

We reduce the number of QPs in the JUNEBERRY server to $C{\times}St$ by employing a client-side delegation mechanism. Specifically, a client machine creates St QPs, each connecting to a remote worker thread. Note that each QP uses a separate context to avoid false synchronization between client threads [58]. Each QP is managed by one client thread, which is called *owner* of the QP. When a client thread is about to send a request to a worker thread, it identifies the owner of the target QP and publishes the request in a reserved slot. The owner scans slots and emits associated requests (including requests generated by itself) to the network using the dedicated QP. The owner is also responsible for processing responses from the QP and delivering these responses to initiators.

5.2.3 Alternative Optimization Techniques

We discuss alternative optimization techniques that can address the above two performance issues.

Array-based SRQ. An alternative approach to streamlining RNIC prefetching is to make RNIC support array-based SRQ (by modifying the firmware of RNIC ASIC). This approach also requires us to modify the driver of RNIC. Compared with array-based SRQ, MP SRQ used by JUNEBERRY has an additional advantage: mitigating CPU overhead by reducing the number of doorbells.

eXtended RC (XRC). XRC [4] is an Infiniband transport service supported by Mellanox RNIC. When using XRC, an RC QP can deliver messages to multiple SRQs in a remote machine. If we adopt XRC in JUNEBERRY, the number of QPs in the server will decrease to $C{\times}Ct$, the total number of client threads in the cluster. Compared with the client-side delegation mechanism (which has $C{\times}St$ QPs at the server side), XRC has a scalability advantage when the server launches a large number of threads.

5.3 Supporting Durability

JUNEBERRY supports durability by leveraging emerging persistent memory (PM) technologies. This subsection first gives a brief background on PM, and then describes how JUNEBERRY uses it for durability and crash consistency. Here, we assume that the back-end storage systems of JUNEBERRY are crash-consistent (unlike Memcached).

A primer on persistent memory (PM). PM is a new class of storage devices that can guarantee persistence with DRAM-like features. Specifically, PM can expose its storage space as physical addresses to CPUs and other devices (e.g., RNICs). Therefore, it can be accessed by CPUs via `load/store` instructions with low latency, or other devices via DMA engines. There are two typical PM devices for datacenters: Intel's Optane PM [5] and Samsung's Memory-Semantic SSD [8]. The former sits in DIMM slots, while the latter is connected by the CXL link. The performance of Optane PM is well-known [62]: it can deliver 2GB/s writes and 6GB/s reads per DIMM, and

[3] We set k to 64, which is the minimum value that our RNICs support, thus reducing internal fragmentation caused by data alignment.

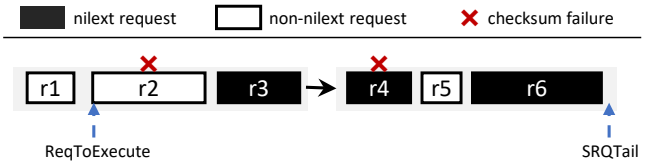


Figure 5: An example of scanning an SRQ upon recovery.

it has the same write latency and 3× higher read latency compared to DRAM. Memory-Semantic SSD is not currently available. Although the CPU cache is volatile, recent CPUs put it into the persistent domain via eADR [3,26,65] or Global Persistent Flush (GPF) in CXL [2], which simplifies the use of PM. With eADR or GPF, upon server crash, the data in the CPU cache will be flushed to PM safely.

Persistent hardware ACKs. We first place SRQ receive buffers in PM, so that requests in these buffers can survive power outages. Recall that JUNEBERRY uses hardware ACKs as commit signals of the nilext requests, to achieve wire latency. Yet, when a client receives the hardware ACK for an RDMA `SEND` command, it only ensures that the `SEND` is allocated a destination address in the SRQ; the DMA write operation in the server-side RNIC may still be in progress, i.e., the `SEND` data may not reach PM successfully. To make the hardware ACK have the persistence guarantee, JUNEBERRY drains DMA operations of `SEND` commands in the server-side RNIC. It does so by letting client threads issue an extra RDMA `READ` (with 1-byte size) to the same QP after every `SEND` command [28, 60]: the PCIe read triggered by the RDMA `READ` will flush the previous PCIe writes. In this way, the hardware ACKs of `READ` ensure the persistence of nilext requests.

In addition, JUNEBERRY performs two optimizations. First, client threads combine the `SEND` command and `READ` command in one round trip, and remove CEs of `SEND`, since the receiver-side RNIC executes these two commands in order [12]. Second, the server disables Data Direct I/O (DDIO), to let the RNIC bypass the CPU cache when performing DMA operations. It can reduce PM write amplification caused by granularity mismatch between a cache line (64B) and a PM internal write (e.g., 256B in Optane PM) [28].

Recoverable SRQ. Simply putting receive buffers in PM does not imply that we can recover requests from the SRQ after power failure. JUNEBERRY devises an identifiable request format to address it. Specifically, each request begins with a 64-bit `checksum` and a 32-bit `size`; `checksum` covers the whole request and `size` is the request length, both being filled by client threads. Moreover, each worker thread persistently maintains two variables for its SRQ in PM: `ReqToExecute` and `SRQTail`. `ReqToExecute` points to the first request not yet executed: it is updated after the worker thread executes a request. `SRQTail` records the last receive buffer in the SRQ; it is updated after the worker thread issues a `RECV` command.

Upon recovery, the server scans SRQs to identify valid requests. Recall that JUNEBERRY leverages the multi-packet feature of RNICs, so a receive buffer may contain multiple requests. Each request has a start address that is aligned to a pre-defined value (§5.2), which simplifies the scanning. For each SRQ, a recovery thread scans receive buffers from `ReqToExecute` to `SRQTail`. When an address results in checksum failure, the thread jumps to the next aligned address in the current receive buffer or the next buffer, continuing to identify valid requests. Finally, as shown in Figure 5, all valid requests (i.e., `r3`, `r5`, and `r6`) are marked. Among them, nilext requests may have returned hardware ACKs as commit signals; therefore, for crash consistency, the recovery thread executes these nilext requests in order (i.e., `r3` and `r6`). Note that we do not need to execute non-nilext request `r5`, because it is uncommitted and thus is concurrent with the subsequent requests in the SRQ (i.e., `r6` in this example).

There is a corner case: `ReqToExecute` points to a valid request that is nilext and non-idempotent (e.g., fetch-and-add without returning values). The recovery thread *cannot* determine whether this request has returned a hardware ACK. It also *cannot* determine whether this request has been executed, since executing the request and updating the `ReqToExecute` are not atomic. As a result, the recovery thread *cannot* decide whether to execute the request (may cause duplicated execution) or skip it (may miss a committed request).

To address the above issue, we make all non-idempotent requests *software controllable* (i.e., have the same workflow as non-nilext requests): the worker thread ❶ executes a request, ❷ updates `ReqToExecute`, and ❸ finally returns a software response; client threads use the software response as the commit signal. Upon recovery, the recovery thread can skip the non-idempotent (or non-nilext) request pointed by `ReqToExecute`, since the client thread has not received the request's response, according to the order of ❶ ❷ ❸.

6 Evaluation

We seek to answer the following questions in our evaluation:

- How do optimization techniques proposed in §5.2 affect JUNEBERRY's throughput? (§6.2)
- How does JUNEBERRY perform when applying to in-memory storage systems like Memcached? (§6.3)
- How does JUNEBERRY perform when applying to persistent storage systems like PMemKV? (§6.4)

6.1 Experimental Setup

We conduct experiments in a cluster having one server machine and 8 client machines. The server machine is equipped with two 18-core Intel Xeon Gold 6240M CPUs, 192GB DRAM, and 1.5TB Optane PM (three Optane DIMMs per CPU). Each client machine is equipped with two 12-core Intel Xeon E5-2650 CPUs and 128GB DRAM. Each machine installs a 100Gbps Mellanox ConnectX-5 RNIC; all RNICs are connected to a Mellanox 100Gbps IB Switch. To avoid the severe NUMA impacts on PM [35, 66], we only use one

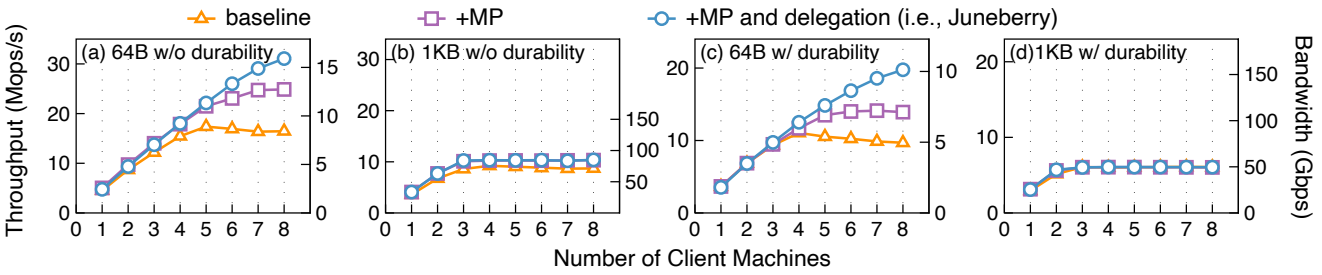


Figure 6: Throughput/Bandwidth with the varying number of client machines.

CPU and the associated DRAM/PM resources in the server.

There are two versions of our system: JUNEBERRY and JUNEBERRY-D. The latter supports durability (§5.3), so it places the server-side receive buffers in PM and disables DDIO. Without explicit mention, each MP SRQ consists of 48 receive buffers of 16KB.

Target comparisons. In §6.3 and §6.4, we will compare JUNEBERRY with three systems:

- **RawRPC.** RawRPC uses the same codebase as JUNEBERRY, including optimizations such as MP features and delegation (§5.2). Like traditional RPC frameworks, in RawRPC, the server CPU handles requests and then returns responses.
- **DeferredExec.** DeferredExec also uses the same codebase as JUNEBERRY. It optimizes nilext requests using a pure software solution: when processing a nilext request, the server CPU first returns a software ACK to commit and then executes it.
- **eRPC.** eRPC [29] is a state-of-the-art RPC framework. It uses unreliable datagrams for network communication and implements retransmission and congestion in software.

6.2 Microbenchmarks

In this experiment, we use microbenchmarks to demonstrate the efficacy of two techniques proposed to improve JUNEBERRY's performance: MP features and client-side delegation. Each client machine launches 24 threads: each client thread keeps one request in flight, and sends the request to a randomly-chosen remote worker thread. The server machine launches 18 threads: when a worker thread receives a request, it replies using the same size message. Figure 6 shows the system throughput with the varying number of client machines, under different settings (i.e., small requests vs. large requests, w/o durability vs. durability).

Small requests without durability. We make the following observations from Figure 6(a). First, with 8 client machines, MP improves the throughput of 64B requests from 16.45Mops/s to 24.86Mops/s (i.e., 1.51×). This is because by accommodating multiple requests in a single receive buffer, MP greatly mitigates the overhead of RNIC prefetching on list-based SRQ WQEs and thus lowers the probability of RNIC cache misses. Second, client-side delegation further increases throughput by 1.25×, since it reduces the number of RC QPs from 3456 (i.e., 8×24×18) to 144 (i.e., 8×18); in this way, the QP states become smaller, alleviating the cache thrash caused by RNICs' size-limited SRAM. Third, these two techniques make JUNEBERRY scale well; without them, JUNEBERRY can not scale to more than 5 client machines.

Large requests without durability. From Figure 6(b) we can see that even with large requests (i.e., 1KB), the baseline can only achieve 71Gbps due to inefficient RNIC prefetching on SRQ WQEs. When using multi-packet (MP) SRQ, the bandwidth increases to 85Gbps. What needs to be explained here is why JUNEBERRY does not reach the raw performance of RNICs (i.e., 100Gbps). We suspect the cause is the hardware limitation of the SRQ, because when we replace SRQs with standard RQs, the bandwidth can approach 100Gbps.

Small requests with durability. We make two observations from Figure 6(c). First, as with DRAM, the two techniques significantly boost the throughput of small requests when using PM: in the case of 8 client machines, MP and delegation improve throughput by 1.44× and 1.42×, respectively. Second, supporting durability brings 36.40% throughput degradation (compared with Figure 6(a)): from 31.07Mops/s to 19.76Mops/s. This is because 1) every RDMA SEND command is followed by an RDMA READ command for durability, consuming the IOPS of the server-side RNIC; 2) handling PM-resident requests consumes more CPU cycles, since PM has 3× higher read latency than DRAM.

Large requests with durability. As shown in Figure 6(d), for large requests, the limited bandwidth of PM (6GB/s for 3 Optane DIMMs) becomes the performance bottleneck.

6.3 In-Memory Caching: Memcached

In this experiment, we use Memcached (version: 1.6.19) as the back-end storage system. We remove network-related parts in Memcached's codebase and produce a local library that can be linked to JUNEBERRY server. We spawn 24 threads per client machine (8 client machines in total); client threads generate requests at a given rate with Poisson arrivals. The server machine launches 16 threads to handle requests and reserves 2 threads for background tasks (e.g., LRU maintainer).

We select four representative in-memory caching work-

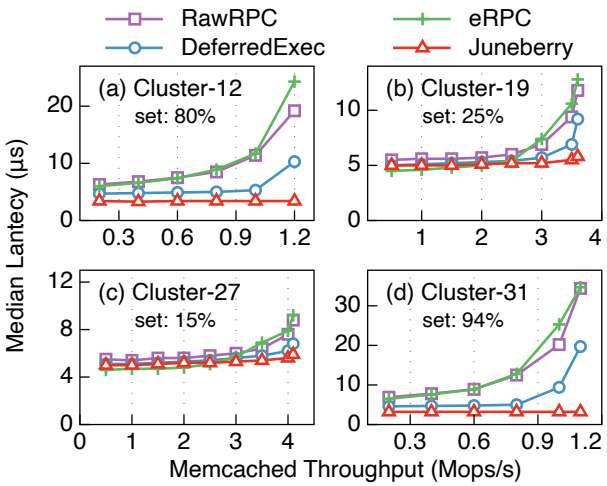


Figure 7: Median latency vs. throughput. *We use Memcached (version: 1.6.19) as the back-end storage system.*

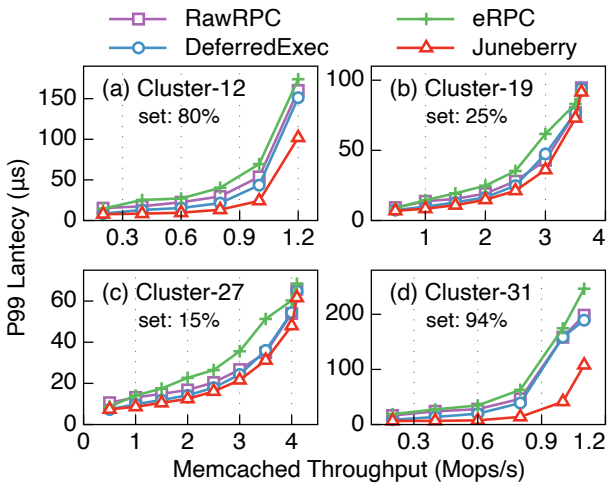


Figure 8: P99 latency vs. throughput. *We use Memcached (version: 1.6.19) as the back-end storage system.*

	ratio of `set`	key size	value size	Zipf alpha
Cluster-12	80%	44B	1030B	0.3048
Cluster-19	25%	42B	101B	0.735
Cluster-27	15%	66B	8B	1.065
Cluster-31	94%	41B	15B	0

Table 1: Settings of in-memory caching workloads.

loads from Twitter [63], as shown in Table 1. Among them, Cluster-12 and Cluster-31 are write-intensive; Cluster-19 and Cluster-27 are read-intensive.

Figure 7 shows median latency under varying throughput. All systems achieve the same peak throughput since the back-end storage engine is the throughput bottleneck. We can make the following observations.

First, under peak throughput, compared with `RawRPC`, JUNEBERRY achieves 82.30%/50.74%/32.89%/90.70% lower median latency in Cluster-12/19/27/31 workloads. This is because JUNEBERRY makes nilext requests (i.e., `set` requests) wire-latency by leveraging hardware ACKs as commit signals. For workloads containing more nilext requests (e.g., Cluster-12 and Cluster-31), the latency reduction is more pronounced.

Second, by deferring execution of nilext requests until after returning responses, `DeferredExec` reduces median latency of `RawRPC`. However, `DeferredExec` is still outperformed by JUNEBERRY: for example, when approaching peak throughput under Cluster-12, `DeferredExec` has 3.03× higher median latency compared with JUNEBERRY. This is because when worker threads of `DeferredExec` are executing requests, they cannot immediately return software responses for nilext requests residing in RQs. In contrast, JUNEBERRY offloads the task of returning nilext requests' ACKs to the RNIC hardware, avoiding software queuing caused by the busy CPU. As a result, the median latency of JUNEBERRY is not only low but also stable under different degrees of loads. This demonstrates that in the microsecond era, hardware/software co-design (e.g., the OQ abstraction) is necessary to further realize latency reduction for storage systems. It is also worth noting that compared to `DeferredExec`, JUNEBERRY reduces the number of network packets used for returning responses.

Third, `eRPC` suffers much higher latency than JUNEBERRY at peak throughput, but when workloads are read-intensive and not heavy, it median latency is slightly lower. For example, as shown in Figure 7(c), when the throughput is less than 2.5Mops/s under Cluster-27, `eRPC` has 100~400ns lower median latency than JUNEBERRY. This is because at this point, software queuing does not affect latency, but the client-side delegation in JUNEBERRY incurs extra software overhead such as cross-thread communication.

Figure 8 shows 99th percentile (P99) tail latency under varying throughput, from which we can make the following observations. First, under write-intensive workloads Cluster-12 and Cluster-31, JUNEBERRY yields a P99 latency that is up to 32.43% and 42.86% lower than the second best system (i.e., `DeferredExec`), respectively. This is because the OQ abstraction enables fast commit of nilext requests, which represent a large fraction of these two workloads.

Second, under read-intensive workloads Cluster-19 and Cluster-27, the P99 latency reduction achieved by JUNEBERRY is limited. Take Cluster-19 for example, compared with `RawRPC`, JUNEBERRY decreases P99 latency from 10.6μs to 7.4μs with 0.5Mops/s throughput, and from 65.8μs to 61.8μs with 4.1Mops/s throughput (peak throughput). This is because a small fraction of `get` requests produce pathological access patterns (recall §4), i.e., the time interval between them and previous `set` requests from the same client threads is too short to hide execution latency of `set` requests, leading to high tail latency. Therefore, the improvement from the OQ abstraction is overshadowed. This also explains why JUNEBERRY is less effective in reducing tail latency than median latency.

6.4 Persistent KV Store: PMemKV

This experiment explores the performance of persistent storage systems with JUNEBERRY-D. We select PMemKV [7] as

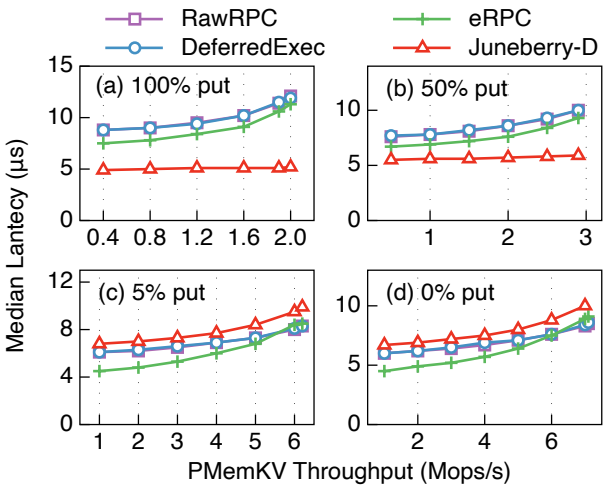


Figure 9: Median latency vs. throughput. *We use PMemKV as the back-end storage system.*

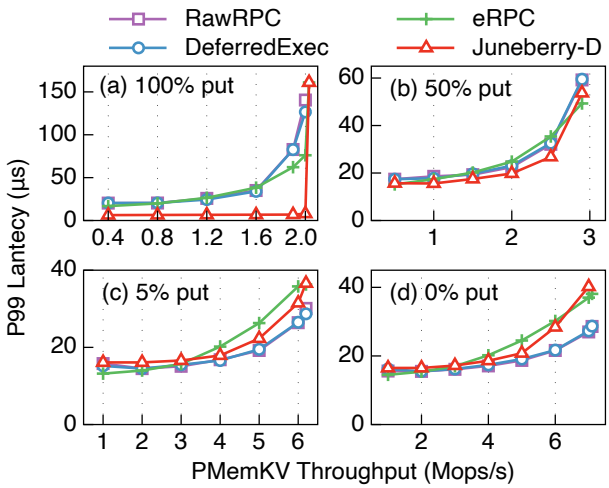


Figure 10: P99 latency vs. throughput. *We use PMemKV as the back-end storage system.*

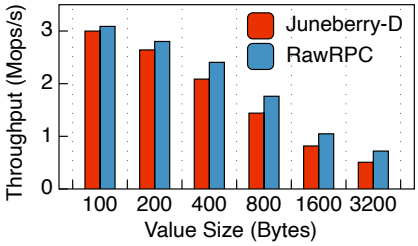


Figure 11: Throughput with different value size. *We use PMemKV as the back-end storage system. The* `put` *ratio is 50%.*

the backend; it is a persistent KV store that maintains objects in PM. The experiment setup is the same as in §6.3, except that the server launches 18 threads. We use YCSB-like workloads with four types of `put`/`get` ratio: write-only (100% `put`), read-only (100% `get`), write-intensive (50% `put` and 50% `get`), read-intensive (5% `put` and 95% `get`). We set the object size to 91-byte, which is the average object size in Meta's largest KV system, i.e., ZippyDB [15]. Workloads follow a Zipf 0.99 access distribution.

Figure 9 depicts median latency with increasing throughput, from which we make two observations. First, Juneberry-D reduces median latency by 57.08%/56.33%/53.92% against `RawRPC`/`DeferredExec`/`eRPC` under write-only workloads; 40.83%/40.83%/36.71% under write-intensive workloads (i.e., 50% `put`). By leveraging persistent hardware ACKs along with PM-resident recoverable SRQ, Juneberry-D can quickly commit a nilext request without sacrificing crash consistency.

Second, Juneberry-D is inferior to other systems under read-intensive (i.e., 5% `put`) and read-only workloads. Specifically, compared with `RawRPC`, Juneberry-D has 1.19× (1.6μs) and 1.20× (1.7μs) higher median latency. This is because in Juneberry-D, the server ① disables DDIO, harming the communication latency between RNICs and host memory, and ② uses PM-resident receive buffers, leading to higher latency spent by the CPU in fetching requests. In addition, under read-only workloads, Juneberry-D has about 1.5% lower peak throughput compared with other systems.

Figure 10 further shows P99 latency of PMemKV. Under write-only workloads (Figure 10(a)), when the throughput is 2.0Mops/s, Juneberry-D exhibits 94.38%/93.76%/89.61% lower P99 latency against `RawRPC`/`DeferredExec`/`eRPC`. However, when the throughput offered is slightly higher than Juneberry-D can handle, its tail latency increases dramatically. This is because when all requests are nilext, the server-side SRQ can easily have no free buffers, causing the server-side RNIC to generate the receive not ready (RNR) error.

When workloads are read-intensive (5% `put`) and read-only, Juneberry-D has up to 1.28× and 1.49× higher P99 latency due to PM receive buffers and disabled DDIO. Combining Figure 9 and Figure 10, we can summarize that when supporting durability, the OQ abstraction brings latency benefits under write-intensive workloads, but incurs latency overhead under read-intensive workloads.

In Juneberry-D, PM-resident receive buffers consume PM write bandwidth, which inevitably affects the performance of back-end storage systems using PM (recall that PM write bandwidth is limited). We conduct an experiment to show it. We use 8-byte keys and vary the value size; the ratio of `put` requests is 50%. Figure 11 presents the result. When the value size is small, the throughput degradation of Juneberry-D is acceptable: only 3.0% with 100B values. However, when values become larger, the size of `put` requests increases and thus more PM write bandwidth is consumed on server-side SRQ, reducing the available PM bandwidth for the back-end storage system. As a result, with 3200B values, `RawRPC` delivers 1.42× higher throughput against Juneberry-D. Fortunately, small objects are very common in real-world workloads [15, 43, 63], so Juneberry-D can achieve wire-latency nilext requests at a small cost in throughput in many cases. In addition, if the back-end storage system uses a storage device that is not a PM (e.g., SSD or HDD), Juneberry will not induce throughput degradation.

7 Discussion

Supporting durability without Optane PM. In our experiments, we use Optane PM to store requests for durability. However, Intel has fully killed off its Optane Memory business. Here, we provide two alternative solutions:

- **CXL-based SSD.** By attaching SSD devices on CXL links, CXL-based SSD can expose load/store interfaces to RNICs. Thus, we can place SRQ buffers on CXL-SSD and the RNIC will DMA requests to it, guaranteeing data durability. Some CXL-based SSD products (e.g., Samsung's Memory-Semantic SSD [8]) will enter the market in the near future.
- **UPS-backed DRAM.** Uninterruptible power supply (UPS) is a type of power system that makes DRAM durable. Specifically, with the help of UPS, the contents of DRAM will be written to SSD when the power fails [19]. Compared with Optane PM, USP-backed DRAM will not induce extra latency overhead to JUNEBERRY. Moreover, since the size of the SRQ buffers is small, the battery provisioning costs of UPS are limited for JUNEBERRY.

Implementing JUNEBERRY with different technologies. JUNEBERRY exploits hardware ACKs generated by RNICs to enable fast commit. It can also be implemented on other NICs supporting hardware ACKs [46, 50], such as FlexTOE [50], a TCP offload engine (TOE) on SmartNICs. Moreover, the idea of fast commits can be applied to other levels of the network stack. For example, for Linux socket-based applications, we can return a software ACK in the Linux kernel by implementing an in-kernel eBPF hook via XDP. By doing so, the latency overhead caused by the context switch can be avoided.

8 Related Work

RDMA communication. Existing works use RDMA for inter-server communication in two ways: 1) implementing RPC upon RDMA verbs [27, 29–31, 52] and 2) directly accessing remote memory via one-sided verbs [9, 18, 55, 59, 64]. RPC enables complicated server-side execution (e.g., index traversal) in one round trip. One-sided access can bypass remote CPUs and thus achieve low latency, but is restricted to simple read and write operations. Thus, retrieving data using one-sided verbs in a storage system typically involves multiple round trips due to indexing and concurrency control (e.g., Sherman [55] and CliqueMap [51]). JUNEBERRY based on the OQ abstraction provides a new way for inter-server RDMA communication: the server CPU can execute requests like RPC, supporting general operations within one round trip, and nilext requests are committed with remote CPU bypass, achieving extremely low latency.

NIC assisted systems. Recent works leverage SmartNICs and customized NICs to accelerate system software [14, 17, 25, 34, 36, 40, 41]. Among them, RPCValet [17], Dagger [36], and nanoPU [25] targets RPC systems: they offload tasks of the RPC stack to NICs, such as request scheduling and (de)serialization, and use memory interconnects (instead of PCIe) for CPU-NIC communication to lower latency. JUNEBERRY can also be viewed as leveraging NICs to accelerate RPC systems: it relies on NICs for providing a fast path to commit nilext requests. However, different from the above works, JUNEBERRY does not require hardware modification; it repurposes hardware ACKs that exist on commodity NICs supporting reliable data transfer (e.g., RNICs).

Combining RDMA and PM. RDMA and PM can offer low-latency data transfer and storage, respectively, so researchers are actively exploring how to combine them efficiently [10, 28, 42, 53, 53, 60, 61]. Kalia et al. [28] and Wei et al. [60] present a set of guidelines to optimize the interaction between RDMA and PM. Octopus [42], Orion [61], and Assise [10] are distributed file systems powered by RDMA and PM. They support one-sided RDMA access to PM-resident file data, thus mitigating CPU overhead and avoiding unnecessary data copying. Clover [53] is a PM-based key-value store that uses RDMA networks. Clients access objects in PM via RDMA `WRITE` and `READ`, and resolve concurrent conflicts via RDMA atomic verbs. JUNEBERRY adopts a different approach to combine RDMA and PM: it places receive buffers in PM to persistently store incoming RDMA `SEND` requests, rather than directly accessing PM-resident data via one-sided RDMA.

Nil-externality. Ganesan et al. identify nil-externality [21], a property for interfaces if they do not externalize effects. By exploiting nil-externality, they propose nilext-aware replication, which improves replication performance by lazily ordering and executing nilext updates. The design of OQ abstraction gets inspiration from their work. Several works on storage systems also perform early commit and asynchronous execution. Xsyncfs [47] commits a file system operation without performing disk I/O, and defers execution until external output is generated (e.g., send data to the screen or network). ScaleDB [44] commits a transaction without updating the global range index, and introduces a concurrency control mechanism for transaction serializability. JUNEBERRY is designed for networked environments and can be used for many storage systems. Moreover, JUNEBERRY's early commit is achieved by exploiting hardware ACKs, thus eliminating software delay from the critical path.

9 Conclusion

In this paper, we propose Ordered Queue (OQ) abstraction, which accelerates nilext requests by leveraging NICs's hardware ACKs for fast commit. By doing so, a nilext request can be committed in a single round trip with remote CPU bypass, achieving extremely low latency. We materialize OQ abstraction by designing JUNEBERRY, a communication framework running on commodity RDMA NICs. JUNEBERRY uses a set of techniques to improve performance and incorporates persistent memory to support durability. JUNEBERRY significantly lowers the latency of storage systems under write-intensive workloads. This work demonstrates that making storage software have visibility into network-level knowledge can bring performance benefits.

Acknowledgements

We sincerely thank our shepherd Mariano Scazzariello for helping us improve the paper. We also thank the anonymous reviewers for their feedback, Junru Li for sharing his experience on networking. This work is supported the National Natural Science Foundation of China (U22B2023, 62472242, 62402204), Young Elite Scientists Sponsorship Program by CAST (2023QNRC001), and Postdoctoral Fellowship Program of CPSF (GZC20231296).

References

[1] Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 49–67, Boston, MA, April 2023. USENIX Association.

[2] Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/download-the-specification, 2024.

[3] eADR. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2024.

[4] eXtended Reliable Connection (XRC). https://downloads.openfabrics.org/Media/SC07/2007_SC_Nov_XRC.pdf, 2024.

[5] Intel's Optane PM. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html, 2024.

[6] Memcached - a distributed memory object caching system. https://memcached.org/, 2024.

[7] PMemKV - Key/Value Datastore for Persistent Memory. https://github.com/pmem/pmemkv, 2024.

[8] Samsung's Memory-Semantic SSD. https://samsungmsl.com/cmmh/, 2024.

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

[10] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.

[11] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, pages 93–110, USA, 2020. USENIX Association.

[12] InfiniBand Trade Association et al. InfiniBandTM Architecture Specification Volume 1 Release 1.3 (General Specifications), 2015.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, April 2012. USENIX Association.

[14] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 228–242, New York, NY, USA, 2021. ACM.

[15] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[16] Inho Choi, Ellis Michael, Yunfan Li, Dan R. K. Ports, and Jialin Li. Hydra: Serialization-Free Network Ordering for Strongly Consistent Distributed Applications. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 293–320, Boston, MA, April 2023. USENIX Association.

[17] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of us-Scale RPCs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 35–48, New York, NY, USA, 2019. ACM.

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

[19] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[20] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association.

[21] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP'21, pages 440–456, New York, NY, USA, 2021. ACM.

[22] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 519–533. USENIX Association, April 2021.

[23] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 202–215, New York, NY, USA, 2016. ACM.

[24] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[25] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[26] Zhicheng Ji, Kang Chen, Leping Wang, Mingxing Zhang, and Yongwei Wu. Falcon: Fast OLTP Engine for Persistent Cache and Non-Volatile Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 531–544, New York, NY, USA, 2023. ACM.

[27] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 217–232, Santa Clara, CA, February 2022. USENIX Association.

[28] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 105–119, New York, NY, USA, 2020. ACM.

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

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

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

[32] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, aug 2008.

[33] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. ACM.

[34] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP'21, pages 756–771, New York, NY, USA, 2021. ACM.

[35] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 424–439, New York, NY, USA, 2021. ACM.

[36] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. ACM.

[37] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 447–461, New York, NY, USA, 2019. ACM.

[38] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, Savannah, GA, November 2016. USENIX Association.

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

[40] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, Boston, MA, April 2023. USENIX Association.

[41] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, November 2020.

[42] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[43] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 243–262, New York, NY, USA, 2021. ACM.

[44] Syed Akbar Mehdi, Deukyeon Hwang, Simon Peter, and Lorenzo Alvisi. ScaleDB: A Scalable, Asynchronous In-Memory Database. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 361–376, Boston, MA, July 2023. USENIX Association.

[45] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 212–227, New York, NY, USA, 2021. ACM.

[46] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, pages 77–92, USA, 2020. USENIX Association.

[47] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 1–14, USA, 2006. USENIX Association.

[48] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC Offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 18–35, New York, NY, USA, 2021. ACM.

[49] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with Shared Receive Rings. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 949–968, Boston, MA, July 2023. USENIX Association.

[50] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'22, pages 87–102, Renton, WA, April 2022. USENIX Association.

[51] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K.

Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, New York, NY, USA, 2017. ACM.

[53] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020.

[54] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[55] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 441–459, Boston, MA, July 2023. USENIX Association.

[57] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. SRNIC: A Scalable Architecture for RDMA NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, Boston, MA, April 2023. USENIX Association.

[58] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 233–251, USA, 2018. USENIX Association.

[59] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[60] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 523–536. USENIX Association, July 2021.

[61] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, Boston, MA, February 2019. USENIX Association.

[62] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[63] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 191–208. USENIX Association, November 2020.

[64] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, feb 2017.

[65] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. NBTree: A Lock-Free PM-Friendly Persistent B+-Tree for EADR-Enabled PM Systems. *Proc. VLDB Endow.*, 15(6):1187–1200, feb 2022.

[66] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, Carlsbad, CA, July 2022. USENIX Association.

ODRP: On-Demand Remote Paging with Programmable RDMA

Zixuan Wang, Xingda Wei, Jinyu Gu, Hongrui Xie, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

ABSTRACT

Memory disaggregation with OS swapping is becoming popular for next-generation datacenters. RDMA is a promising technique for achieving this. However, RDMA does not support dynamic memory management in the data path. Current systems rely on RDMA's control path operations, which are designed for coarse-grained memory management. This results in a trade-off between performance and memory utilization and also requires significant CPU usage, which is a limited resource on memory nodes.

This paper introduces **O**n-**D**emand **R**emote **P**aging, the first system that smartly chains native RDMA data path primitives to offload all memory access and management operations onto the RDMA-capable NIC (RNIC). However, efficiently implementing these operations is challenging due to the limited capability of RNIC. ODRP leverages the semantics of OS swapping and adopts a client-assisted principle to address the efficiency and functionality challenges. Compared to the state-of-the-art system, ODRP can achieve significantly better memory utilization, no CPU usage while introducing only a 0.8% to 14.6% performance overhead in real-world applications.

1 INTRODUCTION

Memory-intensive applications [4, 6, 9, 44] have been widely deployed in modern datacenters. However, accurately predicting their memory requirements is challenging due to fluctuations over time [37, 27]. Over-provisioning memory for host servers would significantly decrease memory utilization. To address this issue, memory disaggregation has been proposed [32, 19, 11, 37, 21, 22, 14, 36, 29, 18, 26, 43]. This architecture separates the CPU and memory resources into CPU and memory pools. Applications running on the CPU pool (CNodes) can dynamically allocate and access memory in the memory pool (MNodes) on demand, thereby improving overall memory utilization.

RDMA is a popular interconnect for bridging CNodes and MNodes. It provides efficient data path primitives, namely one-sided RDMA, which allow RNICs to directly read and write to MNode memory without involving their CPUs. This is particularly beneficial for memory disaggregation because MNodes often have limited or no CPU resources to ensure better cost efficiency [45]. Unfortunately, RDMA is not a panacea for memory disaggregation, which requires more than simple memory access. CNodes need to dynamically allocate and deallocate remote memory during runtime, which is not supported by one-sided RDMA. Even worse, dynamic allocation necessitates registering memory to the RNIC, an RDMA control path operation that is not optimized for performance and can only be performed by the CPU. As a result, naively performing memory (de)allocation in the data path would significantly degrade application performance and overwhelm the wimpy MNode CPU (§2.3).

Due to the above limitation of RDMA, previous systems [19, 11, 40, 25] have chosen to over-provision remote memory with large memory slabs (1 GB or larger) to amortize the memory (de)allocation overhead and reduce CPU usage on MNodes. However, using large memory slabs would significantly reduce the memory utilization of MNodes due to internal fragmentation. Therefore, current RDMA-based memory disaggregation systems face a fundamental trade-off between high remote memory utilization, no MNode CPU usage and high performance.

This paper demonstrates that it is possible to achieve high memory utilization, no MNode CPU usage, and high performance on commodity RDMA hardware. It describes the design and implementation of **O**n-**D**emand **R**emote **P**aging, allowing CNodes to swap pages to the MNode with 4 KB allocation granularity—the same as Linux swapping granularity for ideal memory utilization. ODRP achieves fine-grained remote memory management in the data path by efficiently offloading memory access and (de)allocation operations to the MNode's RNIC, bypassing its CPU. Our key insight is to fully exploit a less-explored chaining feature of RDMA that allows developers to *combine various one-sided primitives to offload complex functionalities onto the RNIC* (§2.4).

Although RDMA offloading has recently been proven to be Turing complete in theory [34], meaning that it can offload arbitrary programs, efficiently offloading the complex functionalities of a swap device presents challenges due to the following three issues: First, a function may require offloading multiple RDMA work requests (WRs) for implementation. However, offloading performance decreases as the number of WRs increases. Therefore, we need to design data structures and operations that minimize the number of offloaded WRs. Second, RDMA primitives are primarily designed for networking and have limited semantics. Consequently, it is challenging or even impossible to implement certain functions efficiently with a relatively small number of WRs, such as the modulo operation. Finally, RNIC drains the offloaded WRs after execution. Re-executing them requires proactive refilling with the help of the MNode CPU. However, the MNode CPU has limited processing power and

can therefore become a performance bottleneck.

To address the aforementioned issues, first, ODRP adopts a client-assisted principle, which splits the offloaded functions and shifts part of the computation to the CNode for better performance (§4.3). Choosing how to split the functions is non-trivial, because we need to minimize the computation and storage overhead added to the CNode, as well as cope with compromised CNodes. Second, we propose utilizing the novel features offered by recent RNICs to efficiently implement several high-level primitives (referred to as meta WR) (§4.2). These include modulo and endianness conversion operators, which are critical for offloading complex functions. Finally, we discover that the RNIC only "logically" drains WRs, meaning that we can reactivate them by properly updating the RNIC's driver data structures. We are the first to propose a method to recycle executed WRs without requiring additional WRs or CPU intervention (§4.4).

We have implemented ODRP on Linux as a standard swap backend, making it transparent to applications. Compared to the state-of-the-art swap-based memory disaggregation system, namely Fastswap [11], ODRP achieves 1.72× to 12× better memory utilization with zero MNode CPU usage, and at the cost of slight performance overhead (up to 14.2%) on real-world applications including Quicksort, Kmeans, Memcached, GAPBS, and VoltDB. Moreover, it demonstrates superior performance, memory utilization, and scalability compared to other RDMA alternatives.

In summary, this paper makes the following contributions:

- We identify the issues with memory utilization and MNode CPU usage in existing RDMA alternatives for realizing swap-based memory disaggregation. (§2).
- We present a novel system design to efficiently offload memory management onto the RNIC, achieving better memory utilization and zero MNode CPU usage. (§4).
- An implementation and evaluation of ODRP to demonstrate its effectiveness and efficiency. (§5).

Although we focus on swap-based memory disaggregation, we believe that the methodology and principle summarized in our work can provide guidance for future offloading functions onto the RNIC. ODRP is available at https://github.com/SJTU-IPADS/ODRP.

2 Background and Motivation

2.1 Memory Disaggregation with RDMA

Memory disaggregation is a promising paradigm to improve overall memory utilization in datacenters [19, 11, 37, 21, 22, 14, 36, 29, 18, 26]. It separates servers into a CPU pool and a memory pool, namely CNodes and MNodes. When CNodes require additional memory, they can allocate it on MNodes and access it as needed. We focus on the asymmetric architecture, in which CNodes have powerful CPUs but limited memory, while MNodes have a large amount of memory but limited (or no) CPU power. It is worth noting that some systems adopt a symmetric architecture (i.e., a node can be both a CNode and an MNode at the same time). ODRP could be extended to support this symmetric architecture.

RDMA is a popular interconnect for CNodes to access MNode memory. One-sided primitives enable direct read, write, compare-and-swap (CAS), and fetch-and-add (FAA) operations on MNode memory, bypassing the MNode CPU. Two-sided primitives (i.e., SEND and RECV) facilitate message exchanges between CNodes and MNodes.

Our target: Swap-based memory disaggregation. We target a setup where CNodes leverage Linux's swap subsystem for memory disaggregation [19, 11, 28, 43, 39, 33, 40]. This approach offers the advantage of transparency compared to other alternatives [16, 17, 36, 21]. In this setup, if a CNode faces memory pressure, the kernel will swap out cold pages to the MNode. Likewise, if the application needs to read a swapped-out page, the kernel retrieves it from the MNode.

2.2 RDMA Alternatives to Implement Swap-based Memory Disaggregation

To support swap-based memory disaggregation, we need to implement the following functionalities on the MNode: (1) Read and write memory pages. (2) Verify the validity of the page address, i.e., whether the CNode has permission to access the page and (3) Memory allocation and deallocation. RNIC has been optimized for (1) and (2) through its data path one-sided primitives and memory region registration (MRR). However, it does not have efficient native support for (3). Specifically, MRR is a control path operation, which is inefficient and requires intensive CPU involvement. Therefore, current RDMA hardware only provides the following alternatives for existing systems to implement swap-based memory disaggregation.

- *One-sided (Static).* The MNode registers coarse-grained memory slabs (*e.g.*, 1 GB or the same size as the CNode's swap device) and allocates them to CNodes statically [11, 19, 40, 25, 33]. CNodes can directly access remote memory with one-sided primitives and require few or no memory allocation operations during runtime, making this approach highly efficient. However, this leads to poor memory utilization on the MNode due to severe internal fragmentation and the lack of elasticity.

- *One-sided (Dynamic).* To improve memory utilization and support elasticity, the MNode can register its memory as fine-grained (*e.g.*, 1 MB) memory regions (MRs) in advance and dynamically allocate them to CNodes. Specifically, if a CNode requires more (or less) remote memory, it can send an RPC message to the MNode using two-sided primitives. The MNode then allocates (or frees) an MR and returns its metadata (i.e., address and *rkey*) to the CNode. Afterwards, the CNode can use one-sided primitives to access the allocated memory like in the first approach.

Table 1: A comparison between ODRP and other RDMA alternatives. **Util**: utilization.

	Memory. Util	No CPU	Efficiency
Static	Poor	✔	Optimal
Dynamic	Medium	✘	Medium
Two-sided	Good	✘	Poor
ODRP	Good	✔	Good

Note that a freed MR must be re-registered before being allocated to other CNode for safety.

However, determining the appropriate memory allocation granularity is challenging due to the inefficiency of RDMA's control path (i.e., memory registration) and the limited CPU power of MNodes. For example, registering a 4 KB or 1 MB MR takes about 80 μs or 600 μs, respectively. Using a smaller memory slab size can improve remote memory utilization but incurs higher management overhead on MNodes and necessitates more frequent memory allocation requests from CNodes. Additionally, the limited CPU power of the MNode may become a performance bottleneck in the system.

- *Two-sided.* To minimize the cost of remote memory (de)allocation, CNodes can utilize two-sided primitives in the data path. The MNode CPU can always mediate the memory accesses from CNodes and then manage the remote memory at a fine granularity. However, this approach suffers from poor performance as two-sided primitives are less efficient than one-sided primitives [41] and requires more intensive involvement of the MNode CPU.

2.3 Issues with RDMA Alternatives for Swap-based Memory Disaggregation

This section conducts an experiment to analyze the issues of aforementioned alternatives. Detailed setup can be found in §5. Figure 1 shows the results of running increasing numbers of Quicksort tasks (each with 50% local memory) on 4 CNodes. For One-sided (dynamic), we adopt a 1 MB allocation granularity. Table 1 summarizes the comparison between ODRP and other RDMA alternatives.

Poor memory utilization of One-sided (Static). Figure 1 (a) illustrates the memory utilization of different alternatives. Memory utilization is measured as the ratio of actually used remote memory to allocated remote memory. As expected, One-sided (static) has the lowest utilization because MNode memory is pre-allocated based on the CNodes' swap space size. While One-sided (dynamic) alleviates this problem, the average remote memory utilization is still 24% lower than Two-sided due to larger allocation granularity (1 MB vs. 4 KB). It is worth noting that recent RNIC provides an on-demand-paging (ODP) MR [5] feature that provides transparent dynamic memory allocation for One-sided (static). However, this feature relies on the in-kernel driver's page fault handler for memory allocation, which is orders of magnitude slower than one-sided primitives. In our experiment, it even fails to complete the application benchmark.

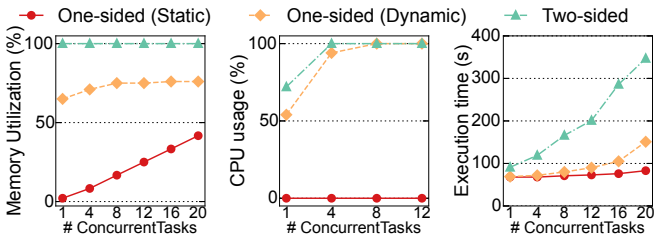


Figure 1: A comparison of RDMA alternatives in terms of (a) remote memory utilization, (b) remote CPU usage, and (c) execution time.

High MNode CPU usage of One-sided (Dynamic) and Two-sided. Figure 1(b) shows that as the number of running tasks increases, CNodes send more frequent allocation and swap requests, causing One-sided (dynamic) and Two-sided to quickly saturate the MNode CPU. As a result, the MNode CPU becomes the bottleneck and consequently slows down application performance due to request queuing.

Poor application performance of One-sided (Dynamic) and Two-sided. As shown in Figure 1(c), the average execution time for 20 Quicksort tasks in One-sided (dynamic) and Two-sided is 1.82× and 4.18× longer than One-sided (static), respectively. The performance degradation in One-sided (dynamic) can be attributed to the extra latency from waiting for memory allocation requests to complete. Moreover, Two-sided faces even more competition for the MNode CPU as every swap request needs to be queued up.

2.4 Programmable RDMA

This section presents necessary background for how to offload complex functionalities onto the RNIC.

Offloading arbitrary logic onto RNIC with WR chains. RNICs are designed to execute simple primitives posted by the CPU, known as work requests (WRs). These WRs can be linked together by recording the address of the next WR in the `next` field. When the RNIC executes the first WR, it will proceed to the next one in the chain. As a result, developers can chain multiple WRs together to offload more complex function onto the RNIC. We refer to a function offloaded to the RNIC as a *WR chain*. RDMA provides `WAIT` and `ENABLE` WRs to control execution timing. By linking a `WAIT` WR and an `ENABLE` WR after the first RECV WR in a chain, the RNIC will automatically execute the WR chain after receiving an RDMA SEND, bypassing the CPU.

An interesting property of the WR chain is its Turing completeness [34], which means we can offload arbitrary logic onto the RNIC, bypassing the CPU. Intuitively, the CAS WR provides branching, and the READ/WRITE WRs handle state reads and writes. Finally, we can implement *while* loops by reusing previously executed WR chains, i.e., after

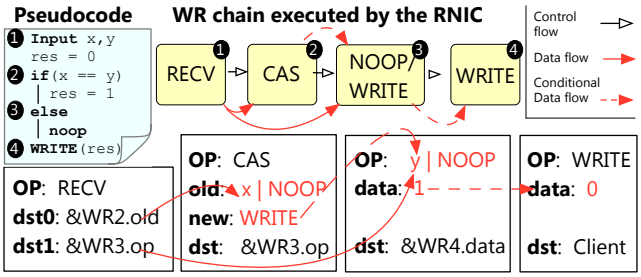


Figure 2: An illustration of how to execute the pseudocode with four RDMA WRs. For simplicity, we omit the `WAIT` and `ENABLE`.

RNIC finishes executing the last WR chain in the work queue (WQ), it wraps around to execute the first one again.

Example. Figure 2 depicts how to offload a simple conditional execution program onto the RNIC with a 4-WR chain (not counting `WAIT` and `ENABLE`). The chain checks whether x is equal to y using a CAS WR. Since the values of x and y are arguments and unknown to the WR chain before execution, the RECV WR will receive them and fill the `old` field of the CAS and the `op` field of the third WR with $x|NOOP$ and $y|NOOP$, respectively. Then, we can compare them with CAS. If x is equal to y, the CAS changes the third WR's `op` field (i.e., opcode) from NOOP to WRITE[1], causing the first branch to execute and update the final result (`WR4.data`). Otherwise, the third WR remains NOOP, and the RNIC takes no action (the second branch). Finally, the WRITE WR writes the result back to a client-side buffer.

3 ODRP Overview

Design goals. ODRP is a kernel-space swapping system that can transparently swap cold pages to the MNode. It has three main design goals: (1) *Fine-grained memory management on the MNode*, allocating and deallocating remote memory at page granularity to improve remote memory utilization; (2) *No remote CPU usage*, preventing the wimpy MNode CPU from becoming a performance bottleneck and realizing true disaggregation; (3) *Good performance*, achieving comparable performance to existing coarse-grained memory management solutions.

System architecture and high-level execution flow. Figure 3 presents the system components of ODRP. On the CNode side, ODRP provides a Linux swap backend that faithfully implements the Linux frontswap [3] API (see Table 2). On the MNode side, ODRP registers all of its provisioned memory as a large MR at initialization. The MNode maintains two types of swap metadata (detailed in §4.1): (1) it divides provisioned memory into 4 KB pages and manages them with a free page queue; (2) it maintains a translation table, initially unmapped, for each CNode that maps CNode swap addresses to page addresses. During runtime, memory

[1]Note that for the RNIC to read the updated WR, we need an `ENABLE` WR before the modified WR. For simplicity, we omit the `ENABLE` throughout the paper without losing generality.

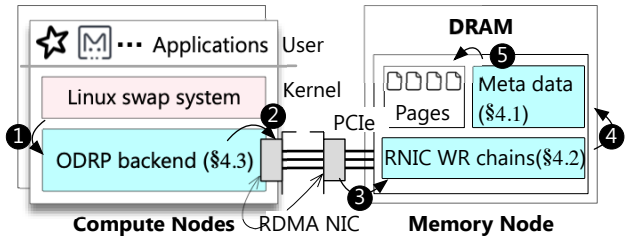


Figure 3: An overview of system components of ODRP.

pages are allocated from the free page queue and mapped in CNodes' translation tables on demand.

When the CNode's backend receives swap commands from the swap system (❶), it sends these requests to the MNode with swap addresses (❷). The MNode's RNIC then serves these requests by executing the pre-programmed logic (i.e., WR chains) stored in work queues (WQs) (❸). Each type of WR chain implements a specific high-level function (see Table 2). The WR chains leverage the swap metadata (❹) to perform address translation (swap address to page address), page access, and page (de)allocations (❺). Since all requests are handled by WR chains, there is no need for CPU intervention on the MNode.

Offloading challenges and solutions overview. While RDMA offloading presents opportunities for bypassing the CPU in fine-grained memory management, it is challenging to implement these logics efficiently and achieve full-fledged programmability due to the limited RDMA WR semantics.

C#1. Efficiency. Executing a new WR requires the RNIC to retrieve it from the work queue (WQ) in host memory via PCIe, which is relatively slow. There are two main sources of overhead that can degrade the performance of WR chains. First, more complex logic requires more WRs per chain. Second, the execution of each chain consumes all its WRs. To serve future requests, existing work [34] necessitates additional FAA and READ WRs to reactivate the executed WRs.

To minimize the number of WRs per chain, ODRP adopts a *client-assisted principle*, as detailed in §4.4. Specifically, to eliminate the need for the WR chain to calculate the address of the translation table entry (`tt_addr`) corresponding to a CNode's swap address, the MNode shares the base address of the CNode' translation table (`tt_base`) with it. Then the CNode can calculate and directly use `tt_addr` as an argument when invoking WR chains. To simplify the design of the page store WR chain, we categorize it into two cases based on whether the swap address is mapped or not (see Table 2), and have the CNode's backend determine which WR chain should be invoked. Additionally, we use the CNode as a helper to reactivate executed WR chains.

C#2. Functionality. Although RDMA offloading is theoretically Turing complete, it is non-trivial to offload complex logic onto the RNIC. First, the RNIC lacks support for the modulo operation (%), which is critical for avoiding overflow during computation. Second, the RNIC's endianness

Table 2: The CNode API and the corresponding implementation on the RNIC.

CNode frontswap API (§4.4)	RNIC Implemented Functions (§4.3)	Descriptions of the Implemented Functions
`load(swap_addr) ->page`	`load(swap_addr) ->page`	Swap in a page from the MNode with `swap_addr`.
`store(swap_addr, page)`	`mapped_store(swap_addr, page)`	Swap out a page to `swap_addr`.
	`unmapped_store(swap_addr, page)`	Allocate a page, map it to `swap_addr` and store the data in that page.
`invalidate(swap_addr)`	`invalidate(swap_addr)`	Free and unmap the page of `swap_addr` on the MNode.

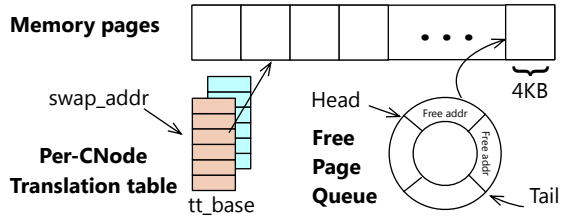


Figure 4: An illustration of the MNode-side data structures.

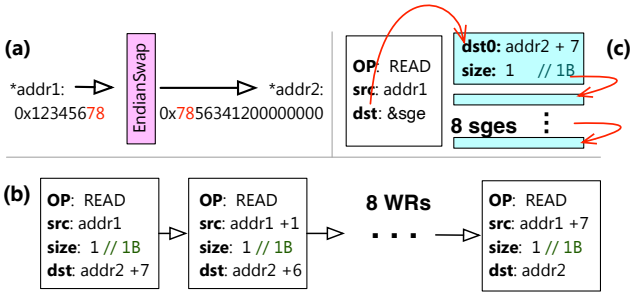


Figure 5: (a) An overview of `EndianSwap` meta WR. (b) A naive implementation using an 8 READ WRs. (c) ODRP utilizes 8 scatter/gather entries to achieve `EndianSwap` with one READ WR.

mismatches that of the host, meaning that the RNIC cannot directly use its computed value as an address to access host memory.

To address these challenges, ODRP has implemented two built-in chains as *Meta WR*s, as detailed in §4.2: one for the modulo operation and one for endianness conversion. By leveraging new RNIC features [2], we show that these operators can be supported efficiently, greatly simplifying the offloading of complex logic onto the RNIC.

4 Design and Implementation

4.1 Data Structures at the MNode

Free Page Queue. As shown in Figure 4, the MNode divides its memory into 4 KB pages and organizes them with a FIFO free page queue. During initialization, it inserts all the free pages into that queue. The queue is implemented as a fixed-sized ring buffer, determined by the number of pages the MNode can provide, and has one head pointer and one tail pointer. Each queue element is 8 bytes (8 B) in size, holding the address of a free page. When allocating a free page to a CNode, ODRP performs a dequeue operation to obtain a free page address, i.e., leverages the RDMA atomic FAA WR on the queue head pointer. When a page is freed, ODRP performs an enqueue operation, i.e., uses atomic FAA on the queue tail pointer and stores the page address in the fetched queue element.

This results in $O(1)$ complexity for (de)allocation operations, making it well-suited for the WR chain. In contrast, using other structures like bitmap to maintain free pages requires a more complex WR chain, which would harm efficiency. For example, allocating a page with a bitmap requires multiple RDMA WRs to search for a free bit and a CAS WR to update the free bit.

Translation Table (TT). ODRP provides each CNode with a swap space and maintains a TT for it, which records the mapping between swap addresses and allocated page addresses on the MNode. Each entry in the TT starts out empty. Pages are allocated from the free page queue and mapped to the TT on demand during CNode runtime. The TT is necessary to ensure memory isolation, as each WR chain can access all the MNode memory to enable fine-grained memory management and sharing (i.e., a freed page can be allocated and mapped to another CNode's swap space). Moreover, we use a single-level page table design for efficient offloading: it requires only one READ WR for address translation (see Figure 6). Finally, since the swap requests are initiated by the Linux swap subsystem in a coordinated manner (i.e., a CNode's concurrent swap requests will not target the same swap address), ODRP does not need to coordinate concurrent operations targeting the same TT entry.

4.2 Meta WRs

Before diving into the details of WR chains, we first describe the design and implementation of two meta WRs that are essential for the WR chains. Although these operations could be performed by the MNode CPU, this approach would require an interrupt from the RNIC to the CPU, which is less efficient. Therefore, we choose to implement these operations within the WR chains as meta WRs.

The modulo WR. The modulo (%) operation is crucial to prevent overflow in the enqueue and dequeue operations of ring buffer. However, without proper hardware support, it is impossible to implement the modulo operator with just a few WRs. Fortunately, recent RNICs provide an Enhanced Atomic Operations [10] feature, which includes a Masked-FAA WR that allows users to split the target value into multiple fields of selectable length for addition. By using this feature, we can mask the upper bits of the value to implement FAA with modulo support.

The endianness conversion WR (EndianSwap). Network protocols, including RDMA [1], traditionally use big-endian

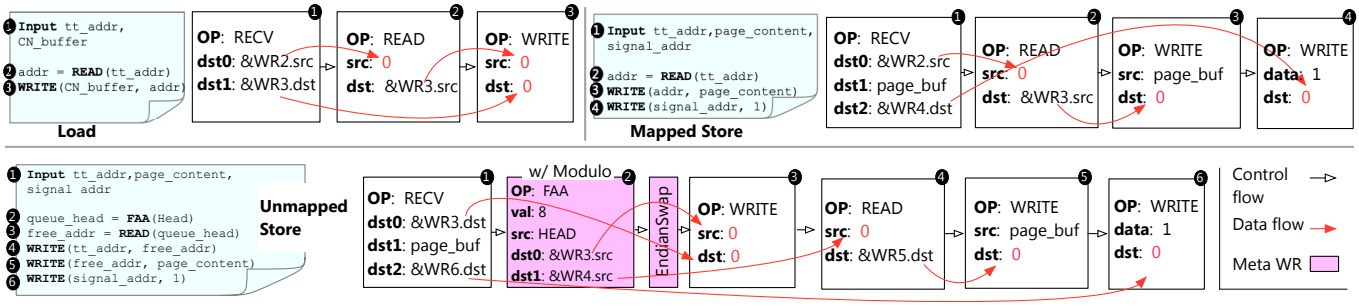


Figure 6: A simplified illustration of the WR chain design in ODRP. We omit fields like size and `WAIT` and `ENABLE` WRs. In the Load, Mapped Store, and Unmapped Store WR chains, the number of omitted `WAIT` and `ENABLE` WRs is 3, 4, and 5, respectively. The `WRITE` writes the memory from `src` to `dst`.

format, which differs from the little-endian format used by most host CPUs (*e.g.*, x86-64). This mismatch can lead to subtle issues in RDMA offloading. For example, the FAA WR performed on the host memory returns the result in little-endian format (*e.g.*, the fetched value of the queue head pointer). However, a READ WR expects its source field in big-endian format. As a result, the fetched value of the queue head pointer by FAA cannot be directly used for a subsequent READ WR without endianness conversion.

To efficiently address this issue without involving the MNode CPU, we designed an `EndianSwap` WR to simplify endianness conversion. A naive implementation would involve using N READ WRs to convert an N-bytes value. Figure 5 (b) shows an example of converting an 8-byte value from little-endian to big-endian: the first WR moves the first byte to the last byte, and so on. However, this approach is inefficient due to the excessive use of WRs. ODRP leverages the scatter-gather entry (SGE) feature provided by RNICs to reduce the number of WRs to just one. As presented in Figure 5 (c), the RNIC can scatter a continuous memory block into multiple destination buffers using SGEs. By filling the `dst` fields of multiple SGEs in reverse order, we can achieve endianness conversion with just one READ WR.

4.3 MNode-side WR Chains

Figure 6 presents the pseudocode and WR chains of offloaded functions (see Table 2). When CNodes need to swap in or out a page, they use RDMA SEND with arguments to trigger these WR chains. We assume the arguments are valid for now. Handling invalid arguments will be detailed in §4.5. Notably, instead of directly providing the swap address, we request CNodes to compute the TT entry address ($tt_base + swap_addr / PAGE_SIZE$) as an argument to all WR chains. This approach eliminates the need for an additional FAA and an EndianSwap WR in the WR chain to perform this calculation. This optimization incurs zero memory overhead for the CNode, as each CNode only needs to store its own `tt_base`.

Page load. Page load takes a TT entry address (`tt_addr`) (❶), looks up the page address stored in that entry (❷), and writes the page stored at that address back to the CNode's buffer—`CN_buffer` (❸). To look up the page address, the RECV fills the source field of the next READ with `tt_addr`. The READ then places the result of address translation (i.e., the page address on the MNode) in the source of the subsequent WRITE. Finally, the WRITE writes the page data back from the MNode memory to the destination buffer, which has been set to `CN_buffer` by the RECV.

Mapped page store. The WR chain receives a `tt_addr` and stores the received page data (`page_content`) at the address specified in that entry. The WRs used in this chain are similar to page load: the RECV (❶) fills the source field of the next READ with `tt_addr` and places the `page_content` in a temporary page buffer (`page_buf`). Then, the READ WR looks up the page address (❷) stored in the TT entry. Subsequently, the next WRITE writes the page content to that page from `page_buf` (❸). Finally, we use an additional WRITE (❹) to notify the CNode of the completion of this request by writing a magic number to a CNode-provided buffer (`signal_addr`).

Unmapped page store. This is the most complicated WR chain in ODRP, which is responsible for allocating a free page, mapping it in the TT, and storing the received page.

At a high level, this WR chain receives a `tt_addr` and allocates a page from the free page queue (❷). It then maps the allocated page address to the TT entry (❸) and stores the received page data (`page_content`) in the newly allocated page (❹–❺). To pop a free page address from the page queue, the chain uses an atomic FAA with modulo support to increment the head pointer by 8 ($head = head + 8$). The fetched old value of the head pointer (old_head) points to the allocated queue element that contains a free page address.

Note that the FAA retrieves old_head in little-endian format, which is consistent with that of a typical host CPU (e.g., x86-64). However, the subsequent WRs assume that their `src` or `dst` fields are in big-endian format. To correctly read the allocated page address ($*old_head$), we use the aforementioned EndianSwap WR to convert it to big-endian format. Given the free page address, it is straightforward to map it in the TT entry (❸) and store the `page_content` with WRITE (❹–❺). Finally, we notify the CNode with another

WRITE (❻), similar to the mapped page store WR chain.

Page invalidation. Due to space limitations, we omit this chain in Figure 6, which is similar to the unmapped page store WR chain. At a high level, we first read the address of the page to be released from the TT entry. Next, we insert the address back to the page queue by incrementing the tail pointer with FAA ($tail = tail + 8$) and using WRITE to store the freed page address to the old tail ($*old_tail = page_addr$). Finallyt, we unmap this page in the CNode's TT by setting the TT entry to zero.

Handling empty free page queue and crashed CNodes. There are two corner cases in ODRP: all memory pages on the MNode are allocated (i.e., the free page queue is empty) and a CNode crashes unexpectedly. First, we use an invalidation-based mechanism to detect the empty free page queue and abort the allocation if necessary. Specifically, after each allocation, we clear the allocated queue element ($*old_head = 0$). As a result, the first allocation attempt on the empty queue will return a null page address, causing the subsequent WRs to throw an RDMA protection error. To this end, we add a WRITE WR after the fifth WRITE WR (❺) of the unmapped store chain to set the queue element to zero. A limitation of this approach is that if such an error occurs, we need to use the MNode CPU to recover the state of the free page queue (i.e., reset the head pointer to the correct value), as page invalidation requests may continue pushing free pages into the corrupted page queue.

Second, when a CNode crashes unexpectedly, ODRP needs to recycle the resources allocated to it, including memory pages and its TT. In ODRP, the MNode detects crashed CNodes via heartbeat signals. Upon detecting a crashed CNode, the MNode scans its TT and releases the pages contained in the mapped TT entries. Then, the MNode releases the crashed CNode's TT.

4.4 CNode-side ODRP Backend

For each type of WR chain, the CNode backend has an RDMA connection to the MNode, and the MNode-side work queue (WQ) is pre-populated with corresponding WR chains to handle requests.

Swap API implementation. ODRP provides a Linux frontswap [3] backend, which operates as a swap device at the CNode. The API implementation is as follows:

- `load`: The backend calculates the `tt_addr` based on the swap address and invoke the page load WR chain by posting an RDMA SEND.
- `store`: The backend checks whether the swap address is mapped. If it is, the mapped page store chain is called; otherwise, the unmapped page store chain is invoked.
- `invalidate`: The backend invokes the page invalidation chain.

In the cases of `load` and `invalidate` operations, the swap address has a valid mapping in the TT because it must have been previously stored. However, `store` lacks this information. A naive approach would be to add extra logic in the WR chain to look up the TT and allocate a page if the target swap address is unmapped. However, it is inefficient due to the extra WRs per chain. Our observation is that a CNode's swap backend actually has full knowledge of the remote allocation status: (1) the entire swap space is unmapped initially (i.e., the whole TT starts out empty), and (2) an allocated page can only be freed by the CNode's page invalidation request, meaning the CNode controls the unmapping operation. Therefore, each CNode can maintain a bitmap to record the allocation state of each page in its swap space. With this information,`store` can determine whether the target swap address is mapped and invoke the appropriate WR chain (mapped page store or unmapped page store).

CNode-assisted MNode WR recycle. After a WR chain is executed, the WRs in it are logically dequeued from the WQ. To serve future requests, an intuitive approach is to post new WRs with the CPU. However, this would cause unexpected stalls in the RNIC's processing pipeline since the wimpy MNode CPU can only post asynchronously at a relatively low rate. To bypass the MNode CPU, ODRP makes a key observation that executed WRs are not physically erased from the WQs. Thus, by modifying the WR metadata, which is stored in WQs, we can reactivate executed WRs[2].

More specifically, to make the RNIC wrap around to the beginning of the WQ, we only need to modify the index field of the RNIC's ordering primitives, namely WAIT and ENABLE. This field indicates the index of the WR affected by these ordering primitives. Special attention needs to be paid to the RECV WR, which serves as the triggering WR of each WR chain. We discovered that to inform the RNIC that the consumed RECV is ready for re-execution, it is only necessary to modify the *doorbell record* value of the WQ.

Putting it all together, to recycle executed WRs without involving the MNode CPU, a straightforward way is to inject additional FAA and EndianSwap WRs in the chain to update the index field [34] and the doorbell record. However, this adds extra WRs to the WR chains and introduces a performance penalty. To avoid using additional WRs for index calculation and updating, we shift this calculation to CNodes. To be specific, the CNode will piggyback the calculated values with other arguments when sending requests. The RECV WR in the WR chain will use these values to update the index field and the doorbell record. As a result, no extra WR is needed for recycling.

4.5 Correctness under Benign and Malicious CNodes

We define the correctness of ODRP in two aspects. First, the TTs and the free page queue must always be consistent.

[2]RedN [34] proposed WQ recycling to implement unbounded loops. However, to the best of our knowledge, no one has found a solution to recycle RECV. RedN's open-source code also does not implement their strategy.

Specifically, these data structures are consistent if and only if (1) the free pages in the free page queue cannot be mapped to any TTs, and (2) each allocated page can only be mapped in exactly one TT. Second, requests from different CNodes must be isolated; i.e., a load or store can only access the page mapped in the CNode's own TT, even in the presence of malicious CNodes.

4.5.1 Correctness under Benign CNodes

Consistency. We informally argue that our WR chains can ensure consistency via induction. We first initialize the MNode to a consistent state by ensuring each page exists exactly once in the free page queue, and leave all the TTs empty. We then demonstrate that the execution of CNode requests will maintain the consistency of these data structures. This is straightforward when not considering concurrent requests. Executing a load or a mapped store operation, which does not modify the TT and the free page queue, naturally preserves this consistency. For the unmapped store operation, since it faithfully performs a dequeue operation, we only need to ensure that the free page address popped from the queue is consistent with the address written to the TT entry, which is obvious under our assumption of benign CNodes. Finally, page invalidation is the reverse operation of an unmapped store, so the argument is the same.

For concurrent unmapped store and invalidation operations, we use RDMA's atomic FAA to avoid race conditions, similar to other concurrent queues [31]. Concurrent load/-store operations targeted on the same `tt_addr` are inherently avoided due to the semantics of the swap subsystem. One tricky case is when the unmapped store and invalidation operations are executed concurrently. Ideally, there is no race condition between them because the ummapped store operation only modifies the queue head, while the invalidation modifies the queue tail. However, a potential race condition emerges when the queue is empty, i.e., the queue head and tail point to the same address. In this scenario, an unmapped store operation may read the queue head while an invalidation operation concurrently writes to this head. We require RDMA to provide atomic 8 B read and write for correctness, which is ensured by hardware with cache-aligned accesses [15, 16, 30, 42]. With this atomicity guarantee, the unmapped store operation will either get the latest invalidated page or a null address. Both cases are correct.

Isolation. Since all load and store requests can only access pages mapped in the CNode's own TT, requests from different CNodes are isolated as long as their TTs are consistent, which has been demonstrated above.

4.5.2 Handling Malicious CNodes

Security model. We assume MNodes are trustworthy, while CNodes can be compromised, the same as existing systems [11, 19, 22]. A malicious CNode might send illegal requests with invalid arguments (e.g., invalid `tt_addr`), or try to exhaust the memory resources of the MNode by continuously sending unmapped store requests. We show that malicious CNodes requests will be rejected by the MNode.

Correctness under illegal requests. Such requests can be categorized into two types: one with an illegal operand code (e.g., an RDMA READ instead of SEND) or one with an invalid argument. For the first one, we can configure the RDMA connection permission such that they will be rejected by the RNIC. For the latter, we only need to handle the invalid `tt_addr`, as the WRs manipulate TTs and the free page queue based on it (see Figure 6). Specifically, an invalid `tt_addr` can be (1) misaligned, (2) out-of-range and (3) being unmapped in a request that assumes mapped `tt_addr`. For (1), we can configure the RECV WR to only use the upper bits of the `tt_addr` to efficiently avoid misaligned `tt_addr`. For (2), we use RDMA's memory registration to ensure that all the WRs in WR chains can only access the CNode's own TT. For (3), we ensure the unmapped TT entries are zero through proper initialization and clean-up. As a result, such an argument will cause an RDMA protect error due to null access, which will abort execution. Additionally, we insert a conditional check similar to the one described in Figure 2 to rule out unmapped `tt_addr` in page invalidation operations.

Fairness. To ensure fairness, we follow existing works [11, 19] that set a budget for the maximum allocated memory for CNodes. If the budget is exhausted, the MNode will reject allocation requests from the CNode to maintain fairness. The challenge is how to keep track of each CNode's usage. Existing systems can leverage the CPU to actively record it before allocating memory to a CNode. However, in ODRP, we bypass the MNode CPU in memory allocation. To address this, we leverage a lazy detection method to periodically check whether a CNode exceeds its budget. Specifically, we use one CPU core at the MNode to monitor the memory usage of each CNode. The challenge is how to efficiently perform this bookkeeping. A naive solution would be scanning the TTs, which would incur significant CPU overhead. Fortunately, we can leverage the RDMA hardware counters [35] for this calculation. These counters record the completion of WRs in the unmapped store and invalidation WR chains. By comparing these counters, we can efficiently get the number of allocated pages of a CNode.

5 EVALUATION

5.1 Evaluation Setup.

Testbed. Our experiments are conducted on a cluster with one MNode and up to 8 CNodes, all connected with an InfiniBand switch. Each node has a 12-core Intel Xeon E5-2650 CPU (hyper-threading is disabled), 128 GB DDR4 RAM, and a 100 Gbps Mellanox ConnectX-5 RNIC. Each node runs Mellanox OFED 4.9 and Linux 4.15. Each CNode is configured with 12 GB swap space.

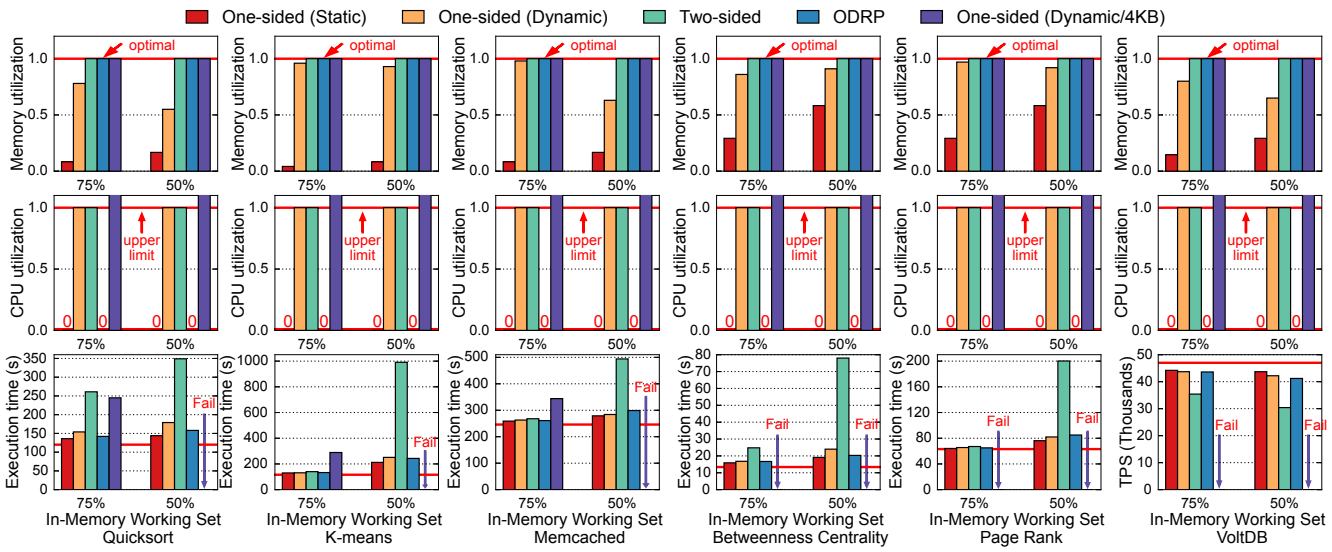


Figure 7: Remote memory utilization (higher is better), remote CPU usage (lower is better), and the execution time (throughput) of ODRP and other four baseline systems under different workloads with 8 CNodes accessing one shared MNode.

Baselines. We compare ODRP with four baselines as described in § 2.2: 1) One-sided (static), pre-registering 12 GB of remote memory for each CNode; 2) One-sided (dynamic), preparing 1 MB MRs in advance while allocating them to the CNode on-demand; 3) Two-sided, using two-sided RDMA operations in the data path to implement 4 KB-granularity memory management; 4) One-sided (dynamic/4KB), same as One-sided (dynamic) but performing 4 KB allocation. The MNode only uses one CPU core to simulate the weak processing power, unless otherwise specified. We implement all the baselines based on Fastswap [11] and compare them to ODRP, aiming to demonstrate that ODRP outperforms baselines regarding application performance, remote memory utilization, and remote CPU usage.

Evaluated applications. We evaluate the applications used in prior memory disaggregation studies [19, 11, 43]. 1) Quicksort with a 4 GB working set; 2) Kmeans with a 2 GB working set; 3) Memcached v1.4.25 with Facebook ETC workload [12] (4 GB working set), a server thread and four client threads; 4) Page rank (PR) and 5) betweenness centrality (BC) from GAPBS [13] on a Twitter dataset [24], 4 threads and a 14 GB working set; 6) VoltDB with TPC-C [8], 8 threads and a 7 GB working set.

5.2 Application Benchmarks

We run the applications on all available hardware in our cluster, which consists of 8 CNodes and one shared MNode. This cluster setup aligns with the typical deployment of memory disaggregation, such as in cloud environments [26].

Memory utilization. The first row of Figure 7 illustrates the average remote memory utilization during application runtime, which is measured as the ratio of actually used remote memory to allocated remote memory. One-sided (static) can only achieve up to 58.3% remote memory utilization because of its coarse-grained static allocation. One-sided (dynamic) can significantly improve remote memory utilization by adopting fine-grained allocation. However, it still cannot achieve ideal utilization because of the fragments within 1 MB memory slabs. It achieves about 95% remote memory utilization for workloads where internal fragmentation is not significant, such as Kmeans, betweenness centrality, and page rank. More severe internal fragmentation can lead to worse remote memory utilization. For example, One-sided (dynamic) can only achieve 55% remote memory utilization in the case of Quicksort with 50% local memory.

In contrast, thanks to the page-granularity memory management, ODRP, Two-sided, and One-sided (dynamic/4KB) can achieve ideal remote memory utilization (100%). This results in up to 1.82× improvement over One-sided (dynamic) and a 1.72× to 12× enhancement compared to One-sided (static).

Remote CPU usage. As shown in the second row of Figure 7, both One-sided (dynamic) and Two-sided reach the upper limit of one MNode CPU core in all workloads. The saturated MNode CPU turns into a performance bottleneck, as discussed in more detail in § 5.3. Some workloads even fail to complete in One-sided (dynamic/4KB). If we use a single-CNode evaluation setup, One-sided (dynamic) results in a maximum of 67% remote CPU usage because the MNode CPU needs to serve memory (de)allocation requests. Two-sided requires higher remote CPU usage because it involves the MNode CPU in every swap in/out request. In Kmeans and graph processing workloads, which involve more frequent swap operations, Two-sided leads to 86% and 99% remote CPU usage, respectively.

For One-sided (dynamic/4KB), we removed the restriction of providing only one MNode CPU core, otherwise the applications fail to complete. One-sided (dynamic/4KB) requires

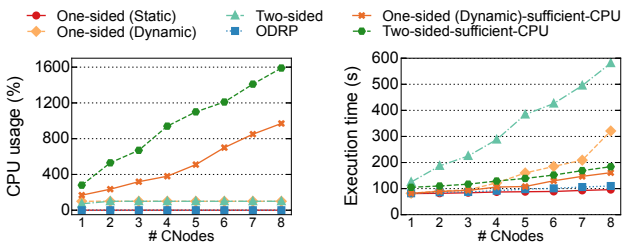


Figure 8: The impact of increasing the number of CNodes (each with 4 Quicksort tasks) on (a) Remote CPU usage and (b) Execution time

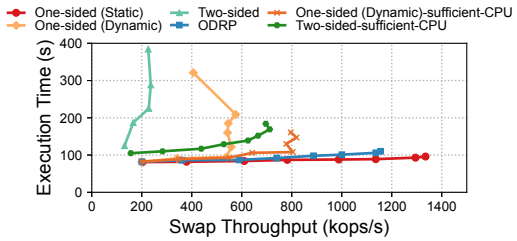


Figure 9: The average swap throughput (i.e., the number of swap in/out requests per second) and the application execution time increase as the number of CNodes (each running 4 Quicksort tasks) increases from 1 to 8.

the most intensive MNode CPU involvement and can use up to 3 MNode CPU cores in the single-CNode setup. In the 8-CNode setup, some applications even fail to complete due to poor performance.

In contrast, One-sided (static) does not require the MNode CPU during runtime due to its static allocation (i.e., all remote memory is provisioned at initialization time). ODRP implements all of its memory management logic through RDMA offloading, thereby also bypassing the MNode CPU.

Performance. The third row of Figure 7 presents the application performance (execution time or throughput). The red line indicates the performance with 100% local memory (optimal). For Quicksort, compared to One-sided (static), ODRP shows a modest performance overhead of 9.7%. In contrast, One-sided (dynamic) experiences a 24.3% performance overhead because the MNode CPU becomes saturated, causing allocation requests to queue up. Two-sided encounters substantial performance degradation, primarily due to significantly slower remote access and swap request queuing. resulting in a 142.4% overhead. One-sided (dynamic/4KB) shows the worst performance due to frequent MR registering. It introduces an 80.1% performance overhead with 75% local memory, and the application fails to complete with 50% local memory.

For Kmeans, reducing the in-memory working set has a more severe impact on its execution time because it triggers the most frequent major page faults that need swap operations in all evaluated workloads, averaging 6,400 times per second (46.7% higher than Quicksort). Compared to One-sided (static), ODRP experiences slightly higher swapping latency due to more frequent swap requests, as detailed in § 5.4. In this swapping-intensive workload, ODRP introduces a 14.2% performance overhead compared to One-sided (static).

For Memcached, both ODRP and One-sided (dynamic) demonstrate negligible performance overhead when local memory is decreased, because the swap operations are less frequent. ODRP incurs a 7.2% performance overhead compared with One-sided (static). Additionally, in terms of the query latency, the average and maximum latency increases introduced by ODRP always remain below 2%.

For BC and PR, ODRP's performance overhead is 6.3% and 11.5% compared to One-sided (static), respectively. It can outperform One-sided (dynamic) by up to 15.4%. For VoltDB, ODRP reduces throughput by 4.1% compared to One-sided (static). Regarding transaction latency, ODRP modestly elevates both the average and maximum latencies by less than 1% and 4.9%, respectively.

5.3 Scalability Analysis

In this section, we analyze the scalability of ODRP by increasing the number of CNodes and compare it to other baselines. We choose Quicksort because its swap frequency is relatively high, which can more effectively demonstrate the sensitivity of ODRP under different scales of swap frequency.

Scalability with the increasing number of CNodes. Figure 8 shows the average execution time and remote CPU usage as we increase the number of CNodes from 1 to 8, each running four Quicksort tasks (50% local memory). Despite achieving ideal remote memory utilization, Two-sided exhibits the poorest performance and scalability. The MNode CPU has been saturated with just one CNode. With 8 CNodes (i.e., 32 Quicksort tasks), it results in a 505% performance overhead compared to One-sided (static). Although One-sided (dynamic) can attain up to 77% remote memory utilization, it also overwhelms the MNode CPU with just one CNode and introduces a 234% performance overhead at 8 CNodes compared to One-sided (static). In contrast, ODRP offloads all memory management operations to the RNIC and is not limited by the wimpy MNode CPU. Consequently, ODRP demonstrates superior performance and scalability compared to Two-sided and One-sided (dynamic).

Throughput-latency. To demonstrate that ODRP can scale, we measure the overall swap throughput with respect to the application execution time. As shown in Figure 9, both One-sided (static) and ODRP reach a saturation point for the swap throughput at 8 CNodes, with ODRP achieving 87.3% of the swap throughput of One-sided (static). Consequently, ODRP incurs performance overheads of 5.7% and 14.6% at 4 and 8 CNodes, respectively. Nevertheless, ODRP significantly improves remote memory utilization, achieving 3 × better utilization than One-sided (static).

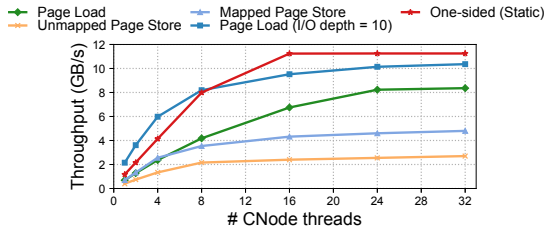


Figure 10: The throughput of different data path operations in ODRP and One-sided (static).

Microbenchmark analysis. To further analyze the scalability of ODRP, we evaluate the throughput of various data path operations in ODRP and One-sided (static). As shown in Figure 10, the throughput of page load operation with high I/O depth reaches 92.1% of the throughput achieved by RDMA READ/WRITE operations, i.e., data path operations in One-sided (static). The unmapped page store operation exhibits the lowest throughput because the throughput of RDMA atomic FAA, which is required by it, is an order of magnitude lower than that of RDMA READ/WRITE [34].

Even though ODRP introduces overhead to swap requests, application performance in ODRP is still on par with that in One-sided (static). This can be attributed to two reasons. First, page load requests account for the majority of all requests (more than 60% of the swap requests are page load) and they are sent by the CNode's swap backend in a batched manner due to page prefetching, where the performance of ODRP and One-sided (static) is similar. Meanwhile, the least efficient unmapped page store operation only accounts for approximately 10% of the requests. Second, the RNIC supports a high throughput of RDMA requests, such that even applications with high swap frequency can barely saturate it (see Figure 9).

Comparisons with baselines with sufficient MNode CPU. We further include two baselines where enough MNode CPU cores are provisioned to One-sided (dynamic) and Two-sided. As shown in Figure 8, Two-sided costs up to 16 MNode CPU cores, but it still cannot achieve the throughput of ODRP and incurs up to a 67.3% performance overhead compared to ODRP. One-sided (dynamic), being less demanding on the MNode CPU, can achieve higher throughput with up to 10 MNode CPU cores. Nevertheless, it remains 46.4% slower compared to ODRP at 8 CNodes due to additional time spent on memory (de)allocation. This is because the (de)allocation operations are costly when executed with the MNode CPU, and they are not rare during execution, occurring on average 3,000 times per second.

5.4 Factor Analysis

Figure 11 shows the latency of various data path operations in ODRP and baseline systems. When there is only one worker issuing requests (without resource competition), the one-sided RDMA READ/WRITE in One-sided (static) exhibits the lowest latency, measuring at 2.9 μs. In ODRP, the most frequent operations, namely load and mapped store, have latencies of 5.5 μs and 4.6 μs, respectively. Note that such an increase in latency can be mitigated by the swap system, e.g., with page prefetching and asynchronous page store. In comparison, the data path operations in Two-sided have significantly higher latency, 3.8× than that of the native one-sided operations. Note that One-sided (dynamic) also utilizes native one-sided operations in the data path, and the latency of `Allocation` is not excessive. This explains why One-sided (dynamic) does not introduce substantial performance overhead when the MNode CPU is not overly overwhelmed.

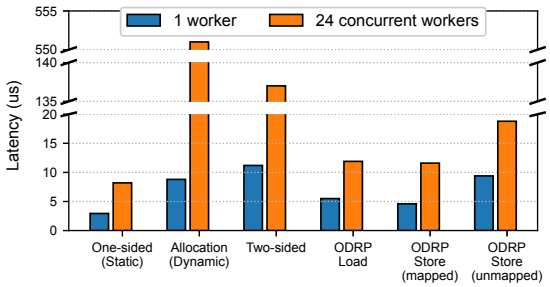


Figure 11: Latency of various data path operations. `Allocation` refers to acquiring a 1 MB memory slab (i.e., MR) in One-sided (dynamic).

When there are 24 concurrent workers issuing requests, there is an order of magnitude increase in the latency of `Allocation` and Two-sided data path operations due to the overloaded MNode CPU. This explains why both One-sided (static) and Two-sided experience a sharp decrease in performance as the number of CNode rises (see Figure 8).

The effect of ODRP optimizations. To quantify ODRP's optimizations for WR chains, we break down the execution time of the unmapped page store WR chain. The naive unmapped page store exhibits an unacceptable latency of 35.9 μs. `StoreDecision` (§ 4.4) splits the page store operation into mapped page store and unmapped page store to avoid an `if` statement in the WR chain, saving one CAS and one WRITE and reducing the latency by 1.9 μs. `CalcTTAddr` (§ 4.3) avoids calculating `tt_addr` in the WR chain, saving one FAA WR and one EndianSwap WR and reducing latency the by 2.4 μs. `RecycleHelper` (§ 4.4) prevents the use of additional FAA and EndianSwap WRs for reactivating WR chains, reducing the latency by 16 μs. Finally, `EndianSwap` (§ 4.2) reduces the number of WRs needed for the endian conversion from 8 READ to 1 READ with 8 SGEs, further reducing the latency by 6.2 μs.

6 Discussion

Extensibility. We implemented ODRP's CNode backend and the WR chain offloading (based on the RedN framework [34]) with about 1,300 and 2,400 lines of C code, respectively. Our prototype implementation only considers a setup where multiple CNodes share one MNode. However,

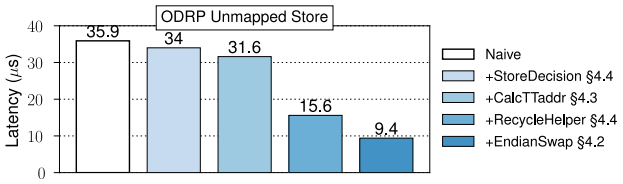


Figure 12: Analysis of ODRP optimizations to the average latency of unmapped store.

ODRP can be easily extended to support a multi-CNode-multi-MNode setup. To achieve this, a CNode only needs to partition its swap space across different MNodes and then locates the respective MNodes accordingly during runtime.

Furthermore, ODRP supports configurable allocation and access granularity, making it compatible for integration with other memory disaggregation systems [36, 43, 20, 33] that focus on optimizing CNode runtime. ODRP can improve their remote memory utilization while preserving the benefits of an optimized CNode runtime.

Memory Overhead of Translation Table. In ODRP, the MNode maintains a TT for each CNode that connects to it. The TT size depends on the size of the swap space provisioned to CNodes. Each TT entry stores an 8 B free page address. For a 32 GB swap space, the TT only occupies 64 MB of memory (0.2% memory overhead). Furthermore, this memory overhead can be alleviated by registering TTs as On-Demand-Paging MRs [5].

Suggestions to Future RNIC Vendors. Currently, the RNIC can only fetch and execute WRs one-by-one from the WQ in host memory if preceding WRs need to update subsequent WRs. This hampers efficiency when offloading complex functionalities. In the future, RNICs could incorporate a prefetching mechanism to speed up the execution of the self-modifying WR chains. Additionally, we encourage RNIC vendors to provide primitives with richer semantics. We have demonstrated that implementing this would not be overly challenging based on the current RNIC hardware. By introducing more powerful WRs, the length of WR chain can be reduced, and the development effort can be saved.

7 Related Work

RDMA-based Remote Swapping. Previous research studies have adopted RDMA remote memory as a swap backend [19, 11, 33, 28, 43, 40]. Infiniswap [19] places frequently accessed area of its swap space in remote memory to boost performance. Fastswap [11] further improves swapping performance by utilizing frontswap [3] interfaces. It makes non-critical operations asynchronous and creates a dedicated thread for asynchronous page reclamation. Leap [28] designs a more efficient page prefetching policy suited for remote memory, which is based on majority-based pattern detection. DiLOS [43] proposes to use a single address space operating system (SASOS) to reduce kernel-user switching overhead and improve performance with user-space semantics. However, none of these studies fully utilize remote memory due to their static and coarse-grained remote memory management. ODRP focuses on providing fine-grained memory management with RDMA offloading.

Fine-grained Remote Memory Management. CoRM [38] reduces memory fragmentation by leveraging Mellanox's ODP feature [5] to achieve remote memory compaction. However, CoRM requires intensive CPU involvement to compact memory, which is not suitable for wimpy MNode CPU cores. Programmable hardware devices have also been adopted to support complex computation offloading, such as SmartNICs and FPGAs. Clio [21] relies on dedicated FPGAs and network to virtualize and manage remote memory at a 4 MB granularity. TDMem [22] also utilizes specialized FPGAs and backbone network to support fine-grained and secure memory disaggregation. However, utilizing such specialized programmable hardware comes with higher costs, additional maintenance burdens, and longer deployment cycles [7]. ODRP aims to provide fine-grained memory management on commodity RNICs.

RDMA Offloading. Hyperloop [23] was the first to demonstrate that offloading complex functionality to RNICs can be achieved by combining native RDMA WRs. It leverages this offloading capability to eliminate CPU's involvement from the critical path of replicated transactions in storage systems. Furthermore, RedN [34] has recently proven that RDMA is actually Turing-complete by carefully chaining self-modifying and data-dependent RDMA WRs. RedN provides a blueprint for offloading arbitrary computation logic onto RNICs, and demonstrates its effectiveness and efficiency with two simple use cases: hash lookup and list traversal. ODRP utilizes RDMA offloading to provide fine-grained and efficient remote memory management, bringing ideal remote memory utilization.

8 Conclusion

ODRP demonstrates the viability of fully offloading a complex system (remote swap device) to RNIC, which achieves nearly zero remote CPU usage, high remote memory utilization, and low performance overhead. We highlight the advantages and limitations of RDMA offloading and, more importantly, we propose the use of client-assisted principle to mitigate many of the limitations.

Acknowledgment

We sincerely thank all the anonymous reviewers, whose reviews, feedbacks, and suggestions have significantly strengthened our work. This research was supported in part by the National Natural Science Foundation of China (No. 62202292), the National Key Research & Development Program of China (No. 2022YFB4500700), the Fundamental Research Funds for the Central Universities, and research grants from Huawei Technologies and Intel. Corresponding author: Jinyu Gu (gujinyu@sjtu.edu.cn).

REFERENCES

[1] Enabling the Modern Data Center – RDMA for the Enterprise. https://www.infinibandta.org/.

[2] Extended Atomics Support. https://docs.nvidia.com/networking/display/rdmacore50/Extended+Atomics.

[3] Frontswap. https://www.kernel.org/doc/html/v5.5/vm/frontswap.html.

[4] memcached - a distributed memory object caching system. https://memcached.org/.

[5] RDMA On-Demand-Paging (ODP). https://docs.nvidia.com/networking/display/OFEDv501000/Optimized+Memory+Access#OptimizedMemoryAccess-On-Demand-Paging(ODP).

[6] Redis. https://redis.io/.

[7] Shop End-to-End High-Speed Ethernet and InfiniBand Solutions. https://store.nvidia.com/en-us/networking/.

[8] TPC-C is an On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[9] VoltDB. Volt Active Data. https://github.com/VoltDB/voltdb.

[10] Advanced transport. https://docs.nvidia.com/networking/display/OFEDv502180/Advanced+Transport, 2023.

[11] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[13] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[14] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 79–92, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Alexandros Daglis, Dmitrii Ustiugov, Stanko Novakovic, Edouard Bugnion, Babak Falsafi, and Boris Grot. Sabres: Atomic object reads for in-memory rack-scale computing. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 6:1–6:13. IEEE Computer Society, 2016.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[19] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[20] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Taekyung Heo, Seunghyo Kang, Sanghyeon Lee, Soojin Hwang, and Jaehyuk Huh. Hardware-assisted trusted memory disaggregation for secure far memory. *CoRR*, abs/2108.11507, 2021.

[23] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the*

2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, page 297–312, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[25] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[26] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892, 2017.

[28] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, July 2020.

[29] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Jacob Nelson-Slivon, Reilly Yankovich, Ahmed Hassan, and Roberto Palmieri. Brief announcement: Rome: Wait-free objects for rdma. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '24, page 371–373, New York, NY, USA, 2024. Association for Computing Machinery.

[31] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

[32] Nathan Pemberton. Exploring the disaggregated memory interface design space. In *Workshop on Resource Disaggregation (WORD)*, 2019.

[33] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, Boston, MA, April 2023. USENIX Association.

[34] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 71–85, Renton, WA, April 2022. USENIX Association.

[35] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4277–4292. USENIX Association, August 2021.

[36] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332. USENIX Association, November 2020.

[37] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[38] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. Corm: Compactable remote memory over rdma. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1811–1824, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, November 2020.

[40] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 161–179, Boston, MA, April 2023. USENIX Association.

[41] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 233–251, 2018.

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

[43] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the 18th European Conference on Computer Systems*, EuroSys '23, Rome, Italy, May 2023.

[44] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[45] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29. USENIX Association, July 2021.

Understanding and Profiling NVMe-over-TCP Using `ntprof`

Yuyuan Kang and Ming Liu

University of Wisconsin-Madison

Abstract

NVMe-over-TCP (NVMe/TCP) is an emerging remote storage protocol, increasingly adopted in enterprises and clouds. It establishes a high-performance reliable data channel between clients and storage targets to deliver block I/Os. Understanding and analyzing the protocol execution details and how well storage workloads run atop are pivotal for system developers and infrastructure engineers. However, our community lacks such a profiling utility, whereas existing solutions are ad-hoc, tedious, and heuristic-driven. Realizing it is challenging due to the unpredictable I/O workload profile, intricate system layer interaction, and deep execution pipeline.

This paper presents `ntprof`, a systematic, informative, and lightweight NVMe/TCP profiler. Our key idea is to view the NVMe/TCP storage substrate as a lossless switched network and apply network monitoring techniques. We model each on-path system module as a software switch, equip it with a programmable profiling agent on the data plane, and develop a proactive query interface for statistics collection and analysis. `ntprof`, comprising a kernel module and a user-space utility, allows developers to define various profiling tasks, incurs marginal overhead when co-locating with applications, and generates performance reports based on prescribed specifications. We build `ntprof` atop Linux kernel 5.15.143 and apply it in six cases, i.e., end-to-end latency breakdown, interference analysis, SW/HW bottleneck localization, and application performance diagnostic. `ntprof` is available at https://github.com/netlab-wisconsin/ntprof.

1 Introduction

Storage disaggregation [31, 76, 77, 92, 98] has gained significant traction recently due to independent resource scaling, high utilization, and cost efficiency. It has been increasingly deployed in data centers, enterprise on-premise clusters, and edge clouds. With the rising networking speed and emerging high performance of NVMe drives, a disaggregated SSD can deliver millions of IOPS at tens of microseconds.

The key technology enabler is an NVMe-over-Fabric (NVMe-oF) protocol [43] that delivers high-performance storage accesses between clients (*initiators*) and remote NVMe subsystems (*targets*). An initiator submits block I/O requests–encapsulated into specialized fabric packets–and delivers them to the remote side through an established protocol channel. A target receives packets from the network transport, converts them to NVMe commands, reads/writes from the SSD, and sends the response back. NVMe/TCP is one such networking fabric, which is increasingly adopted and has enabled many real-world deployments [2, 3, 12, 25, 27, 28].

However, our community lacks a system profiling tool to understand and analyze the execution characteristics of the NVMe/TCP protocol. This makes answering many design, operation, testing, and deployment questions impractical. For example, how well do the storage applications run over NVMe/TCP? How should we design the I/O merging and pacing policies of the block layer? What are the system bottlenecks along the protocol pipeline? How should we reconfigure the NVMe/TCP configurations if the initiator/target processors, networks, or NVMe drives are congested?

Building such a utility is non-trivial due to the workload non-determinism, intricate system layer interaction, and deep execution pipeline. First, storage workloads embody diverse I/O profiles, including block size distribution, read-/write mix ratio, sequential/random access pattern, and I/O concurrency. This issue is further exacerbated by (a) emerging NVMe drives with high queue depth; and (b) dense storage appliances, equipped with dozens of SSDs, cumulatively offering large data volumes (10+ terabytes) and massive bandwidth (100+ GB/s). Second, the NVMe/TCP layer closely interacts with other system modules, including the block layer, TCP/IP networking stack, and NVMe subsystem. Each presents certain execution parallelism (e.g., multi-queue, multi-connection, and multi-data path) with flexible load-balancing policies (to maximize throughput and mitigate superfluous I/O stalls). Third, an I/O has a deep pipeline with processing stages spreading two physical hosts across the CPU core, networking, and NVMe drive (Figure 2).

Developers thus resort to a heuristic top-down analysis strategy to diagnose performance anomalies, dissect

overheads, and explore optimization opportunities in an NVMe/TCP-based disaggregated storage. They combine a sequence of development utilities (from the application and system layer [11, 16, 17, 23, 37, 38, 44, 45, 94] to the infrastructure level [18, 39, 51, 66, 67, 85, 86]), embarrassingly combine them based on empirical observations, and manually synthesize isolated profiling reports from different tools to infer the root causes. The entire synthesis process is tedious and inefficient, driven by a vertical view of experimental results, requiring an esoteric understanding of the system and lots of back-and-force test/verification. More importantly, it is generally challenging to reproduce online interleaved I/O scenarios, which are common in a multi-tenant environment, rendering existing approaches futile for root cause exploration.

In this paper, we design and implement `ntprof` to enable systematic tracing and analyzing of the NVMe/TCP. *Our key insight is viewing the NVMe/TCP subsystem as a lossless switched network–an initiator sends I/O requests, a target returns the responses, and I/O blocks encapsulated as TCP packets are forwarded by different system entities along the I/O path–and apply network monitoring techniques.* Essentially, we make each on-path system component a multi-queue software switch, equip it with a programmable profiling agent on the data plane, and develop a proactive query interface for data collection and analysis. As such, our design resembles prior active network-based profiling systems (like TPP [57, 58]).

`ntprof` consists of four system components. First, it provides a profiling task specification template that allows developers to prescribe targeted I/O workloads, deployment environments, profiler configurations, and reporting statistics, enriching profiling functionalities. Second, we develop a lightweight programmable profiling agent based on the Linux tracepoint mechanism, co-locate with each modeled software switch for statistics monitoring, and take translated specifications as predicates. Third, we introduce an in-band TPP-like profiling query protocol, which injects a special NVMe/TCP command capsule to query and collect statistics periodically. Last, `ntprof` produces an informative performance report (including end-to-end application I/O performance, latency breakdown, throughput bottleneck, protocol running details, etc.) either online or offline through a sequence of steps (such as serialization, calibration, and MapReduce-like processing).

`ntprof`, built atop Linux kernel 5.15.143, comprises a kernel model and a user-space utility at the NVMe/TCP initiator and target sides. We evaluate it in a typical NVMe/TCP setup in CloudLab [40] and demonstrate its effectiveness via six case studies. Specifically, `ntprof` can report the I/O latency breakdown of the deep NVMe/TCP pipeline (§4.2) and dissect the I/O interference issue under mixed workload profiles (§4.5). Next, `ntprof` enables developers to locate the system bottlenecks at both software processing pipelines (§4.3) and the underlying I/O substrate (§4.4). Further, `ntprof` can help with application performance diagnostic when deployed in an NVMe/TCP setting (§4.6 and §4.7). `ntprof` is open-source

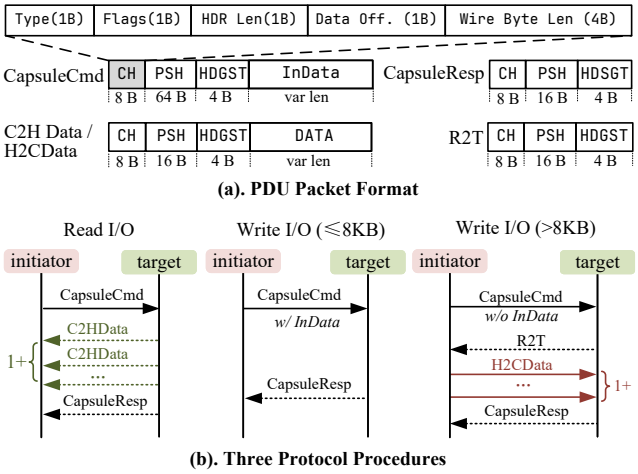


Figure 1: NVMe/TCP protocol details. (a) presents the packet format of five types of PDUs. (b) depicts the protocol procedure of an I/O read and write. InData=In-Capsule Data.

and we will keep working with the community to improve it.

2 Background and Motivation

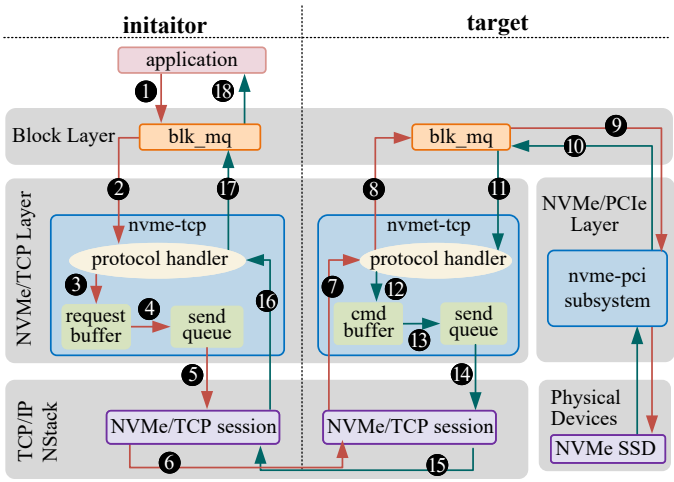
This section provides some background on the NVMe-over-TCP protocol and its Linux implementation. We then discuss the challenges and limitations of existing profiling solutions.

2.1 NVMe-over-TCP Protocol

NVMe-over-TCP (NVMe/TCP) [43] is a high-performance data transfer protocol that defines block-level storage accesses between clients (*initiators*) and remote NVMe subsystems (*targets*) via TCP/IP. An initiator submits block I/O requests–encapsulated into specialized NVMe/TCP packets–and delivers them to the remote side through an established protocol channel. A target receives packets from the network transport, converts them to NVMe commands, reads/writes from the SSD, and sends the response back. NVMe/TCP makes the remote NVMe subsystem behave similarly to a local one.

NVMe/TCP, inherited from the NVMe base protocol [41, 42], constructs reliable bidirectional communication channels (also called NVMe/TCP session) between two sides, where each comprises an initiator-side submission-completion queue pair, a TCP connection, and a target-side NVMe queue pair. The number of channels can be configured to the number of cores of an initiator's processor, which also benefits the multi-queue architecture of an NVMe drive. There is a specialized channel to deliver I/O admission commands.

Protocol Data Unit (PDU) is the basic communication granularity in NVMe/TCP that carries I/O command, data, and control status. As shown in Figure 1-a, it begins with an 8-byte Common Header (CH), followed by a 16 or 64-byte PDU Specific Header (PSH), a 4-byte header digest (HDGST) for cache alignment and integrity check, and variable length in-capsule data (optional). CH defines the PDU type, special flags (e.g., whether digest is present in the header and data), header length, data offset, and the total PDU transmission



Step	Description
❶	Submit a read/write I/O via system calls
❷	Forward the request to NVMe/TCP protocol handler
❸	Generate a CapsuleCmd PDU and insert it into the request buffer
❹	Pop the request from the request buffer and enqueue it to sending queue
❺	Copy the CapsuleCmd or H2CData PDU to the TCP socket buffer
❻	Transmit over the NVMe/TCP session
❼	Read PDUs from the TCP socket buffer
❽	Proccess the PDU, extract an NVMe command, and submit it to the blk_mq
❾	Submit the NVMe command to local NVMe subsystem
❿	Receive the I/O completion from the NVMe subsystem
⓫	Send the block layer response to the protocol handler
⓬	Produce the R2H/C2HData PDU and then insert it into the command buffer
⓭	Pop the response from the command buffer and enqueue it to sending queue
⓮	Dequeue PDUs from the sending queue and copy it to the TCP socket buffer
⓯	Reply over the NVMe/TCP session
⓰	Read a C2HData/R2T/CapsuleResp PDU from the TCP socket buffer
⓱	Perform PDU protocol processing and then produce a block I/O response
⓲	Return an I/O response to the application

Figure 2: The left side depicts the NVMe/TCP system architecture and I/O data path. The right side lists detailed processing stages.

byte. There are five types of PDUs defined in NVMe/TCP for I/O: (a) *CapsuleCmd PDU*, issued from the initiator and transmitting a command capsule that encodes a read/write request in the PSH; (b) *CapsuleResp PDU*, replied from the target and carrying a completion response capsule; (c) *H2CData PDU*(*C2HData PDU*), used for transfer I/O data from an initiator (target) to a target (initiator); (d) *R2T PDU*, a control signal from the target to notify when it is able to accept data from the initiator. The PSH header is not consistent across different PDUs. It encodes the NVMe-oF command capsule submission/completion Queue Entry in CapsuleCmd PDU/-CapsuleResp PDU, carries data in H2CData PDU/C2HData PDU, and stores the R2T PDU's metadata.

NVMe/TCP has three protocol procedures (Figure 1-b). An I/O read sends a CapsuleCmd PDU and receives one or more C2HData PDUs, ending with a CapsuleResp PDU for completion notification. A small write (less than or equal to 8KB) uses CapsuleCmd PDU with the command and data inlined, resulting in a CapsuleResp PDU response from the target, which entails only one round-trip. Large writes require one more admission control phase. First, the initiator submits the write I/O request without data. Upon receiving an R2T PDU, it sends multiple H2CData data transfer PDUs and waits for a CapsuleResp PDU as an acknowledgment. Note that all PDUs are encapsulated as TCP packets and traverse through the corresponding TCP connection.

2.2 NVMe-over-TCP in Linux

NVMe/TCP is first officially released in Linux Kernel 5.0 [1] by Lightbits Labs [26] with host and target drivers, conforming to NVMe over Fabrics 1.1 specifications [43]. It is divided into two kernel modules. One is `nvme-tcp`, staying at the initiator, which (a) instantiates and manages the NVMe/TCP session; and (b) accepts block I/O requests, translates them to NVMe/TCP PDU capsules, and delivers these TCP packets to the remote storage. The other is target-side `nvmet-tcp`, which (a) manages NVMe subsystems of the storage appliance; and (b) executes the NVMe/TCP protocol to facilitate I/O handling. This section describes their system architecture and I/O path based on Linux Kernel 5.15.143 (Figure 2).

`nvme-tcp` (Initiator). Here's its workflow on the submission path of a session. First, applications issue I/O requests via system calls and insert them into the `blk-mq` of the block layer (❶). Next, `nvme-tcp` dequeues a request (❷), traverses the protocol handling, constructs an NVMe/TCP PDU capsule (`struct nvme_tcp_request`), and pushes it into a per-session request buffer (❸). It is a LIFO (Last-In-First-Out) lockless linked list (`struct llist_head req_list`) that accepts block I/Os from all CPU cores. After that, the `nvme-tcp` kernel thread pops a ready-to-transmit request and enqueues it into a per-session LIFO sending queue (`struct list_head send_list`) (❹). Last, these ready capsules are dequeued (❺) and delivered to the corresponding NVMe/TCP session (❻) for transmission via the TCP/IP networking stack. A session is represented as a transport queue (part of `struct nvme_tcp_queue`), associated with a dedicated TCP socket handler and assigned to a specific CPU core. This is realized via either (i) the current kernel thread when the queue occupancy is low (e.g., only one outstanding I/O) or (ii) a deferred tasklet (`nvme_tcp_io_work`) that interleaves the sending and receiving PDU processing to optimize throughput.

The completion path is completely handled by the `nvme_tcp_io_work` routine after the networking stack processing. It reads data from the TCP socket buffer and constructs an NVMe/TCP capsule (⓰). A corresponding PDU handler is triggered based on the PDU type (i.e., C2HData, CapsuleResp, or R2T), which produces a block layer response and returns it to the block layer (⓱) and applications (⓲), or sends an H2CData PDU for data transmission (❸).

`nvmet-tcp` (Target). Storage traffic arrives at the NVMe/TCP session socket, stored in the `nvmet-tcp` transport queue (`struct nvmet_tcp_queue`). Akin to the `nvme-tcp` module, there is a per-session LIFO-based response command buffer (`struct llist_head resp_list`) and a per-session

LIFO-based response sending queue (`struct list_head resp_send_list`). The target workflow is as follows. First, the `nvmet-tcp` kernel thread (`nvmet_tcp_io_work`) reads a batch (up to 64) of PDUs from the TCP socket buffer and activates the PDU handler (❼). As shown in Figure 1-b, a large write without in-capsule data triggers the R2T PDU processing, produces a corresponding response (`struct nvmet_tcp_cmd`), and inserts it into the command buffer (⓬). Otherwise, it processes data payload, extracts the NVMe command (`struct nvme_command`), and delivers it to `blk_mq` (❽). Second, the block layer then interacts with its local NVMe system, submits an IO read/write (❾), accesses the SSD drive, and waits for an I/O completion (❿). Next, similar to Steps 2–4, block I/Os traverse target-side PDU handling (⓫), where the fabricated response capsules are inserted into the command buffer (⓬) and delivered to the send queue (⓭). Finally, the NVMe/TCP session takes response PDUs from the send queue (⓮), performs TCP/IP protocol processing, and sends data back to the initiator (⓯).

2.3 Challenges and Existing Solutions

Challenges. Profiling storage applications running over Linux NVMe/TCP is non-trivial. First, applications exhibit diverse I/O profiles regarding block size, access pattern, and I/O concurrency. Second, the NVMe/TCP layer closely interacts with other system modules on both control and data planes. As shown in Figure 2, it takes I/Os from the block layer, transmits requests/responses via the TCP/IP networking stack, and submits (accepts) NVMe commands to (from) the SSD through the NVMe subsystem. All layers have certain execution parallelism with flexible load-balancing support. Third, each request traverses a long data path between two physical hosts (Figure 2). An I/O can be queued at several locations in a NVMe/TCP setting. There is the block layer [33], where (a) a software staging queue buffers I/Os for flexible scheduling; and (b) a hardware dispatch queue controls the submission rate. Within the TCP/IP networking stack, we have (a) packet pending queue (`sk_write_queue`) to buffer in-network data, (b) traffic control (i.e., `qdisc`), e.g., the TCP Small Queues mechanism (TSQ) limiting the number of transmitted packets to reduce RTT and avoid bufferbloat [80]; (c) driver RX/TX queue, i.e., fixed-sized ring buffer. Further, NVMe employs the multi-queue interface between a host driver and an NVMe controller to carry submitted and completed I/O commands.

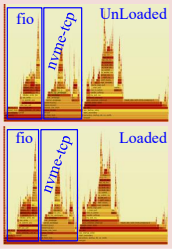
Existing Solutions. There are no structured and systematic profiling utilities in the Linux NVMe/TCP domain for performance diagnostic, overhead analysis, and optimization exploration. Developers today leverage a sequence of development utilities, embarrassingly combine them based on empirical observations, and manually synthesize isolated profiling reports from different tools to infer the root causes.

People usually take a top-down strategy and gradually apply tools from the application layer to the system stack, down to the infrastructure level. For example, one would first

Application Layer *fio*

Metrics	Unloaded	Loaded
Latency (usec)	867.1	1999.6
Throughput (MB/s)	1441.0	625.0

Hot System Layer *Perf*



Top-6 Function Calls Breakdown Sorted by Overhead(%)

Rank	UnLoaded Function Name	%	Loaded Function Name	%
1	get_io_u	24.6	get_io_u	22.7
2	internal_get_*	7.6	__blk_queue*	6.5
3	__blk_queue*	5.9	internal_get_*	6.2
4	try_grab_*	4.1	try_grab_*	3.7
5	__bio_try_*	2.2	bio_iov_iter_*	2.6
6	bio_iov_iter_*	1.9	__bio_try_*	2.5

Infrastructure Layer *iperf3, qperf*

Metrics	Unloaded	Loaded
Latency (usec)	11.6	11.4
Throughput (Gb/s)	88.5	93.8

Figure 3: Analyzing NVMe/TCP via existing solutions. We set up the unloaded and loaded cases on the NVMe/TCP target, where the latter has background I/O streams contending the SSD. In both scenarios, we run the `fio` workload (128KB write, iodepth=10) and show how existing tools from different system layers (`Perf`, `iperf3`, and `qperf`) fail to diagnose the issue.

use application-provided microbenchmarks (like RocksDB's `db_bench` [16] and Lustre's `lnet_selftest` [11]) and tracing utilities to characterize the basic I/O performance and collect running logs. Next, after figuring out anomaly/unexpected execution, she would use system tools (such as `gprof` [45], `JProfiler` [94], `cProfile` [44], `Valgrind` [23], `blktrace` [17], `Oprofile` [37] and `Perf` [38]) to examine CPU usage, memory footprint, thread hotspot, or even some hardware architectural events (like L2/L3 cache misses and retired instructions per cycle) and debug the issue. Last, she might also need to employ low-level infrastructure utilities, e.g., `iperf3` [39], `qperf` [85], `iostat` [18], and `iometer` [66], to validate if the underlying networking fabric and I/O substrate satisfy the deployment assumption. This process is tedious and requires lots of expertise and trials. Worse yet, it is hard to reproduce some interleaved I/O scenarios, especially in a multi-tenant disaggregated storage environment.

We configure an unloaded and loaded NVMe/TCP target and run the same `fio` workload atop (Figure 3). When the SSD is congested (loaded scenario), its I/O throughput drops from 1441.0 MB/s to 625.0 MB/s with average latency increasing from 867.1 µs to 1999.6 µs. However, when using `Perf` to locate the issue, it reports nearly no difference in both cases. For example, the user-space function `get_io_u` consumes the most CPU cycles, i.e., 24.6% and 22.7%, respectively. Kernel functions `__blk_queue_split` and `internal_get_user_pages_fast` rank second and third. When using networking utilities (like `iperf3` and `qperf`), we find that the network pipe between the initiator and target provides enough bandwidth (i.e., 88.5 GB/s and 93.8 GB/s), ruling out the network bottleneck. Thus, using existing tools to pinpoint the root cause is not straightforward.

Therefore, an adequate NVMe/TCP profiling utility should (a) generate different kinds of performance reports for ar-

bitrary I/O workloads (like average/tail latency, throughput, queueing delay); (b) capture I/O data path execution horizontally between distributed hosts and vertically through the system stack; (c) provide diverse and efficient analysis capabilities that serve either online/offline scenarios, identify ephemeral/persistent I/O behaviors, and incur marginal overheads for co-located applications. We target NVMe/TCP-based storage disaggregation due to its wide adoption [2, 3, 12, 25, 27, 28].

3 `ntprof`: An NVMe-over-TCP Profiler

This section describes the design and implementation of `ntprof`. Our system design goals are as follows:

- **Informative.** `ntprof` should provide adequate running statistics about NVMe/TCP processing at the prescribed I/O granularity and profiling scope for protocol understanding, performance diagnostic, and bottleneck localization;
- **Profiling rich.** `ntprof` should provide diverse end-to-end profiling capabilities, accommodating various storage workload profiles (e.g., IO sizes, read/write ratio, access pattern, and concurrency), dissecting deep and pipelined protocol handling, analyzing inter-layer interaction, and reporting characteristics based on the task specification;
- **Lightweight.** `ntprof` should allow online and offline profiling based on the developer's need. Since `ntprof` operates at the data plane and interacts with crucial system components (§ 2.2), it is pivotal to ensure `ntprof` incurs minimal system overheads in terms of memory footprint and CPU cycles with a marginal impact on running I/O applications.

3.1 Key Idea: NVMe/TCP as a Network

`ntprof` treats the NVMe/TCP subsystem as a lossless data network–an initiator sends I/O requests, a target returns the responses, and PDUs encapsulated as TCP packets are forwarded by different system entities along the I/O path. Inspired by the efficacy of network telemetry systems [32, 48, 54, 103], we model the NVMe/TCP setup as a switched networking system and apply network monitoring techniques. Essentially, we make each on-path system component a multi-queue software switch, equip it with a programmable profiling agent on the data plane, and develop a proactive query interface for data collection and analysis. As such, our design behaves similarly to active network-based profiling [57, 58].

In active networks [95, 96], switches, routers, and gateways employ a programmable architecture, and can execute mini-programs (also called "active capsule") carried by network packets. This enables several in-network capabilities, such as application-specific multicast, information fusion, monitoring, and middlebox functionalities (like address translation, load balancing, monitoring, deep packet inspection, and intrusion detection). TPP (tiny packet program) [57, 58] is an active network-based monitoring system. An Ethernet packet, embedding a TPP code segment (based on six load/store instructions), can execute the TPP program on the switch to collect runtime statistics when forwarded. The endhost then develops the data-plane network telemetry using the TPP execution results. `ntprof` employs the same approach to the NVMe/TCP context. We use a special TPP-like I/O request to collect statistics from each on-path system module and reconstruct the profiling report at the end.

Figure 4 presents the overall system architecture of `ntprof`. Programmers define the profiling task and specify its configuration (§3.3). We then translate these specifications into different profiling rules and install them into the corresponding profiling agents (§3.4), co-located with modeled switching components (§3.2). After launching the application, `ntprof` (a) bookkeeps prescribed execution statistics of traversed I/Os at all active entities on the data plane (§3.4); (b) issues a special query command periodically to collect temporary and isolated profiling summary from individual layers (§3.5); (c) reconstructs the entire I/O running profile and performs stipulated analyses (§3.6). `ntprof` works in both offline and online modes. The former targets in-depth analysis and diagnosis based on full profiling data sets, whereas the latter aims for real-time running insights through window-based monitoring.

3.2 Modeling the I/O Path

We model the NVMe/TCP-based storage substrate as a multi-stage Clos network [36, 88], where senders (initiators) and receivers (targets) communicate via a sequence of buffered software switches (Figure 4). Thus, the I/O path of an NVMe/TCP submission request (completion response) consists of a chain of switching queues. We then categorize the queueing model of different software layers using Kendall's notation [60]. In the following expression `A/S/c/K/N/D`, `A` is the arrival process, `S` is the service time distribution, `c` is the number of workers (consumers), `K` is the queueing capacity, `N` is the number of customers (producers), and `D` is the queueing discipline. We summarize three models based on the execution characteristics of different on-path software components.

- The *Central-FCFS* model (formally $G/G/1/Q/X/FCFS$) takes requests from one or several producers ($X \geq 1$) and processes them in the first-come-first-serve (FCFS) order. The per-core software staging queue of the `blk-mq` layer, the initiator-side per-session NVMe/TCP transport queue, and `nvme-pcie` submission queue belong to this;
- The *Split-FCFS* model (formally $G/G/X/Q/1/FCFS$) partitions I/Os to multiple FCFS workers ($X \geq 1$) based on the execution mapping, such as a target-side session transport queue and a `nvme-pcie` completion queue;
- The *Central-PS* model (formally $G/G/1/Q/X/PS$) is a little complex, used to describe the networking stack handling [34]. Even though NVMe/TCP packets traverse each connection in order, the connection-core mapping happens differently, depending on the system setup. For example, one core could concurrently take several sessions ($X \geq 1$)

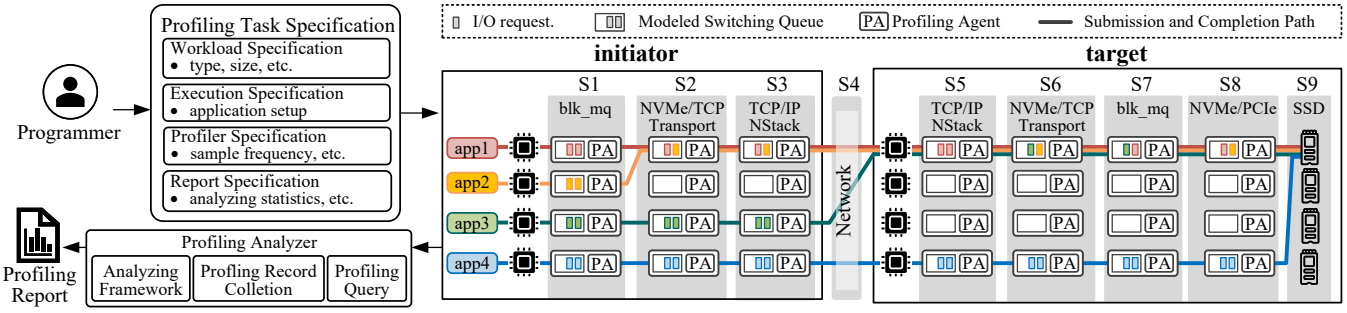


Figure 4: System overview of `ntprof`. PA refers to the Programmable Profiling Agent, attached to each switching queue. "S" refers to a processing stage. The right side of the figure illustrates four example applications. The I/O paths of app1 and app2 overlap at Stage 2 because both are mapped to the same NVMe/TCP queue. App3 shares the same I/O path as app1 at Stage 4, as their TCP connections are handled by the same core on the target side. The I/O path of app4 overlaps with app1 at Stage 8 as both mount the same SSD drive.

and load balances CPU cycles following some processor sharing (PS) discipline. One session usually can not be touched by more than one core at any given time.

These models allow us to abstract the software switch accurately, effectively integrate profiling agents into each layer, and enable more accurate queueing stall estimation.

3.3 Profiling Task Specification

`ntprof` requires a specification template (Figure 4) to define the profiling task, including I/O workloads, running environment setup, profiler configuration, and profiling statistics.

- **Workload Specification** prescribes what I/Os to profile. It tells `ntprof` (a) which NVMe/TCP sessions (i.e., name and mounted address) and (b) what types of I/Os (e.g., I/O read-/write type and I/O sizes) to trace and analyze. We support exact match (size=4KB) and range match (size≥32KB);
- **Execution Specification** defines how the application is deployed and run. This includes (a) application setup, e.g., whether application threads are pinned; (b) NVMe/TCP configuration, e.g., the number and core mapping of transport queues, connections, and NVMe queues; (c) hardware limits of network and storage substrates, e.g., maximum bandwidth of network links and NVMe drives;
- **Profiler Specification** determines the workflow of `ntprof`. First, we support both online and offline modes, where the former reports runtime statistics (like PAPI [19]), and the latter provides an execution summary based on entire tracing. Second, we allow users to specify the sampling frequency (in terms of time epoch or the number of I/Os) and profiling time window, e.g., the start and end time pair being the conventional `YYYY-MM-DD hh:mm:ss` format. Third, `ntprof` requires setting the maximum in-memory buffer size to store the profiling record. When exceeded, it either overwrites the old ones or dumps them on the local drive based on user preference. Last, in the online mode, we provide several programmable primitives to enable more advanced statistical reporting. For example, `MIN|MAX|AVG|DIST` computes the min, max, average, and distinct value of a given profiling range or time window;

Tracepoint	Triggered Condition
`_queue_rq`	When a block I/O enters the NVMe/TCP layer
`_queue_request`	After a block I/O is converted to a CapsuleCmd PDU
`_try_send_cmd_pdu`	When a CapsuleCmd PDU is copied to the TCP skbuff
`_try_send_data_pdu`	When a H2C Data PDU is copied to the TCP skbuff
`_try_send_data`	When a chunk of data is copied to the TCP skbuff
`_done_send_req`	When the entire PDU is copied to the TCP skbuff
`_handle_c2h_data`	After parsing the header of a C2HData PDU
`_recv_data`	After receiving a chunk of C2HData PDUs
`_process_nvme_cqe`	After receiving a CapsuleResp PDU
`_handle_r2t`	When parsing the header of an R2T PDU

Table 1: Tracepoints added in the `nvme-tcp` (initiator) module.

- **Report Specification** lists the format of profiling results. There are two aspects. One is analyzing statistics, including (a) I/O latency distribution, (b) end-to-end throughput, (c) latency breakdown of different layers across the I/O path, and (d) queueing occupancy. The other one is data presentation. `ntprof` allows users to stipulate grouper and aggregator functions to organize execution statistics, such as grouped by the I/O type, size, and session.

3.4 Programmable Profiling Agent

We equip each modeled software switch with a profiling agent that includes a collector capable of gathering I/O running characteristics, and predicates to filter out irrelevant ones.

Statistics collector. We build the collector based on the Linux tracepoint mechanism [10]. A tracepoint is a static instrumentation point in the kernel and exposes a hook to call a function provided by the developers. It can take parameters (Appendix A.1) and enrich the execution logic. This makes collecting detailed low-level statistics at specific points of interest possible. We add tracepoints when a request (a) enters/leaves a software module; (b) traverses essential processing stages (Figure 2). Tables 1/2 list the tracepoints that `ntprof` introduces into the `nvme-tcp` and `nvmet-tcp` modules.

Upon triggering a tracepoint, `ntprof` creates, updates, and seals a profiling record based on the registered function. Each I/O is associated with one record, consisting of request metadata (such as type, size, tag, and command/session ID) and time-series triggered events throughout the lifetime of an I/O (Figure 5). An event log is a 12-byte tuple, including the event name and activating time, which is appended to the profiling

Tracepoint	Triggered Condition
_try_recv_pdu	When nvmet-tcp tries to read data from a skbuff
_done_recv_pdu	After reading the header of a received PDU
_exec_read_req	After receiving a complete read request
_exec_write_req	After receiving a complete write request
_queue_response	When inserting an element into the response queue
_setup_c2h_data_pdu	When starting to make a C2HData PDU
_setup_r2t_pdu	When starting to make an R2T PDU
_setup_response_pdu	When starting to make a CapsuleResp PDU
_try_send_data_pdu	After copying the header of the C2HData PDU
_try_send_r2t	After copying an R2T PDU to the TCP skbuff
_try_send_response	After copying a CapsuleResp PDU to the TCP skbuff
_try_send_data	After copying a chunk of data to the TCP skbuff
_handle_h2c_data_pdu	After reading the header of an H2CData PDU
_try_recv_data	After reading a chunk of data from the TCP skbuff

Table 2: Tracepoints added in the `nvmet-tcp` (target) module.

record. Generally, a record is initialized when the tracepoint `nvme_tcp_queue_rq` happens and completed upon triggering `nvme_tcp_process_ncme_cqe`. The size of a record is I/O-dependent. For example, a read I/O, a write I/O without inline data, and a write I/O with inline data consume at least 207, 183, and 303 bytes, respectively. Appendix A.2 presents the event list of these three cases. In `ntprof`, a profiling record is pinned to a dedicated session that handles the I/O. The collection of records from all sessions forms the entire profiling statistics. Relating to Figure 4, `ntprof` collects the ingress and egress times of the profiled I/O, except for Stage 3-5 and Stage 9, as in the network stack, PDUs are flattened and packed into binary data inside TCP packets. Additionally, `ntprof` also collects the length of the PDU sending queues at Stage 2 and Stage 6. `ntprof` follows the kernel log design and uses an in-memory ring buffer to store records, whose size is configured during initialization.

Predicate. `ntprof` translates profiling specifications (§3.3) into predicates to filter out uninterested I/Os, improving the profiling efficiency. A predicate is a logical expression composed of one or more conditions, often connected by logical operators such as "and". For example, if the programmer specifies capturing 4K read I/Os, the predicate would be: $type = read \wedge size = 4KB$. When a tracepoint is active, the I/O is checked by one or several predicates to determine whether a record update is needed. `ntprof` merges predicates from one I/O stream and allows online editing.

3.5 Profiling Query Protocol

`ntprof` views the NVMe/TCP-based storage substrate as a networking system and develops an in-bind query protocol to obtain statistics at runtime. Akin to TPP [58], we design a special NVMe/TCP command capsule (called `ProbCmd`), carrying a small program segment that states what and where to query via basic load/store/reset primitives. We then introduce a response capsule (`ProbResp`) to return the results. `ProbCmd` and `ProbResp` follow the existing PDU format (Figure 1), consisting of an 8-byte command header and other corresponding headers. The data fields are restructured as a sequence of "query instructions" in the format of `[Query_Opcode] [Software_Switch_ID] [Statistics_Type] [Parameters]`. These two commands

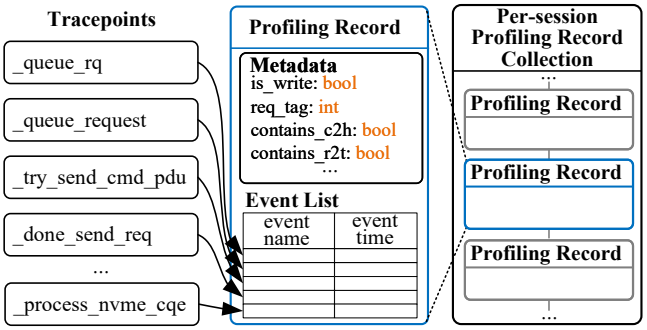


Figure 5: Tracepoints, profiling records, and per-session profiling record collection in the `nvme-tcp` layer.

specify the NVMe/TCP session, not the actual logical block address (LBA), and traverse the same path as read/write I/Os. Our protocol works as follows across different entities:

- **Requester.** The `ntprof` host utility, co-located with storage applications, fabricates the query request and submits it to the block layer. We support both frequency-based (per X I/Os) and time-based operating modes (per Y seconds). Upon receiving the responses, it then extracts the query results and delivers them to the profiling analyzer;
- **Executor.** When receiving the probe command, the software switch extracts the embedded program, teases out belonging instructions (based on the switch ID), and executes them sequentially. It blocks the profiling agent temporarily, collects all profiling records associated with the NVMe/TCP session, and manipulates their profiling records based on the statistic type field. For a load primitive, we copy the statistics, build the response capsule, and then directly push it to the completion path. A store/reset opcode causes updating/clearing up the profiling record and sending an acknowledgment back. Since there might be multiple I/O paths for one session, when forwarding a probe request, our switch would duplicate the special command and submit them to all traversed egressed paths, resembling the switch port mirroring technique [14];
- **Responder.** Unlike TPP which relies on the destination node to construct a reply packet and piggyback queried results, `ntprof` allows each software switch to reply directly due to the large memory usage of profiling records. This indicates that one `ProbCmd` capsule incurs `ProbResp` capsules back. Further, when the probing request reaches the NVMe/PCIe layer at the target side (Figure 2), we send back a probing completion response to notify the controller.

3.6 Profiling Analyzer

During the querying phase, the host sends profiling queries and buffers the received results. `ntprof` then processes and analyzes the data. The analysis involves four steps:

First, `ntprof` provides different ways to organize profiling records, such as by NVMe/TCP session, I/O path, issuing core, target NVMe drive, or request category. We first categorize

these records based on the prescribed criteria (defined in the report specification), wherein each is further serialized based on the I/O submission timestamp (tagged when entering the block layer). Second, `ntprof` performs statistics calibration. End-to-end profiling results are obtained directly at the initiator host. The latency breakdown gathers statistics from all on-path agents which contain event processing time and queueing delay. We timestamp the enqueue and dequeue events whenever possible and use our abstracted queueing model (§3.2) with the queueing occupancy and I/O concurrency distribution for validation. Third, `ntprof` encodes grouper/aggregator functions into a multi-threaded map-reduce framework. We spawn several mappers based on the profiling data volume and group records based on the mapper keys (e.g., 4KB I/O). Intermediate results are shuffled to a couple of reducers that perform the aggregation; Fourth, we output profiling results in the JSON format, which could also be visualized.

3.7 Implementation

`ntprof` requires kernel patches, adds a new kernel module, and provides a user utility, ∼10K LOCs in total.

Kernel Modifications. Tracepoints are the key approach enabling our programmable profiling agent. Here is an example showing how we track the enqueue event at the NVMe/TCP transport layer. It defines three parameters: a pointer to a request struct, the NVMe command, and the queue id.

```
/* TRACE_EVENT Example */
TRACE_EVENT(nvme_tcp_queue_request,
    TP_PROTO(struct request *req,
             struct nvme_command* cmd,
             int qid),
    TP_ARGS(req, cmd, qid),
    TP_STRUCT__entry(...),
    TP_fast_assign(...),
    TP_printk(...));
```

We insert this tracepoint into the `nvme-tcp` module when `trace_nvme_tcp_queue_request` function is called.

```
trace_nvme_tcp_queue_request(
    blk_mq_rq_from_pdu(req),
    req->req.cmd,
    nvme_tcp_queue_id(queue));
```

Kernel Module and User-space Utility. `ntprof` introduce new kernel modules at the initiator and target side. Here is an example of the callback function for the above tracepoint. It is registered during the module initialization, and it will be executed once the tracepoint is triggered. The user-space utility provides a CLI interface, and it takes the developer's inputs, interacts with kernel modules, and reports profiling results.

```
void on_nvme_tcp_queue_request(
    void* ignore, struct request* req,
    struct nvme_command* cmd, int qid) {...}
```

Implementation Alternative. `ntprof` might also be implemented using eBPF [6] instead of a standalone kernel module. One could (1) define and insert the tracepoints to the kernel as described earlier; (2) define eBPF programs, which are functions attached to the predefined tracepoints using the `SEC` macro; (3) implement a user-space analyzer to read profiling records through the eBPF maps and generate the final profiling results. The eBPF programs can retrieve the tracepoint's parameters and store profiling records in eBPF maps [7].

Extension. One could also extend `ntprof` to support the kernel-bypass storage stack. For example, over SPDK NVMe-oF, we can use its tracing framework to define corresponding tracepoints [5, 21, 22], capture them via the eBPF mechanism, and reuse some of `ntprof`'s building blocks, like profiling agent, query protocol, and profiling analyzer.

4 Case Studies

4.1 Experimental Methodology

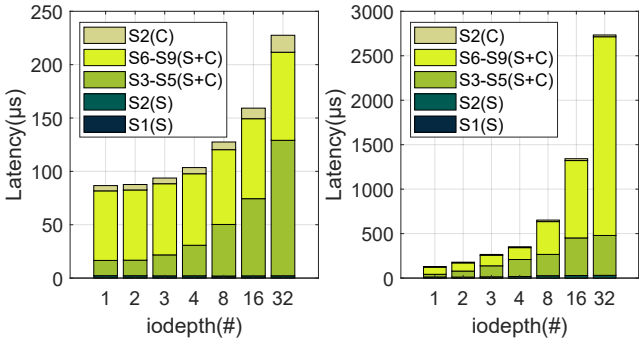
Testbed. We use two hardware setups. The first (testbed-1) consists of 2 sm110p servers from CloudLab [40]. Each node has a 32-core Intel Xeon Silver 4314 CPU, 128 GB DDR4, 1 NVIDIA/Mellanox CX6 25GbE NIC, 1 NVIDIA/Mellanox CX6 100GbE NIC, 1 Intel SATA SSD, and 4 Samsung (MZQL2960HCJR-00A07) 960GB NVMe SSDs. We configure its network MTU to 9KB. The second (testbed-2) includes 2 m510 servers from Cloudlab. Each node has a 16-core Intel Xeon D-1548 CPU, 64 GB of DDR4 memory, 1 Mellanox CX3 10 GbE NIC, 1 TOSHIBA (THNSN5256GPU7) 256 GB NVMe SSD. We disable the turbo boost and set the CPU frequency governor to "performance" in both cases. All nodes run Ubuntu 20.04 with kernel v5.15.143.

Workloads. Case studies 1–4 use synthetic workloads generated by `fio` v3.36. Case 5 runs the TSBS benchmark [20] over Apache IoTDB [4, 99] v1.2.0 in a write-heavy scenario. Case 6 runs the Filebench's [75] videoserver workload over F2FS [65] issuing read-heavy I/Os. We mainly examine I/O latency and throughput in each case. To demonstrate how `ntprof` reacts to different hardware settings, we conduct use case 1 on both testbeds. Due to space limitations, the results for the remaining use cases are presented only for testbed-1. The results for testbed-2 are presented in the Appendix A.3.

4.2 Use Case 1: Latency Breakdown

Description. `ntprof` can break down the end-to-end I/O latency into different stages (Figure 4) and provide useful system execution insights. Developers often complain about the unexpected poor performance in NVMe/TCP [15, 24], but the issue is sometimes unrelated to the protocol. More often, it's due to misconfigurations, such as having insufficient parallelism in the NVMe subsystem or mismounting a hard drive. Latency breakdown is helpful in these scenarios to pinpoint the root cause. Further, when introducing new software layers in the I/O path [55, 56, 77] or upgrading hardware, latency breakdown can help assess the impact of these changes.

Experimental Setup. On testbed-1, we break down the latency of a 4KB random read and 128KB sequential write



(a) 4K Random Read. **(b) 128K Sequential Write.**

Figure 6: Latency breakdown for the 4K random read and 128K sequential write as increasing the IO depth on testbed-1. We report the five most time-consuming parts. S1(S) refers to the time an I/O stays at the S1 stage on the submission path. S2(C) captures the time spent at the S2 stage on the completion path. S3–S5(S+C) aggregates the time spent in the S3, S4, and S5 stages across the submission and completion paths.

while increasing the iodepth from 1 to 32. We configure `fio` to run 1 job and issue requests to the Samsung drive in a direct I/O mode. The I/O engine is set to `libaio` for asynchronous operations. On testbed-2, we set the iodepth to 1, but increase the number of jobs from 1 to 32. We define the profiling task specification as follows: read(write) type, 4KB(128KB) block size, NVMe session named "nvme4n1" (the remote storage subsystem), a maximum of 10,000 profiling records, sampling frequency=1 per 1,000, grouping by NVMe session (all I/O sent to "nvme4n1" treated as one group), and `AVERAGE` as the aggregator. We launch the `fio` job with `ntprof`, which collects and analyzes profiling records and generates the report.

Result Analyses. Figure 6a illustrates the latency breakdown for 4K random read requests. As the iodepth varies from 1 to 32, "S3-S5(S+C)" increases the most, from 14.3 µs to 127.0 µs, indicating that queue is built up the networking substrate at both the initiator and target side. For example, when the iodepth equals 32, the above aggregated stages account for 92.2% of the total latency. The reason why `ntprof` can capture it is because our installed profiling agent takes timestamps at two tracepoints at the NVMe/TCP initiator (`_try_send_data` and `_handle_c2h_data`) and two tracepoints (`_done_recv_pdu` and `_try_send_reponse`) at the target, which allows the profiling agent to dissect the latency contribution from one or several stages (Figure 4). Figure 6b shows the latency breakdown for 128K sequential writes. "S6-S9(S+C)" becomes the most time-consuming part, rising from 79.2 µs to 2234.9 µs, as the iodepth increases from 1 to 32. `ntprof` is capable to further break down "S6-S9(S+C)" and reveal that the "S5-S9(S+C)" on the target side takes 95.9% of the time when iodepth is 32. Similarly, such capability comes from the fact that `ntprof` records the triggering time of the two key tracepoints (`_exec_write_req` and `_queue_response`) at the target side. Figure 11a (Appendix A.3) presents the I/O latency breakdown on testbed-2.

Takeaways. `ntprof` models the NVMe/TCP data path as a chain of software switches, where each can be equipped with a programmable profiling agent that uses tracepoints to collect statistics. This allows us to dissect the path segment and perform time accounting in a fine-grained way, and makes `ntprof` to operate across a wide range of hardware models.

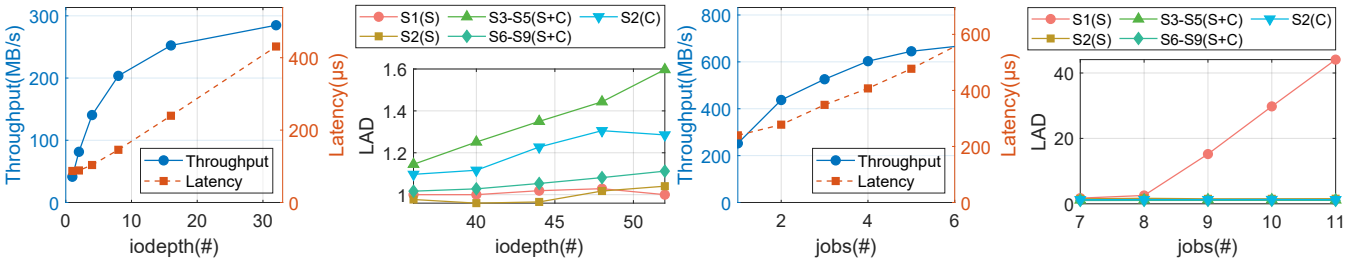
4.3 Use Case 2: Software Bottleneck Localization

Description. NVMe/TCP interacts with several vital OS kernel modules (e.g., block layer, TCP/IP networking stack, and NVMe subsystem). Any system bottleneck from multi-core execution, TCP connection, or NVMe submission/completion command delivery could throttle the achieved I/O throughput, yielding a drastic latency increase. These issues are sometimes subtle, but hard to detect and reproduce, especially given that (a) application I/O ordering is not preserved; (b) multi-tenant deployments are common. Accurate software stack bottleneck localization is a demanding feature for developers.

Experimental Setup. We configure two bottleneck scenarios. One is at the NVMe/TCP target core, where each NVMe drive is provisioned with one target core regardless of the I/O load. We launch a 4KB random read stream and gradually increase the iodepth. The other is at the TCP connection, which carries an increasing number of I/O streams. We spawn multiple `fio` jobs (iodepth=16) and make sure they are pinned to different cores on the initiator side. Both cases are common in an enterprise/cloud disaggregated setting for cost efficiency. The profiling task specification is similar to the first case (§4.2), except for the generated report format.

Result Analyses. Figure 7a depicts the performance when iodepth varies from 1 to 32. We observe a marginal throughput increase (from 252.4 MB/s to 284.9 MB/s) when the iodepth is beyond 16, while latency rises significantly, from 238.7 µs to 431.0 µs, indicating that a system bottleneck occurs. `ntprof` introduces a new metric called *Latency Amplification Degree (LAD)*–defined as T_{cong}/T_{base}, where T_{cong} and T_{base} refer to processing latency at the overloaded and unloaded scenarios–to localize the bottleneck. As shown in Figure 7b, when iodepth is 52, the "S3-S5(S+C)" sees a 1.6 LAD, dominating other parts. `ntprof` can capture this because our profiling agent records timestamps at two tracepoints on the NVMe/TCP target (`_try_send_data` and `_try_send_response`), allowing it to report the time spent in these combined stages. Further LAD analysis shows that "S6(C)" latency increases from 26.38 µs to 47.14 µs (LAD=1.8). This happens because, for the switching queue of S6, the target is not fast enough to send the I/O completion, causing queue buildup and latency increase, indicating the target core becomes the bottleneck. `ntprof` uses the `_done_recv_pdu` tracepoint on the target side to achieve this.

Figure 7c shows that when the number of `fio` jobs increases from 5 to 6, throughput only rises by 3.1%, from 645.2 MB/s to 665.5 MB/s, while latency rises from 477.3 µs to 556.7 µs. As we consolidate jobs from 6 to 11, as shown



(a) Throughput and Latency. **(b) LAD.** **(c) Throughput and Latency.** **(d) LAD.**

Figure 7: Software bottleneck localization. (a) and (b) depict the target core bottleneck case, where we run one `fio` job and increase its iodepth from 1 to 64. (c) and (d) shows the TCP connection bottleneck case, where we increase the number of consolidated `fio` jobs (with iodepth=16) from 1 to 11. LAD=Latency Amplification Degree.

in Figure 7d, the LAD of "S1" rises to 51. Such a latency increase indicates that there are inadequate TCP connections to deliver I/Os from the initiator to the target. `ntprof` locates this through a key tracepoint on the initiator (`_queue_rq`).

Takeaways. `ntprof` views each system component as a buffered software switch. A system bottleneck causes queueing buildup at one or several molded queues (§3.2). `ntprof` thus dissects the queueing delay and introduces a new metric (latency amplification degree) for bottleneck localization.

4.4 Use Case 3: Hardware Bottleneck Localization

Description. The underlying hardware infrastructure can also be a system bottleneck. For example, people find that a malfunctioned NIC [31,64,67] can reduce its bandwidth capacity dramatically. A fragmented NVMe SSD would deliver much lower throughput in a read/write or write-only workload. Further, hardware heterogeneity has become common in the data center, which can also cause hardware bottlenecks.

Experimental Setup. First, we create a "slow" SSD. On the initiator side, we run 1 `fio` job (referred to as "remote `fio`") issuing 128K sequential writes with an iodepth of 10. Simultaneously, we run 1 `fio` job on the target side (referred to as "local `fio`") also performing 128K sequential write workloads to the same drive. Second, we create a "slow" NIC. We run two `fio` instances on the initiator side, each targeting a different remote SSD. Both `fio` instances issue 128K random reads with 8 jobs and an iodepth of 5. Additionally, we run `iperf3` clients on the target side, sending data to the initiator at 100Gbps. The profiling spec is the same as before.

Result Analyses. Figure 12a (Appendix A.3) shows the performance when local `fio` iodepth increases from 2 to 8. The aggregated throughput reaches 1440.0 MB/s, maxing out the SSD bandwidth limit. Compared to no local `fio` jobs, the bandwidth shared by remote `fio` decreases to 627.2 MB/s, while the latency increases from 859.7 µs to 1984.6 µs. Figure 12b shows when local `fio` iodepth is 8, "S6-S9(S+C)" sees a 3.6 LAD, significantly higher than the other parts. `ntprof` can locate this because the profiling agent uses 2 tracepoints on the target (`_don_recv_pdu` and `_try_send_response`). A further drill-down shows that "S7-9(S+C)" is the primary contributor, with its latency increased from 303.8 µs to 1729.2 µs (LAD=5.7). This occurs because the local `fio` pushes more I/Os into the "S7-9(S+C)" stage while the SSD is already operating at its maximum speed, causing the queue to build up and latency to rise. `ntprof` is capable of capturing this because it records timestamps of `_exec_eritw_req` and `_queue_response` on the target.

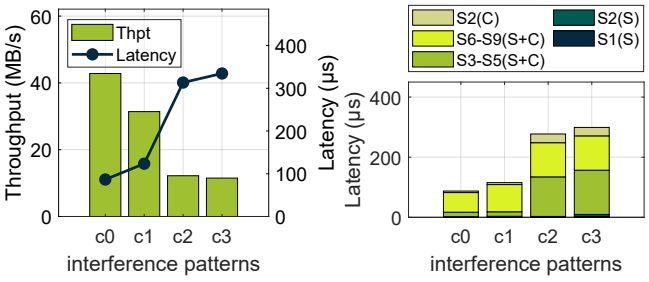
Figure 12c shows the aggregated throughput of `fio` and `iperf3` approaches 100 Gb/s, indicating the NIC is the bottleneck. The latency of `fio` I/O rises from 834.9 µs to 2803.5 µs when there are 16 `iperf3` clients, compared to when there are none. Figure 12d shows "S3-S5(S+C)" increases most significantly when increasing `iperf3` clients. It sees a 4.5 LAD when there are 16 clients. That is because on the completion path, the NVMe/TCP network flows share less bandwidth when there are more `iperf3` clients, leading to queue buildup. `ntprof` can capture it because we take timestamps at two tracepoints (`_try_send_data` and `_handle_c2h_data`) at the NVMe/TCP initiator and two tracepoints (`_done_recv_pdu` and `_try_send_reponse`) at the target.

Takeaways. `ntprof` views the SSD or NIC as one stage. Its queueing model is inherent from the upstream storage, making `ntprof` capable of localizing the hardware bottleneck.

4.5 Use Case 4: Interference Analysis

Description. Multi-tenancy is a pressing concern for storage disaggregation. When latency-sensitive I/O streams co-locate with throughput-oriented ones, one would observe significant head-of-line blocking and unfair bandwidth partition [56,77].

Experimental Setup. We run 2 `fio` instances on the target side. The latency-sensitive workload is configured to issue 4K random read requests with 1 job (iodepth=1). The throughput-sensitive one issues 128K random read requests to the same NVMe drive. We design 3 interference patterns. For case "c1", the two I/O flows are pinned to different cores on the initiator side, each using a separate NVMe/TCP session. For case "c2", they are pinned to the same core and therefore share the same NVMe/TCP session. For case "c3", they are pinned to different cores but share the same NVMe/TCP session. We define the profiling task specification as follows: read type, 4KB block size, NVMe session named "nvme4n1", a maximum of 10,000 profiling records, sampling frequency=1 per 1,000, grouping by NVMe session, and `AVERAGE` as the aggregator. We launch the `fio` job with `ntprof`, which collects



(a) Throughput and Average Latency. (b) Latency Breakdown.

Figure 8: Latency breakdown and throughput analysis of the 4K random read I/O flow when introducing a 128K random read I/O in different scenarios. There is only the 4K read I/O flow running in "c0". "c1" Two I/O flows are pinned to different cores and using different NVMe/TCP sessions in "c1". Both flows are pinned to the same core, sharing the same NVMe/TCP session in "c2". Both flows are pinned to different cores but share the same NVMe/TCP sessions in "c3".

and analyzes profiling records and generates the report.

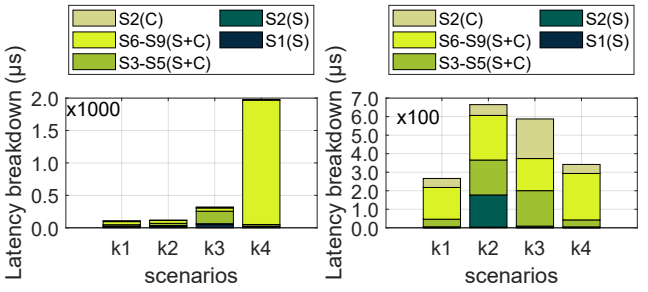
Result Analyses. Figure 8a shows the latency of 4KB reads is increased in all interference cases, with a 26.7%, 71.5%, and 73.1% throughput degradation in "c1", "c2", and "c3", respectively. Figure 8 shows the latency breakdown of the 4KB read requests. In "c1", the "S6-S9(S+C)" latency rises the most, from 65.4 µs to 90.6 µs. A further breakdown reveals that the majority of this increase, 25.2 µs, comes from the "S7-S9(S+C)". A queue is built up at the NVMe/TCP session at the target due to the increasing I/O load. `ntprof` captures this because it utilizes the timestamp when the 2 tracepoints on the target are triggered (`_exec_read_req` and `_queue_response`). In "c2", "S3-S5(S+C)" increases the most, from 14.9 to 131.6 µs, also due to "S6-S9(S+C)". The impact of latency in "c3" is similar to "c2", where "S3-S5(S+C)" is increased to 114.2 µs.

Takeaways. `ntprof` enables profiling at different I/O granularities with separated profiling paths. Under mixed I/O traffic, `ntprof` generates different profiling records to collect statistics, making interference analysis possible.

4.6 Use Case 5: Apache IoTDB

Description. Apache IoTDB [4, 59, 99] is an LSM-tree-based high-performance time-series database designed for IoT scenarios. Storage disaggregation gains significant traction in such settings given that edge computing nodes have limited storage resources. We used the Time Series Benchmark Suite (TSBS) [20, 89] to generate DevOps workloads to evaluate the performance of time-series databases. By using the default parameters, the workload is set to contain 4000 devices' information running for 3 days, with each device generating multiple time-series metrics, such as CPU utilization, memory usage, disk I/O, and network activity, respectively.

Experimental Setup. Both the Apache IoTDB server and client run on the initiator node, with IoTDB's data directory mounted to a remote Samsung SSD. Using the default con-



(a) Use case: IoTDB. (b) Use case: F2FS.

Figure 9: I/O Latency breakdowns for real-world applications under different scenarios. k1 is the baseline, where there are 32 NVMe/TCP sessions. The number is reduced to 1 in k2. 8 `iperf3` clients are running in k3, and `fio` workload is running on the target side in k4. The IoTDB and F2FS show the latency breakdown for 4KB writes and 128KB reads.

figuration of TSBS, we run 32 workers to write data into the database. We analyze the latency breakdown of all write I/Os under the following scenarios: (1) "k1" uses 32 NVMe-TCP sessions, (2) "k2" pins all I/O to a single NVMe/TCP session, (3) in "k3", there are 8 `iperf3` clients on the initiator side, each sending data to one of 8 servers on the target side, emulating a NIC bottleneck, and (4) in "k4" two `fio` jobs run on the target side, issuing 128 KB sequential writes to the same SSD (iodepth=10), creating an SSD bottleneck. We configure `ntprof` similarly to §4.2, except that we monitor ANY types of write I/Os, and results are grouped by I/O type/size.

Result Analysis. The TSBS benchmark reports the throughput of IoTDB as follows: 10.6M matrices/s in k1, 10.4M in k2, 8.7M in k3, and 9.6M in k4. `ntprof` shows different sized writes are issued, ranging from 4KB to several MB. `ntprof` reports this because it generates different profiling records of different types of I/O. Across all scenarios, 4 KB write requests constitute the majority, 80.9% in scenario k1, 80.2% in k2, 77.5% in k3, and 80.1% in k4. Figure 9a shows the latency breakdown for 4KB writes (since it dominates the write traffic). k3 sees a drastic increase in "S3-S5(S+C)". That is because NVMe/TCP sessions are competing with `iperf3` flows. The queue is built up in stage S3. In k4, "S6-S9(S+C)" increases most significantly, from 47.8 µs to 1917.3 µs. A further breakdown shows an 1859.7 µs increase in "S7-S9(S+C)" is the primary contributor, indicating that the background I/O streams consume bandwidth and make SSD a bottleneck.

Takeaways. Real-world applications exhibit diverse I/O profiles. `ntprof` exposes a rich task specification interface for developers, translates these profiling requirements into predicates, and installs them into the modeled software switch. As such, it enables concurrent heterogeneous I/O profiling.

4.7 Use Case 6: F2FS

Description. F2FS [65] is a Linux file system designed for modern flash storage devices. F2FS performs exceptionally well with NVMe SSDs, as it efficiently handles the inherent characteristics of flash memory, such as wear-leveling and

garbage collection. In this case study, we focus on analyzing how file I/Os traverse the NVMe/TCP. We take Filebench [75] and run the 'videoserver" workload, which emulates a video streaming server managing multiple concurrent streams.

Experimental Setup. We configure the "videoserver" to run 32 threads issuing 128KB read requests to the active videos, while there is another thread issuing 1MB write requests to the passive videos. The video size is set to 10 GB. Direct I/O is enabled. We also run the application under the 4 scenarios as described in §4.6, except in k3, the `iperf3` clients are on the target side, while the servers are on the initiator side. We use the same profiling task specification and focus on reads.

Result Analysis. Filebench reports 117µs per operation in k1, 319 µs in k2, 235 µs in k3 and 114 µs in k4. Figure 9b shows the latency breakdown for 128KB read I/O. Comparing to k1, "S2(S)" in k2 increases significantly, from 4.6 µs to 173.5 µs. This is because a single TCP connection in the nvme-tcp queue struggles to handle the read I/Os submitted by all the cores, indicating the need to increase the number of NVMe/TCP queues. Akin to §4.2, `ntprof` locates this via two tracepoints (`_queue_request` and `_try_send_cmd_pdu`). "S6-S9(S+C)" is increased from 171.3 µs to 240.6 µs. A further breakdown shows that "S6(C)" is increased by 55.2 µs, dominating the entire latency. That is because by reducing NVMe/TCP sessions we also reduce the dequeuing rate in S6 on the completion path. `ntprof` uses two tracepoints (`_queue_response` and `_setup_response_pdu`) on the target side for analysis. In k3, "S3-S5(S+C)" is increased from 41.0 µs to 191.1 µs. The reason is similar to the case in §4.6. In addition, "S2(C)" is increased from 48.7 µs to 213.8 µs, shown via 2 tracepoints on the initiator side (`_handle_c2h_data` and `_process_nvme_cqe`). In k4, "S6-S9(S+C)" is increased from 171.3 µs to 250.8 µs, aligning with our scenario design.

Takeaways. `ntprof` enables hierarchical I/O path analysis due to our Clos network view. Similar to the Flame Graph [8], one can gradually drill the I/O path segment using different metrics and locate the system bottleneck.

4.8 Overhead Analysis

`ntprof` operates at the data plane and interacts with critical system components. It is important to minimize overhead, including memory and CPU usage. We configure four sets of `fio` experiments (Appendix A.3) and measure the application performance, CPU usage, and memory usage when enabling and disabling (Table 5). Across all workloads, the overhead introduced by `ntprof` is marginal. The CPU usage increases by 0.6% and 2.9% for the read and write cases. We observe up to 17MB more memory usage during the profiling.

5 Related Work

Network Telemetry. `ntprof` benefits from prior network telemetry systems [32, 48, 54, 74, 81, 93, 101, 103, 104, 106]. For example, Sonata [48] develops a declarative interface to express telemetry tasks and partitions query execution to streaming processors and switches. [32, 68, 72, 74, 81, 90, 91, 101, 104, 106] introduce probabilistic data structures (like Sketch) to capture high-volume data streams with low storage on the programmable data plane, i.e., in-network telemetry.

Profiling Systems. Many software utilities are developed to identify code hotspots, analyze concurrency dependency, and break down stalled cycles [9, 13, 18, 19, 38, 45, 66, 102]. For example, Linux `perf` [38] is a widely used tool to instrument CPU performance counters, tracepoints, and probes and reports application execution statistics. Intel VTune [9] takes a top-down analysis strategy [102] and drills down the performance analysis mostly using architectural counters. Some are also integrated into the language system facilitating application development [13, 35, 46, 61, 70, 73, 82–84, 87, 100]. We focus on profiling and analyzing the NVMe/TCP protocol.

Storage Disaggregation. People have explored disaggregated storage extensively given the rising networking bandwidth and fast remote storage protocol [47, 49, 50, 52, 55, 62, 63, 71, 77–79, 105]. Ana Klimovic *et al.* characterize the performance of iSCSI-based disaggregated storage [62]. i10 [55] develops an efficient in-kernel TCP/IP remote storage based on dedicated end-to-end IO paths and delayed doorbell notifications. Researchers [49] also report the performance characteristics of server-based NVMe-oF boxes.

Intra-host Diagnosis. The host interconnect is becoming a bottleneck under high-bandwidth networks. Researchers have developed benchmarking frameworks and diagnostic tools to analyze it [29, 30, 51, 53, 67, 69, 97]. For example, Saksham Agarwal et al. [29, 30] analyze the host congestion issues and build the host congestion control protocol. NSight [51], leveraging CPU profilers, captures sources of overheads in the end-host stack at the nanosecond scale. Hostping [67] monitors and diagnoses intra-host bottlenecks in RDMA networks. `ntprof` can use them to improve its efficiency.

6 Conclusion

This paper presents a profiling and development utility for NVMe/TCP. It enables developers to understand and analyze the execution characteristics of the NVMe/TCP systematically. The key idea is to view the NVMe/TCP storage substrate as a lossless switched network and apply network monitoring techniques. `ntprof` models each on-path system module as a software switch, integrates a programmable profiling agent on the data plane, and introduces a proactive query interface for statistics collection and analysis. We build `ntprof` over Linux kernel and apply it for six case studies.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Soujanya Ponnapalli, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, and CAREER-2339755.

References

[1] Linux Kernel 5.0 Release. https://kernelnewbies.org/Linux_5.0, 2019.

[2] x-cellent technologies Selects Lightbits for metalstack.cloud. https://www.lightbitslabs.com/press-releases/x-cellent-technologies-selects-lightbits-for-metalstack-cloud/, 2022.

[3] How We Built It: Block Storage for AI/ML Workloads, Powered by Lightbits. https://crusoe.ai/blog/how-we-built-it-block-storage-for-ai-ml-workloads-powered-by-lightbits/, 2023.

[4] Apache IoTDB: Database for Internet of Things. https://iotdb.apache.org/, 2024.

[5] BPF tracing with SPDK. https://spdk.io/news/2021/09/28/bpf_tracing_with_spdk/, 2024.

[6] eBPF. https://ebpf.io/, 2024.

[7] eBPF Maps. https://docs.ebpf.io/linux/concepts/maps/, 2024.

[8] Flame Graphs. https://www.brendangregg.com/flamegraphs.html, 2024.

[9] Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.ed42rh, 2024.

[10] Linux Kernel Tracepoints. https://docs.kernel.org/trace/tracepoints.html, 2024.

[11] Lustre's Benchmarking. https://wiki.lustre.org/Category:Benchmarking, 2024.

[12] Nebul Chooses Lightbits to Deliver Powerful AI Cloud. https://www.lightbitslabs.com/press-releases/nebul-chooses-lightbits-for-ai-cloud-data-platform/, 2024.

[13] NVIDIA Nsight. https://developer.nvidia.com/tools-overview, 2024.

[14] Port mirroring. https://en.wikipedia.org/wiki/Port_mirroring, 2024.

[15] Proxmox and NVMe/TCP. https://forum.proxmox.com/threads/proxmox-and-nvme-tcp.149160/, 2024.

[16] RocksDB's Benchmarking Tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools/, 2024.

[17] The Block Layer I/O Tracing Utility. https://linux.die.net/man/8/blktrace, 2024.

[18] The iostat Utility. https://linux.die.net/man/1/iostat, 2024.

[19] The PAPI Performance Application Programming Interface. https://icl.utk.edu/papi/, 2024.

[20] Time Series Benchmark Suite (TSBS). https://github.com/benchANT/tsbs, 2024.

[21] Tracing Framework Guide Part 1. https://spdk.io/spdk_trace/2022/09/26/tracing-library-guide-pt1/, 2024.

[22] Tracing Framework Guide Part 2. https://spdk.io/spdk_trace/2022/11/25/tracing-library-guide-pt2/, 2024.

[23] Valgrind: an Instrumentation Framework for Building Dynamic Analysis Tools. https://valgrind.org, 2024.

[24] Very slow write speed (NVME drive) on 10G network. https://serverfault.com/questions/1065820/very-slow-write-speed-nvme-drive-on-10g-network, 2024.

[25] FI-TS Delivers Disaggregated Storage Platform. https://www.intel.com/content/www/us/en/customer-spotlight/stories/finanz-informatik-technologie-customer-story.html, 2025.

[26] Lightbits Labs: Software-Defined Storage. https://www.lightbitslabs.com, 2025.

[27] Lightbits Powers One of the Worlds Largest eCommerce Platforms. https://www.lightbitslabs.com/wp-content/uploads/2023/06/LBITS-ecommerce-case-study-digital-LBCS02-2023-03.pdf, 2025.

[28] Zenlayer Accelerates Edge Cloud Services. https://www.intel.com/content/www/us/en/customer-spotlight/stories/zenlayer-customer-story.html, 2025.

[29] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher De Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, et al. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 198–204, 2022.

[30] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 275–287, 2023.

[31] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[32] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM'20)*, pages 662–680, 2020.

[33] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery.

[34] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[36] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[37] William E Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 2004.

[38] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[39] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/, 2024.

[40] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[41] NVM Express. Nvm express base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf, 8 2024.

[42] NVM Express. Nvm express overview. https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf, 8 2024.

[43] NVM Express. Nvm express tcp transport specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-TCP-Transport-Specification-Revision-1.1-2024.08.05-Ratified.pdf, 8 2024.

[44] Python Software Foundation. The python profilers. https://docs.python.org/3/library/profile.html, 2024.

[45] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, jun 1982.

[46] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, page 916–929, 2023.

[47] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power,

Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 1012–1027, 2023.

[48] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication (SIGCOMM'20)*, pages 357–371, 2018.

[49] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–9, 2017.

[50] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. Performance characterization of nvme-over-fabrics storage disaggregation. *ACM Trans. Storage*, 14(4), dec 2018.

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

[52] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, page 33–47, 2023.

[53] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 297–312, 2024.

[54] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM'20)*, pages 404–421, 2020.

[55] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

[56] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 113–128, 2021.

[57] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: using packets for low latency network programming and visibility. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, page 3–14, 2014.

[58] Vimalkumar Jeyakumar, Mohammad Alizadeh, Changhoon Kim, and David Mazières. Tiny packet programs for low-latency network control and monitoring. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

[59] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. Separation or not: On handing out-of-order time-series data in leveled lsm-tree. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 3340–3352. IEEE, 2022.

[60] David G Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics*, pages 338–354, 1953.

[61] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 535–546. IEEE, 2010.

[62] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, 2016.

[63] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 345–359, 2017.

[64] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 31–48, Boston, MA, April 2023. USENIX Association.

[65] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[66] David D Levine. Iometer user's guide. *Intel Server Architecture Lab*, 40, 1998.

[67] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. Hostping: Diagnosing intra-host network bottlenecks in RDMA servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 15–29, Boston, MA, April 2023. USENIX Association.

[68] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[69] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[70] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, page 318–333, 2019.

[71] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 305–323, 2020.

[72] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 795–809, 2017.

[73] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 363–378, 2019.

[74] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[75] Richard McDougall and Jim Mauro. Filebench. *URL: http://www. nfsv4bat. org/Documents/nasconf/2004/filebench. pdf (Cited on page 56.)*, 2005.

[76] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*, pages 753–766, 2022.

[77] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, page 106–122, 2021.

[78] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[79] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage*, 20(3), June 2024.

[80] J. Nagle. On Packet Switches with Infinite Storage. *IEEE Transactions on Communications*, 35(4):435–438, 1987.

[81] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 85–98, 2017.

[82] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 663–679, 2018.

[83] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, page 16–22, 2020.

[84] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated SmartNIC Offloading Insights for Network

Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 772–787, 2021.

[85] Linux RDMA. qperf. https://github.com/linux-rdma/qperf, 2018.

[86] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 30(4):65–79, 2010.

[87] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 740–755, 2021.

[88] Steve Scott, Dennis Abts, John Kim, and William J Dally. The blackwidow high-radix clos network. *ACM SIGARCH Computer Architecture News*, 34(2):16–28, 2006.

[89] Daniel Seybold and Jörg Domaschka. Benchmarking-as-a-service for cloud-hosted dbms. In *Proceedings of the 22nd International Middleware Conference: Demos and Posters*, Middleware '21, page 12–13, New York, NY, USA, 2021. Association for Computing Machinery.

[90] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 1–16, 2018.

[91] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 685–699, 2020.

[92] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 783–799, 2024.

[93] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. In-band network telemetry: A survey. *Computer Networks*, 186:107763, 2021.

[94] Ej technologies. The definitive guide to jprofiler. https://www.ej-technologies.com/resources/jprofiler/help/doc/JProfiler.pdf, 2024.

[95] David L Tennenhouse, Jonathan M Smith, W David Sincoskie, David J Wetherall, and Gary J Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[96] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, oct 2007.

[97] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the Host Network. In *Proceedings of the ACM SIGCOMM 2024 Conference (SIGCOMM'24)*, page 581–594, 2024.

[98] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 449–462, 2020.

[99] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. Apache iotdb: A time series database for iot applications. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[100] Ziyang Xu, Yebin Chon, Yian Su, Zujun Tan, Sotiris Apostolakis, Simone Campanoni, and David I August. Prompt: A fast and extensible memory profiling framework. *Proceedings of the ACM on Programming Languages*, 8:449–473, 2024.

[101] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*, pages 561–575, 2018.

[102] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[103] Minlan Yu. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review*, 49(1):11–17, 2019.

[104] Minlan Yu, Lavanya Jose, and Rui Miao. Software {Defined}{Traffic} Measurement with {OpenSketch}. In *10th USENIX symposium on networked systems design and implementation (NSDI'13)*, pages 29–42, 2013.

[105] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

[106] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

A Appendix

A.1 Tracepoint Parameters

Table 3 and Table 4 show the parameters of our added tracepoints in the `nvme-tcp` and `nvmet-tcp` module, respectively.

Tracepoint	Parameters
_queue_rq	struct request *req, void *pdu, int qid, struct llist_head *req_list, struct list_head *send_list, struct mutex *send_mutex
_queue_request	struct request *req, struct nvme_command* cmd, int qid
_try_send_cmd_pdu	struct request *rq, struct socket *sock, int qid, int len
_try_send_data_pdu	struct request *rq, void *pdu, int qid
_try_send_data	struct request *rq, void *pdu, int qid
_done_send_req	struct request *rq, int qid
_handle_c2h_data	struct request* rq, int qid, int data_remain, u64 recv_time
_recv_data	struct request* rq, int qid, int len
_process_nvme_cqe	struct request *rq, void* pdu, int qid, u64 recv_time
_handle_r2t	struct request *rq, void* pdu, int qid, u64 recv_time
_try_send	struct request *rq, int qid

Table 3: Parameters of tracepoints in the module `nvme-tcp` (initiator).

Tracepoint	Parameters
_done_recv_pdu	struct nvme_tcp_cmd_pdu *pdu, int qid, long long recv_time
_exec_read_req	struct nvme_command *cmd, int qid
_exec_write_req	struct nvme_command *cmd, int qid, int size
_queue_response	struct nvme_command *cmd, int qid
_setup_c2h_data_pdu	struct nvme_completion *cqe, int qid
_setup_r2t_pdu	struct nvme_command *cmd, int qid
_setup_response_pdu	struct nvme_completion *cqe, int qid
_try_send_data_pdu	struct nvme_completion *cqe, void *pdu, int qid, int size
_try_send_r2t	struct nvme_command *cmd, void *pdu, int qid, int size
_try_send_response	struct nvme_completion *cqe,void *pdu, int qid, int size
_try_send_data	struct nvme_completion *cqe, int qid, int size
_handle_h2c_data_pdu	struct nvme_tcp_data_pdu *pdu, struct nvme_command *cmd, int qid, int size, long long recv_time
_try_recv_data	struct nvme_command *cmd, int qid, int size, long long recv_time

Table 4: Parameters of tracepoints in the module `nvmet-tcp` (target).

A.2 Triggered Events of an NVMe/TCP I/O

Figure 10 enumerates the ordered list of events that an I/O may experience throughout its lifetime in the `nvme-tcp` layer. Some events may occur multiple times, indicated by "1+". Generally, all types of I/O begin with `QUEUE_RQ` and end with `PROCESS_NVME_CQE`.

A.3 More Evaluation

Latency Breakdown on Testbed-2. In Figure 11a, we observe similar breakdown results as in Figure 6a when the number of jobs increases in testbed-2. As shown in Figure 11b, for 128KB sequential writes, "S3-S5(S+C)" becomes the dominant component when the job count exceeds 16, indicating

read I/O

Event Name	Num
QUEUE_RQ	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_CMD_PDU	1
DONE_SEND_REQ	1
HANDLE_C2H_DATA	1
RECV_DATA	1+
PROCESS_NVME_CQE	1

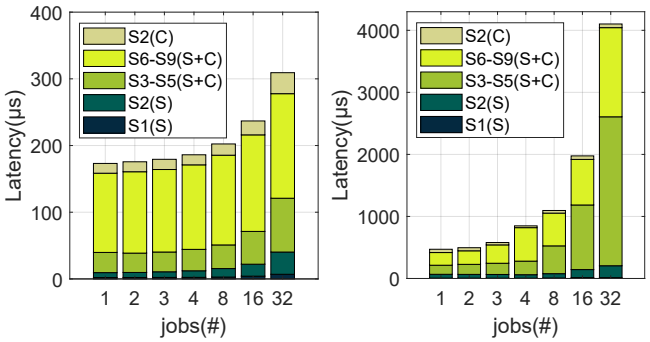
write I/O without inline data

Event Name	Num
QUEUE_RQ	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_CMD_PDU	1
DONE_SEND_REQ	1
HANDLE_R2T	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_DATA_PDU	1
TRY_SEND_DATA	1+
DONE_SEND_REQ	1
PROCESS_NVME_CQE	1

write I/O with inline data

Event Name	Num
QUEUE_RQ	1
QUEUE_REQUEST	1
TRY_SEND	1
TRY_SEND_CMD_PDU	1
TRY_SEND_DATA	1+
DONE_SEND_REQ	1
PROCESS_NVME_CQE	1

Figure 10: Event series of different types of I/O in the `nvme-tcp` layer. The event name corresponds to the tracepoint name, omitting the "nvme_tcp_" prefix.



(a) 4K Random Read. **(b) 128K Sequential Write.**

Figure 11: Latency breakdown for the 4K random read and 128K sequential write as increasing the number of jobs on testbed-2. We report the five most time-consuming parts. S1(S) refers to the time an I/O stays at the S1 stage on the submission path. S2(C) captures the time spent at the S2 stage on the completion path. S3–S5(S+C) aggregates the time spent in the S3, S4, and S5 stages across the submission and completion paths.

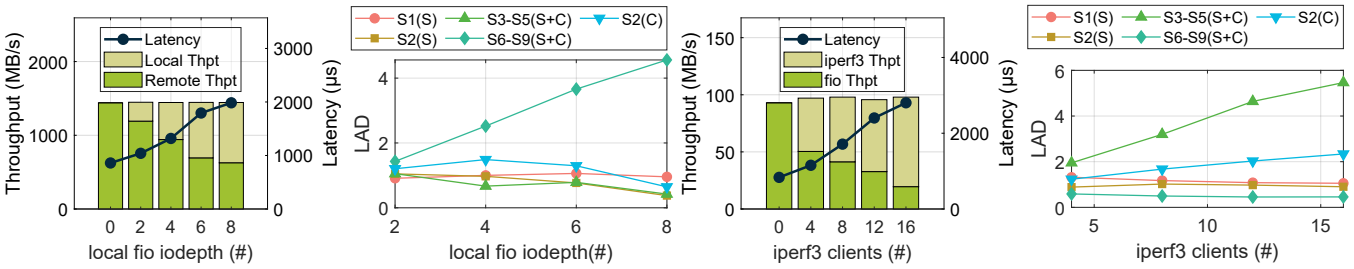
workload	without `ntprof`		with `ntprof`			
	lat(µs)	bw(MB/s)	lat(µs)	bw(MB/s)	imem(MB)	tmem(MB)
4k read	449.26	2202.91	460.57	2099.11	12	10
4k write	66.75	883.91	69.47	835.86	1	7
128k read	1136.72	3493.03	1136.24	3491.88	13	3
128k write	338.75	1439.89	337.44	1439.17	1	17

Table 5: Performance comparison with and without `ntprof`. imem represents the memory usage increase on the initiator side, while tmem indicates the increase on the target side. lat and bw denote the average latency and bandwidth, respectively.

that the queue builds up during data transmission through the network stack. When there are 32 jobs, 2400.4µs is spent out of 4102.2µs of end-to-end latency.

Use Case 3 Results. Figure 12 presents our hardware bottleneck localization results.

Experimental setup for `ntprof` overhead analysis. We configure four sets of `fio` experiments to analyze the `ntprof` overheads: (a) 4K random read, with 16 jobs and an iodepth of 16; (b) 4K sequential write, 4 jobs and an iodepth of 4; (c) 128K sequential read, with 4 jobs and an iodepth of 8; (d) 128K sequential write, with 2 jobs and an iodepth of 2. Each job is assigned to a different core. These parameters were selected based on experiments designed to prevent any



(a) Throughput and Latency. **(b) LAD.** **(c) Throughput and Latency.** **(d) LAD.**

Figure 12: Hardware bottleneck localization. (a) and (b) depict the "slow SSD" case, where we run one extra `fio` job on the target and increases the iodepth from 2 to 8. (c) and (d) show the "slow NIC" case, where we run `iperf3` server on the initiator side and clients on the target sides, and we increase the TCP flows from 4 to 16. LAD=Latency Amplification Degree.

component in the I/O path from becoming a bottleneck, while applying sufficient load to stress the system. The `fio` tests ran for 60 seconds, during which total memory consumption was recorded every second, and an average was calculated at the end of each run.

Building an Elastic Block Storage over EBOFs Using Shadow Views

Sheng Jiang[1] and Ming Liu[2]

[1]Carnegie Mellon University [2]University of Wisconsin-Madison

Abstract

The EBOF (Ethernet-Bunch-Of-Flash) has emerged as an enticing and promising disaggregated storage platform due to its streamlined I/O processing, high scalability, and substantial energy/cost-efficiency improvement. An EBOF applies a smart-sender dumb-receiver design philosophy and provides backward-compatible storage volumes to expedite system deployment. Yet, the static and opaque internal I/O processing pipeline lacks resource allocation, I/O scheduling, and traffic orchestration capabilities, entailing bandwidth waste, workload non-adaptiveness, and performance interference.

This paper presents the design and implementation of a distributed telemetry system (called **shadow view**) to tackle the above challenges and facilitate the effective use of an EBOF. We model an EBOF as a two-layer multi-switch architecture and develop a view development protocol to construct the EBOF running snapshot and expose internal execution statistics at runtime. Our design is motivated by the observation that fast data center networks make the overheads of inter-server communication and synchronization negligible. We demonstrate the effectiveness of shadow view by building a block storage (dubbed **Flint**[1]) atop EBOFs. The enhanced I/O data plane allows us to develop new three techniques–an elastic volume manager, an eIO scheduler, and a view-enabled bandwidth auction mechanism. Our evaluations using the Fungible FS1600 EBOF show that a Flint volume achieves 9.3/9.2 GB/s read/write bandwidth with no latency degradation, significantly outperforming the defacto EBOF volume. It achieves up to 2.9× throughput improvements when running an object store. Flint is tenant-aware and remote target-aware, delivering efficient multi-tenancy and workload adaptiveness.

1 Introduction

Storage disaggregation has gained increasing attraction in the past few years because of its independent scaling, high utilization, and cost-efficiency improvement. With the advent of 400+GbE network and fast remote storage protocols, a remote storage server can deliver millions of IOPS at tens to hundreds of microseconds, approaching the performance of a direct-attached storage system. We have seen a rising number of deployments [15, 52, 72, 76] at public clouds, enterprise on-premise clusters, and edge data centers.

Ethernet-Bunch-Of-Flash (EBOF) is an emerging disaggregated storage solution that packs an Ethernet switch with NVMe drives into one system SoC (System-on-a-Chip). Compared with today's server JBOFs (Just-a-Bunch-Of-Flash), an EBOF implements hardware-assisted remote I/O processing pipelines, elides the power-hungry general-purpose CPUs on the I/O path, and increases the I/O scalability by eliminating the DRAM and PCIe subsystems, significantly improving the system energy efficiency. For example, the Fungible FS1600 (our evaluation target) achieves 200K and 5.9K IOPS per Joule for reads and writes, nearly outperforming existing server storage appliances by one order of magnitude!

However, there is no well-trodden understanding of the EBOF capabilities and limitations, hindering deploying this new storage platform. An EBOF applies the *smart-sender dumb-receiver* design philosophy and provides backward-compatible volume-oriented storage functionalities. This simplifies the system integration and expedites platform deployment but imposes several inefficiencies (§2.2). First, an EBOF volume is statically mapped to an NVMe drive and embodies location oblivious block placement, failing to harness the massive internal I/O bandwidth. Second, an EBOF volume employs size-dependent bandwidth allocation, whose performance only scales with the volume capacity, unadapted to what the actual workload I/O needs. Third, an EBOF is tenant-unconscious, unaware of the device condition, and incurs substantial interference among concurrent I/O streams.

The root cause is that its static I/O processing pipelines only expose block read/write access interfaces, but lack resource allocation, I/O scheduling, and traffic orchestration capabilities. To address this, our idea is to develop a distributed telemetry system (dubbed **shadow view**) that continuously monitors the EBOF running condition. The shadow view serves as the intermediate layer between an EBOF and storage clients,

[1]Flint is available at https://github.com/netlab-wisconsin/Flint.

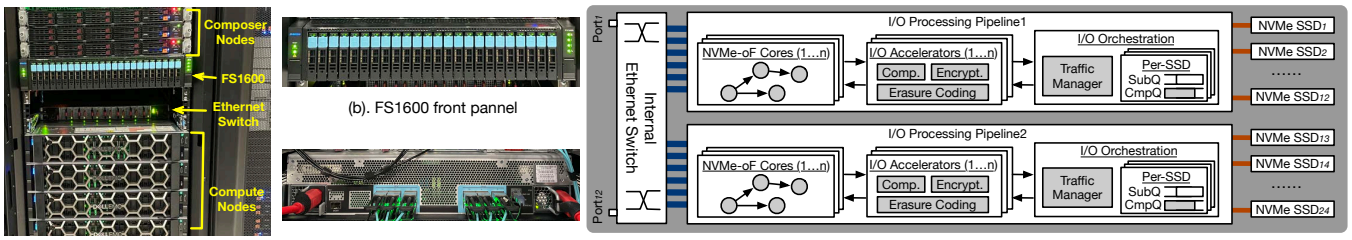


Figure 1: The system deployment and internal architecture of the Fungible FS1600 EBOF platform. (a) depicts our evaluation testbed. (b)/(c) show the front/back panels of the chassis. (d) details the internal architecture of an FS1600.

which exposes the EBOF's internal execution characteristics and assists clients in making efficient I/O execution (such as block placement, bandwidth allocation, and scheduling). The key observation behind this is that the rising data center network speeds make fast (single-digit microsecond) inter-server communication and data synchronization possible. Such a marginal latency to develop the shadow view will then have a negligible impact on the storage applications.

Realizing this vision is non-trivial due to the inherent opaqueness and non-programmable interface of EBOF internals. Towards this, we model an EBOF as a *two-layer multi-switch architecture* that bridges frontend Ethernet ports and backend NVMe drives. Based on the model, we develop a view construction scheme by enabling performance monitoring at multiple vantage points along the I/O path, tracing I/O execution from storage clients, and collectively analyzing the performance characteristics of different components. Our shadow view service runs at view agents (co-located with clients) and a view controller, continuously synchronized via a managed protocol to ensure an up-to-date view is shared among clients. The view controller employs a back-propagation bottleneck analysis mechanism to detect internal EBOF congestions and affected storage streams.

We then build **Flint**–an elastic block storage for an EBOF–atop the shadow view to demonstrate its effectiveness. With the enhanced EBOF I/O data plane, Flint develops three techniques leveraging the shadow view's capabilities. The first one is an elastic volume manager that places storage blocks across all internal NVMe drives on-demand, guided by a runtime EBOF execution profile. The second one is an eIO scheduler that dynamically computes per-IO priority as rank and uses it to orchestrate concurrent I/O submission. The third one is a view-enabled bandwidth auction mechanism, which partitions available bandwidth among competing volumes in a max-min fairness manner at fine granularity. In tandem, Flint achieves high throughput with tolerable latency cost, maximizes device utilization, and ensures efficient multi-tenancy.

We evaluate Flint on a real-system testbed comprising commodity Dell servers and Fungible FS1600 EBOFs. Using FIO-based synthetic micro-benchmarks, we show that one elastic volume over Flint achieves up to 9.3/9.2 GB/s read/write bandwidth, outperforming the defacto EBOF volume by 14.5/13.6×, whose I/O latencies are similar. We further configure various experiments and demonstrate that Flint is tenant-aware and remote target-aware, which can mitigate I/O interference, deliver fair bandwidth allocation, and offer workload adaptiveness. When deploying an object store over Flint, one can achieve up to 2.9× throughput improvements with 66.4%/74.6% average read/write latency savings.

2 Understanding EBOF

This section provides some necessary background about EBOFs and presents a characterization study.

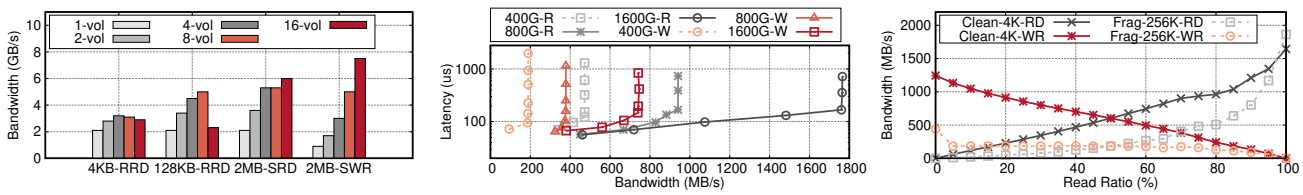
2.1 Ethernet Bunch of Flash (EBOF)

EBOFs are an emerging disaggregated storage solution that holds a number of NVMe drives using an Ethernet switch-integrated SoC (System-on-a-Chip). Compared with today's server/SmartNIC-based JBOFs (Just-A-Bunch-Of-Flash), an EBOF (a) increases the I/O scalability since it elides system-level bottlenecks due to the constrained CPU processing speeds and DRAM/PCIe bandwidths; (b) improves the overall resource utilization and mitigates the resource stranding issue as it eschews the fixed ratio between CPU and SSDs induced by the number of PCIe lanes; (c) achieves high energy/cost-efficiency because it eradicates the hefty computational resources at the storage target. There are some early engineering prototypes being developed, tested, and sampled, such as Ingrasys ES2000 [7] and Fungible FS1600 [5].

FS1600 EBOF. An EBOF is a standalone 2U chassis resembling commodity servers in physical appearance (Figure 1-a). Take the Fungible FS1600 EBOF as an example (which is our development and evaluation target). It encloses 12× 100GbE ports (Figure 1-c) and 24 NVMe drives (Figure 1-b), capable of processing 1.2Tbps storage traffic, sustaining 15M IOPS small random reads and 4.4M IOPS large sequential writes at most, aggregated across all the drives. The typical active power consumption is 750W (850W peak), translating to 200K and 5.9K IOPS per Joule for reads and writes, respectively, significantly outperforming existing JBOF boxes.

Figure 1-d presents the FS1600's internal architecture, consisting of an integrated Ethernet switch, two I/O processing pipelines, and 24 NVMe SSDs. First, the internal switch has 12 ports for external connectivity and delivers network traffic to two pipelines based on the storage targets. Next, packets undergo a sequence of processing stages along the pipeline:

- *Stage 1: NVMe-oF protocol processing.* This includes (i)



(a) Static mapping and location oblivious. **(b) Size-dependent and workload-agnostic.** **(c) Tenant-unconscious and device unaware.**

Figure 2: Performance characteristics of EBOF volumes on FS1600. (a) compares the throughput of four I/O workloads under five logical volume configurations. Physical volumes are from different SSDs. A logical volume uses the striped type (with the 128KB stripe size) where the striping width equals the number of physical volumes. We max out the bandwidth using multiple I/O streams with adequate queue depth. RRD=Random Read. SRD=Sequential read. SWR=Sequential Write. (b) presents the latency-throughput of 4KB random reads and 4KB sequential writes for 400GB, 800GB, and 1600GB volumes, respectively. The y-axis is log scale. (c) shows read/write throughput varying with the read ratio of a mixed workload. The 4KB I/O runs over a clean drive, while the 256KB one runs on a fragmented SSD (preconditioned with 128KB random writes).

networking stack handling and (ii) NVMe-oF capsules and NVMe I/O commands conversion, which are executed over a chain of specialized NVMe-oF cores;

- *Stage 2: Per-IO acceleration.* An EBOF is equipped with several domain-specific accelerators for compute-intensive tasks, such as compression, encryption, and erasure coding;
- *Stage 3: I/O orchestration.* It delivers I/O commands to each SSD via a traffic control module. Akin to an Ethernet switch, there is a traffic manager and a queueing subsystem for both submission and completion.

Last, an EBOF usually supports NVMe drives in U.2 and the emerging compacted EDSFF form factors [3].

EBOF Deployment. An EBOF imposes two differences compared with deploying a traditional JBOF. First, it requires several redundant dedicated servers (called composer nodes in Figure 1-a) to hold the management plane, which (a) configures the EBOF system (e.g., adding/removing forwarding rules for the internal switch, formatting the hot-plugged NVMe drives); and (b) provides a remote control panel for users. Clients interact with these composers on creating/deleting/modifying volumes, monitoring the EBOF running status, and alerting the health condition. Second, each pipeline of an EBOF organizes its Ethernet ports (6 in our case) as one trunk port, with a L3 IP address assigned. The ToR switch, connecting to the EBOF, should then create LACP (Link Aggregation Control Protocol) [6] trunk groups for the corresponding ports, and configure the fallback mode.

An EBOF system provides the physical volume abstraction as today's storage disaggregation solutions. To use it, one should first create a fixed-sized physical volume, which is mapped to one specific SSD. Clients then instantiate an NVMe-oF session, mount the remote volume locally, and make it an NVMe block device. Users can further create logical volumes for capacity/bandwidth expansion and fault tolerance. On our FS1600 testbed, a physical volume, consisting of a sequence of extents, can also be compressed and encrypted if enabled during the volume initialization phase. It primarily supports the NVMe-over-TCP protocol [8].

2.2 Characterizing an EBOF

An EBOF system applies the *smart-sender dumb-receiver* design philosophy and provides backward-compatible volume-oriented storage functionalities. It exposes physical volumes with basic block I/O read/write capabilities, and delegates advanced storage support (such as load balancing and replication) to clients. This drastically expedites the system integration and platform deployment because one can seamlessly transition to an EBOF setup from traditional server-based storage appliances. However, such simplicity comes at a cost. Below we empirically characterize an EBOF and identify several limitations that hinder its performance and efficacy.

Issue #1: Location oblivious block placement. An EBOF volume is statically mapped to an NVMe drive and cannot harness the massive internal I/O bandwidth. Its data placement strategy is straightforward: (i) a newly created volume is placed at one SSD with enough available space; (2) when there are multiple candidates, it chooses one randomly. This simplifies the hardware logic and reduces the in-memory metadata footprint, but bounds the maximum performance of a volume to just one drive, far below the bandwidth capacity of an EBOF. As shown in Figure 2-a, the bandwidth of random reads and sequential writes under large blocks of a physical volume is capped at 2.1GB/s and 0.9GB/s, respectively, approaching the NVMe SSD device limit.

A workaround solution is to create multiple physical volumes (from different SSDs), combine them as a logical volume, and format it into the striped type. This could benefit large sequential I/Os. For example, the 2MB sequential write (Figure 2-a) achieves nearly twice the bandwidth when doubling the number of physical volumes because each I/O hits the strip group, yielding multiple concurrent I/Os to different SSDs. However, for many other I/O profiles, one would observe marginal benefits because the inherent extent layout of a logical volume is location oblivious, unable to harness the parallel I/O paths. As depicted in Figure 2-a, the 4KB/128KB random read maxes out with 4/8 physical volumes, achieving 3.2GB/s and 5.0GB/s. Similarly, the 2MB sequential read shows inferior performance and only sustains 6.0GB/s, less

than one-fifth of the throughput in the 16-vol case.

Takeaways. Flexible and location-aware data placement is essential to unleash the EBOF bandwidth capacity. Ideally, a volume can distribute its extents to one or several bandwidth-capable drives based on online profiling.

Issue #2: Size-dependent bandwidth allocation. The performance of an EBOF volume only scales its capacity and is inadaptable to the actual workload requirements. An EBOF takes the *IOPS/GB* as the key metric when reserving per-volume bandwidth on the control plane upon initialization. The data plane then ensures that the actual bandwidth usage doesn't exceed its allocated share. Such a design only requires a static traffic policer installed at the processing pipeline without employing some dynamic mechanisms like the water-filling algorithm [66]. This completely moves the I/O resource allocation and scheduling out of the performance path, entailing a streamlined I/O pipeline. Take the FS1600 as an example. It configures the QoS upper bound of a volume via $\frac{EBOF\ Read/Write\ IOPS}{EBOF\ Capacity} \times Volume_{size}$.

Figure 2-b depicts the latency-throughput of two workloads running under different-sized volumes. The latency of 4KB random reads (sequential writes) rises drastically when the bandwidth approaches 472 (188) MB/s, 941 (378) MB/s, and 1760 (741) MB/s for 400GB, 800GB, and 1600GB volumes, respectively. This matches the bandwidth upper bound prescribed as above. Such a size-dependent EBOF volume, benefits workloads embodying the same requirements on capacity and throughput, but conversely, contradicts many other application usage patterns. For example, a throughput-oriented workload with a small size requirement, like a metadata service in a distributed blob store [38, 57, 59], has to request a large volume to ensure enough bandwidth is provisioned, entailing storage capacity waste. Similarly, a capacity-oriented application with low throughput needs, like a logging service, allocates a high-capacity volume with excessive throughput reserved implicitly, causing bandwidth waste.

Takeaways. Decoupling capacity allocation and bandwidth reservation functionalities of an EBOF helps improve its efficiency when holding I/O workloads with incongruent size and throughput requirements. This necessitates the I/O processing pipeline to control bandwidth on-demand at the data plane.

Issue #3: Interference heavy. An EBOF volume is tenant-unconscious and device condition unaware, entailing great interference among concurrent I/O streams. An EBOF system enforces performance isolation at the volume granularity. There is a per-volume rate limiter inside the traffic manager that ensures a busy or noisy volume cannot exceed its bandwidth share. This mechanism mitigates inter-volume I/O interference when requests transmit within the pipeline, but completely ignores interference at the NVMe drive. As an example, we co-locate one victim volume with four adversarial volumes on one SSD. The victim volume runs a 4KB random read with QD (queue depth) =1, and we measure its latency. The rest volumes run other throughput-intensive read/write streams. We observe that the P999 latency of the victim volume increases from 169us to several milliseconds!

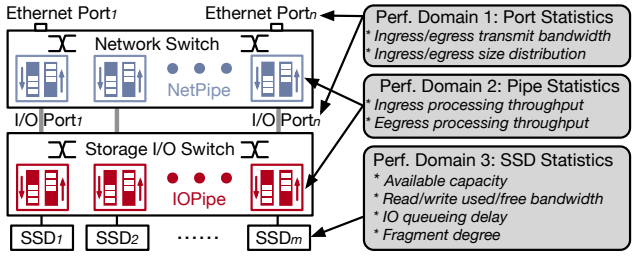


Figure 3: The hardware model of an EBOF and the performance monitor domains of the shadow view.

Further, intra-volume performance isolation is not supported on the EBOF. As shown in Figure 2-c, a 4KB read-only stream achieves 439.0 MB/s higher bandwidth than the mixed case with a 50/50 read/write ratio on a clean SSD. When the drive becomes fragmented, even with a large 256KB block size, it sustains 59.5% and 63.4% less read and write throughput on average across all scenarios. The fundamental issue is that an EBOF schedules I/Os without considering the request cost and SSD condition, yielding inefficient multi-tenancy. As shown in prior studies [11, 37, 50, 54, 74], the per-I/O execution cost is determined by read/write mix ratio, I/O block size, and I/O concurrency. When contention happens at the SSD controller, the device queue, or NAND channels/chips/dies, an I/O would be stalled and incur a high execution cost. Capturing the SSD condition is non-trivial because (a) the number of clean flash pages and their location distribution hinge on the previous write history, which is not exposed; (b) the SSD internal parallelism is unknown due to the FTL mapping and private proprietary internal architecture; (c) housekeeping operations (like garbage collection and wear leveling) are unpredictable and thereby consume a non-deterministic amount of internal bandwidth when triggered.

Takeaways. Mitigating I/O interference requires an EBOF to monitor its end-to-end bandwidth availability (including network, I/O pipeline, and SSD) at the runtime and prudently schedule individual IO in a tenant-aware manner.

3 Shadow View: A EBOF Telemetry System

This section introduces the shadow view, a distributed telemetry system that enhances the EBOF data-plane capabilities.

3.1 Key Idea

The root cause that yields EBOF inefficiencies is the static I/O processing pipelines that only provide block read/write access functionalities, but lack resource allocation, I/O scheduling, and traffic orchestration capabilities. Our key observation is that the rising data center network speeds make fast (single-digit microsecond) inter-server communication and data synchronization possible. Such a marginal latency overhead will have a negligible impact on the storage applications

because the per-I/O latency is in the order of tens to hundreds of microseconds. Therefore, our idea is to develop a software-based telemetry system (called **shadow view**) that continuously monitors the EBOF running condition. It serves as an intermediate layer between an EBOF and storage clients, which exposes EBOF runtime internal statistics and assists clients in efficient I/O execution (including block placement, bandwidth allocation, and scheduling).

3.2 Definition and Capabilities

The shadow view is a software-managed distributed telemetry system that profiles and reports the computation and I/O resource availability of an EBOF at runtime. It decouples from the EBOF hardware box and works as a standalone distributed service, deployed over a bunch of general server hosts. The key challenge to realizing it is the black box nature of EBOF internals that exposes little interaction and programmable interfaces. Hence, we develop a reactive approach that tracks, collects, and analyzes the end-to-end per-I/O statistics from all mounted volumes to construct an EBOF running snapshot.

To facilitate the view development, we model an EBOF as a two-layer multi-switch architecture (Figure 3) that bridges frontend Ethernet ports and backend NVMe drives.

- The **upper half** is a network switch, connecting N Ethernet ports on one side and an equal amount of I/O backplane ports on the other side, which forwards incoming/outgoing Ethernet packets. The switch internally comprises N bidirectional *NetPipe*s. For each pipe, we associate a virtual packet queue with each direction;
- The **bottom half** is a storage I/O switch that connects N I/O ports and M NVMe drives. Except for the basic forwarding functionality, it converts between NVMe-oF capsules and NVMe commands and accesses the SSD drive. Similarly, we model the switch as M bidirectional *IOPipe*s, staying close to the SSD, where each is equipped with a virtual I/O queue on the submission and completion paths.

Note that (a) unlike traditional Ethernet switches, both switches are asymmetric with two distinct sets of operating ports; (b) our hardware model is generic, only requiring minimal knowledge (like port #) of the EBOF specification.

Based on the proposed hardware model, we then organize the shadow view into three performance monitoring domains. The first one is port statistics, where we capture the traffic usage of Ethernet/IO backplane ports at packet/flow/session granularities, including used/available bandwidth and size distribution. The second one is pipe statistics that focus on reporting the processing throughput and queueing delay of *NetPipe*s and *IOPipe*s. We separate ingress and egress metrics to capture I/O submission and completion paths. The last one is SSD statistics. Due to the inherent opaqueness, except for the SSD's available space, we estimate the available read/write bandwidth headroom, I/O delay, and NAND fragment degree (an indicator to determine the I/O write cost) following prior studies [11, 37, 50, 54].

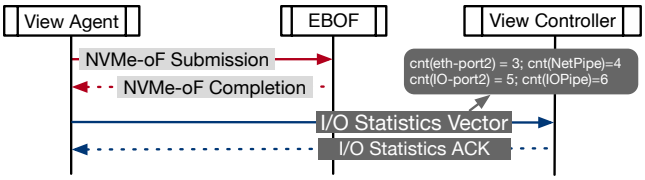


Figure 4: The view construction procedure.

Akin to other telemetry systems, our shadow view behaves as a database for an EBOF. It imposes two main capabilities. One is the runtime statistics query in which one can flexibly integrate the shadow view into a system and explore EBOF-conscious optimizations. The other is event notification, where applications can register callbacks to trigger proactive resource management. We will show how to use the shadow view to develop a block storage (§4).

3.3 View Construction

We construct the shadow view by tracing I/O execution from storage clients, collectively analyzing the performance characteristics of different components, and mapping them to the underlying hardware model. Our shadow view service is distributed across two entities: view controller and view agent. The view controller is integrated into the central arbiter and maintains the entire picture of an EBOF. It receives the per-I/O running statistics from view agents, performs collaborative analyses, and exposes RPC interfaces for interaction. The view agent, colocated with storage clients, monitors the I/O execution of its mounted block devices and only holds a partial view of an EBOF. We employ a hybrid pull/push approach to synchronize the view between these two (§3.4).

An EBOF embodies two types of execution paths, described as follows: (a) NVMe-oF submission: $\textit{Ethernet Port} \Rightarrow \textit{NetPipe}_{ingress} \Rightarrow \textit{I/O Port} \Rightarrow \textit{IOPipe}_{ingress} \Rightarrow \textit{SSD}$; (b) NVMe-oF completion: $\textit{SSD} \Rightarrow \textit{IOPipe}_{egress} \Rightarrow \textit{I/O Port} \Rightarrow \textit{NetPipe}_{egress} \Rightarrow \textit{Ethernet Port}$. Based on this, we show how the view is developed across the following two entities:

- **View Agent.** It takes the timestamp when submitting the I/O request and receiving the response. By subtracting these two, we know the end-to-end round-trip time (RTT), i.e., the sum of NVMe-oF submission and NVMe-oF completion. Upon completion (Figure 4), the agent (a) forwards the I/O statistic vector–<NVMe-oF session, target SSD, I/O type, I/O size, RTT>–to the controller; (b) updates corresponding metrics for its partial view. The I/O statistics vectors are batched before sending to the controller to reduce network communications. This indicates that the view development process is out of the system's performance path;
- **View Controller.** We maintain an in-memory representation of the hardware model (Figure 3) at the controller. It covers all system components from three performance domains. We employ a window-based approach to profile each metric to report the current and EWMA (exponentially

weighted moving average) value. Upon receiving an I/O vector message, the controller first dissects its traversed path within an EBOF based on the NVMe-oF session and target SSD, including *NetPipe*, *IOPipe*, and *I/O Port*. It then updates the corresponding metrics of the full view.

Bookkeeping Statistics. Our agent and controller use the same methods for aggregating running statistics. The packet and I/O distribution are captured via a typical counting array. Regarding bandwidth and throughput, we consider both bit-per-second and packet (I/O)-per-second because the processing capacity of an EBOF is size-dependent. Except for the available capacity, SSD statistics are generally challenging to obtain due to the opaqueness. Following prior work [37, 54, 70], we combine an offline profiling and an online estimation strategy that uses the end-to-end I/O delay as an indicator to decide the queueing delay at the controller and bandwidth headroom. The fragment degree is approximated using the dynamic write cost [54]. Due to the window-based profiling scheme, the view service supports aging automatically. Future commodity EBOFs could do fine-grained timestamping at different stages across the pipeline, which would give us more internal execution visibility. Our shadow view leaves the extension interface to integrate them later.

3.4 View Synchronization

The shadow view is collaboratively built by the view controller and view agents. We develop a view synchronization protocol to update and synchronize the view. Our design is inspired by prior distributed cache-coherence protocols [40, 42]. The key idea is to use a monotonically increasing counter to represent the view recency. Thus, each entity of the hardware model (like port, pipe, and SSD) is assigned a dedicated counter at both agent and controller. We don't use a shared counter because the view can be updated partially, i.e., an I/O request only traverses a subset of an EBOF.

The view controller works as a serialization point of an EBOF to order all issued I/Os. Upon receiving the statistic vector of a completed I/O request, it performs three operations atomically: updates the corresponding view entries, increments their counters by one, and returns an acknowledgment message that attaches the changed counters for the agent's partial view (Figure 4). The controller also maintains a directory of mounting storage clients who hold the partial view. Next, when the view agent receives the response, it first compares the attached counter (X') with the previously held one (X) to decide if the view is obsolete as follows:

- If |X'-X| = 1, this indicates the client is the latest EBOF user and its partial view is update-to-date;
- If |X'-X| > 1, it means there are multiple concurrent storage clients issue I/Os and its partial view is outdated;

The view agent always overwrites the counter with the newly received value because only the controller can update the counter. A careful reader might notice that the message order of the I/O statistics vector received by the controller might be different from the I/O completion order of an EBOF. For example, a client might complete an I/O request earlier than another one, while its statistics update message comes late. We resort to the EBOF timestamp information of the statistics message to resolve the issue. Our controller connects to EBOF composer nodes (§ 2.1) via the management network and has time synchronized. When updating each component of the shadow view, it uses the timestamp to decide if the statistics stay within the profiling window.

Algorithm 1 Back-propagation Bottleneck Analysis.

```
1:  procedure ANALYZE_BOTTLENECK                ▷ Executed per-epoch
2:      for SSD in all_SSDs do
3:          if SSD.avail_bw ≤ Threshold then
4:              Congested_SSDs.add(SSD);
5:              Congested_IOPipes.add(SSD.IOPipe);
6:      for Session in all_NVMeoF_Sessions do
7:          if Session.target_SSD in Congested_SSDs then
8:              Congested_NetPipes.add(Session.NetPipe);
9:              Victim_Sessions.add(Session);
10:     for Session in all_NVMeoF_Sessions do
11:         if Session.NetPipe in Congested_NetPipes then
12:             if Session not in Victim_Sessions then
13:                 Victim_Sessions.add(Session);
```

Based on the above support, our shadow view service provides two types of synchronization. In the *PUSH* mode, the view controller periodically publishes its latest shadow view to its view agents. If a client registers a particular callback function to monitor an event (such as if the SSD available bandwidth has dropped to a threshold), the controller then copies the view proactively. In the *PULL* model, the view agent eagerly fetches its accessed partial view if its own is obsolete. This is realized via the controller RPC.

3.5 Bottleneck Analysis

When an internal component of an EBOF reaches its processing limit, one would observe queue buildup along the EBOF I/O processing pipeline, entailing head-of-line blocking and jeopardizing I/O latency. The shadow view offers two possible solutions to identify the bottleneck: one is examining the queueing delay at the *NetPipe*, *IOPipe*, and *SSD*; another one is comparing the achieved bandwidth with the device limit (obtained from the EBOF specification). Based on our operating and development experience, we find that (a) *SSD*s are always the first to hit the system bottleneck; (b) the processing capability of the *NetPipe* and *IOPipe* is generally over-provisioned. Because of the path symmetry between I/O submission and completion within an EBOF, congestion only happens at the ingress path, not egress.

The shadow view employs a back-propagation analysis scheme to dissect the congestion region and identify victim NVMe-oF sessions (ALG1). When an NVMe drive becomes over-subscribed, the shadow view can quickly locate the affected *IOPipe* because there is a 1-to-1 mapping between

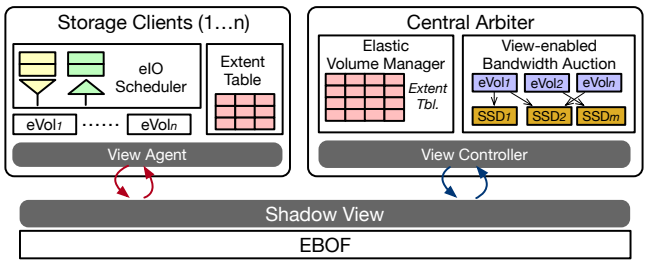


Figure 5: The system architecture of Flint.

IOPipe and *SSD* (ALG1 L1-5). Next, since the mapping between *NetPipe* and *IOPipe* is n-to-m, to identify victim *NetPipe*s, we scan through active NVMe-oF sessions whose destination is the congested SSD (ALG1 L6-9). Last, for each victim *NetPipe*, the algorithm reexamines all other NVMe-oF sessions and finds out the ones that share the congested pipe (ALG1 L10-13). The bottleneck analysis process is launched periodically at the end of each profiling epoch.

4 Flint: an Elastic Block Storage

This section presents Flint that overcomes the drawbacks (§2.2) of existing EBOF volumes. Our design goals are:

- **High throughput with tolerable latency cost.** Massive I/O bandwidth is the crucial capability of an EBOF compared with conventional JBOFs. Flint aims to fully utilize the storage bandwidth across all NVMe drives and eliminate any potential I/O stalls along the pipeline;
- **High utilization and deployment flexibility.** The volume provided by Flint should use any available space from any SSDs, regardless of whether the storage blocks are continuous on one SSD or fragmented across multiple drives. Its performance should scale with the application demands instead of being determined by volume sizes;
- **Efficient multi-tenancy.** Flint should provide the max-min fairness guarantee when partitioning bandwidth among co-located volumes. It should estimate the device bandwidth availability at runtime and admit the right amount of I/Os to avoid oversubscription. The head-of-line blocking issue should be tackled in an end-to-end fashion.

4.1 Overview

Flint comprises two types of nodes (Figure 5): storage client and central arbiter, running atop the shadow view. The client, co-locating with a view agent, allocates/deletes/manipulates volumes, submits read/write I/Os, and orchestrates requests through **an eIO scheduler** (§4.3 and §4.5) to mitigate the head-of-line blocking. The central arbiter, cooperating with a view controller, acts as the EBOF resource manager. It encloses two key system components: (1) **an elastic volume manager** (§4.2), which places data blocks, maintains an extent mapping table, and serves volume management requests; (2) **a view-enabled bandwidth auction mechanism** (§4.6), partitioning available bandwidth among competing volumes in a max-min fairness manner at the fine granularity. A volume

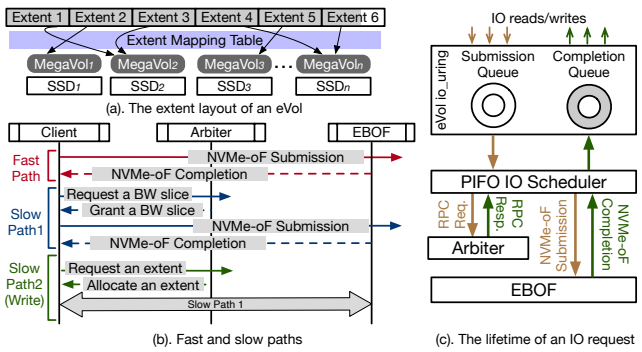


Figure 6: *eVol* extent layout, I/O lifetime and I/O path.

can be replicated using chain replication for high availability (§4.4). The arbiter is not replicated because it is not in the data path and clients can continue performing I/Os on an open volume even if the arbiter is down, although it can be made reliable. All the cross-node communications are realized via RPCs, and we implement them atop eRPC [35].

4.2 Elastic Volume and Volume Manager

Flint provides an elastic volume (*eVol*) to address the limitations of the defacto EBOF volume. An *eVol* consists of a sequence of fixed-sized extents (2MB in our case), spreading across more than one SSD. To enable flexible data placement over all EBOF drives, Flint employs a two-level hierarchical volume manager at the arbiter. As depicted in Figure 6-a, the bottom layer creates one mega-volume (*MegaVol*) for each SSD covering the entire storage space through the default EBOF control plane. The upper layer then performs block allocation and places extents to different mega-volumes based on the given policy. Each *eVol* has an extent mapping table, where our volume manager uses an aggregated one for the whole EBOF. Each mapping entry has 16 bytes with the following format: SSD index, physical extent number, and replication node. We implement the extent table using an in-memory hashtable. It takes only 8MB to address a 1TB *eVol*, which is feasible on commodity servers.

A storage client requests a new volume via the `create_vol` API. It takes the volume size and allocation policies as parameters. We apply a lazy allocation scheme. Upon the `create_vol` request, the volume manager only checks whether there is enough capacity to hold the volume, reserves the space, and sends back the acknowledgment. A successful return doesn't indicate the actual data placement is done. When an I/O write happens and there is no corresponding mapping entry in the local extent table, clients would interact with our volume manager to allocate a new extent from one SSD (mega-volume), and then update extent tables on both sides. The extent tables are stored in RocksDB [1] backed by a replicated file system (such as Ceph [77]), and all updates are executed in a transactional manner.

We use a weighted score function to realize different data placement policies. Under the extent allocation request from an *eVol*, our volume manager calculates the intended score

for each SSD and chooses the one with the lowest score. The function (shown below) balances a slew of factors: (a) prior allocation history, i.e., how many extents have been placed; (b) available capacity, where we aim to balance the SSD utilization; (c) recent busy status from the bandwidth perspective; (d) fragment (or wear-leveling) degree of the SSD; (e) user-defined preferences. We obtain the parameters of (c) and (d) from the shadow view.

$$Score(eVol_i, SSD_j) = \alpha \times Extent_{\#} + \beta \times Capacity_{used} + \gamma \times Busy_{degree} + \delta \times Fragment_{degree} + \eta \times Preference$$

The volume manager maintains a score matrix for each *eVol* and updates it periodically. Clients set the initial placement policy in `create_vol` and can alter the strategy on the fly via `update_vol`. When a volume is destroyed (`destroy_vol`), the volume manager frees the space, updates the metadata from the extent table, and deletes the related score matrix.

4.3 I/O Path

In Flint, a storage client submits I/O requests first to a co-located block I/O scheduler, then to a per-thread `io_uring` engine. We choose the `io_uring` interface [2, 10, 22] due to its efficient asynchronous support, rich feature set, and high scalability. The I/O scheduler (§4.5) leverages the shadow view to determine the I/O issuing order, mitigating the head-of-line blocking. Finally, Flint uses the NVMe-over-TCP protocol [8] to carry I/O reads/writes between clients and an EBOF. Figure 6-c depicts the lifetime of an I/O.

Flint introduces fast and slow paths when accessing the EBOF because its bandwidth resources are managed by the central arbiter (Figure 6-b). A read can be issued to the EBOF directly only if the allocated bandwidth slice (§4.6) can hold the request size (fast path). Otherwise, it should request a new slice from the arbiter and then submit the I/O (slow path1). A write behaves similarly except that it may introduce one more slow path–asking the arbiter to allocate a new extent. However, in practice, we noticed that this extra RPC call takes only a few hundred microseconds and incurs negligible performance overhead, given that: 1) a typical write I/O to consume an extent (2MB) takes thousands of microseconds; 2) this slow path can only happen for write I/Os; 3) write I/Os are directed to the most performance-capable extent that is dynamically allocated based on the current system condition.

4.4 Replication

Flint employs the chain replication [65] as an optional configuration to improve system availability. It replicates data blocks at the extent granularity. When enabled, given an extent allocation, the volume manager chooses three different SSDs with the lowest scores for placement. Then in the extent mapping table, each entry records all the replicas along with the ordering in the replication chain.

Flint follows standard protocol execution logics as chain replication prescribes. I/O writes are issued to the head, which are then passed along the chain. We pass down commands instead of actual updates because the target LBAs (logical block addresses) are different. I/O reads are handled by the tail of the chain. Chain replication achieves strong consistency naturally because requests are ordered by the tail, which operates as the primary in a traditional primary-backup system. There are two ways to facilitate request passing over the chain. One is using the EBOF's recirculation capability, where a completed I/O can be reinserted into the I/O processing pipeline with some modifications. Such an intra-EBOF coordination is an ideal solution, reducing the replication latency and saving external network bandwidth. The other one is implementing the chain replication coordinator at the client host. Essentially, after receiving the I/O completion from the predecessor, it refabricates the I/O submission request and sends it to the successor. This incurs multiple round trips between the client and EBOF. Flint takes the latter approach because our FS1600 box doesn't expose the recirculation directly. We are currently working with the vendor on realizing the former approach.

Algorithm 2 eIO Scheduler.

```
1:  procedure EIO_ENQUEUE
2:      io = pending_queue.poll();
3:      compute_rank(io);
4:      sort_window.binary_insert(io);
5:  procedure EIO_DEQUEUE
6:      for io IN sort_window do
7:          if io.id == io.stream.sq.front().id then
8:              io.admit();
9:              sort_window.remove(i);
10:             io.stream.sq.pop();
11:             aging(); break;
```

4.5 eIO PIFO Scheduler

Flint runs an I/O scheduler at each client to improve the *eVol* performance. To capture the dynamic I/O execution cost and mitigate the resulting head-of-line blocking, Flint develops a priority-based I/O scheduler based on the ranking concept. The rank of an I/O determines its issuing order.

Our eIO PIFO scheduler uses a scheduling tree for the implementation. It encompasses a priority queue for submission and a FIFO queue for completion. Upon an incoming I/O, the scheduler computes its rank and inserts it into the corresponding location. Rank, representing the I/O cost, takes I/O characteristics (size and type), queueing time, and allocated bandwidth into account. A lower-ranked I/O would be dequeued first. The completion side delivers I/O simply following the arrival order. When the client receives a bottleneck report (§3.5), all victim I/Os will experience a rank increase due to the rising cost, thereby hedged from submission.

Algorirthm 2 presents the pseudo-code of our eIO scheduler. It polls I/Os issued from different threads, calculates their ranks based on the current shadow view, and inserts them into a binary search tree that is sorted using the rank value. The *sort_window* uses binary search since requests

Algorithm 3 Client-Side Bandwidth Auction.

```
1:  procedure IO_SUBMISSION(io)
2:      ssd_nqn = io.ssd_nqn;
3:      avail_slices = shadow_view[ssd_nqn].avail_slices;
4:      slice = avail_slices.back();
5:      if slice.is_full() then
6:          slot = RPC_request_slice(avail_slice);
7:      slice.consume_io(io);
8:      io.admit();
9:  procedure IO_COMPLETION
10:     slice = io.slice
11:     slice.completes += 1;
12:     if slice.is_full() AND slice.submits = slice.completes then
13:         slice.close();
14:     RPC_report_completion();
```

Algorithm 4 Arbiter-Side Bandwidth Auction.

```
1:  procedure ARBITER_BW_AUCT
2:      while true do
3:          epoch_advance(); estimate_avail_bw();
4:          reassign_quota();
5:          while not epoch_expired() do
6:              refresh_drr_lists();
7:              for tenant in active_list do
8:                  tenant.deficit += Quantum;
9:                  if tenant.deficit > DEF_THRESHOLD then
10:                     tenant.state = Idle; tenant.deficit = 0;
11:                     idle_list.add(tenant);
12:                     active_list.remove(tenant);
13:                     reassign_quota();
```

are sorted by their ranks. During dequeue, Flint iterates over the *sort_window* until an I/O is in the front of the submission queue of an I/O stream and admits it. Hence, Flint ensures: (1) no I/O reordering happens within a stream, therefore following their submission order; (2) requests belonging to different streams are submitted following their ranks. Finally, we apply aging for the pending requests to prevent starvation.

4.6 Bandwidth Auction

Flint allocates bandwidth fairly among competing storage streams using an RTS/CTS-like mechanism [17]. An NVMe-oF client can only issue an I/O when granted enough bandwidth slices. Hence, each active client periodically asks the arbiter for available bandwidth, and the arbiter sends back grant messages after the auction. The challenge to realizing it is that the bandwidth allocation of an EBOF should holistically consider each component between Ethernet ports and SSDs, instead of only targeting one entity. For example, a storage stream might receive adequate bandwidth from an SSD but then be bottlenecked by the I/O processing pipeline (like *NetPipe*), causing bandwidth waste.

Our arbiter combines deficit round-robin (DRR) [71] and gang scheduling [23,24] techniques to address the issue. Each active NVMe-oF session–that has issued a bandwidth allocation request–is assigned a three-tuple deficit counter vector, representing resource requirements for *NetPipe*, *IOPipe*, and *SSD*. We then divide all sessions into different groups based on their target SSD. The shadow view reports the bandwidth headroom of each component, which we translate to the total amount of I/Os that can be admitted. For each group, the arbiter walks through its associated sessions whose deficit counter vector is accumulated with a prescribed quantum (whose value is in proportion to the session's priority) at each round. A bandwidth slice is granted when (a) all the deficit counters of a session have reached their target thresholds, and (b) there are adequate resources indicated by the shadow view. After that, the deficit counter vector is reset and the session waits for the next allocation. Users can specify their performance targets and Flint will then calculate the quantum to ensure the total bandwidth of a SSD is partitioned accordingly. By default the bandwidth is equally shared.

Algorithms 3 and 4 detail the bandwidth auction mechanism on the client and arbiter sides. When a client issues an I/O, it looks up the current bandwidth slice from the local shadow view and checks its availability. If the slice has no room, it asks for a new one from the arbiter. The slice consumes an I/O based on its weighted cost (profiled by the shadow view) and then calls *admit*() to notify the I/O executors for submission (through the `io_uring`). Upon completion, the client reports the I/O latency to the arbiter. If the slice becomes empty and there are no outstanding I/Os, it requests for a new bandwidth slice, which is piggybacked through the report RPC (ALG 3 L12–14).

On the arbiter's side, there is a DRR routine running for each SSD device. Each DRR routine maintains three different lists: (a) an active list containing tenants that are actively issuing I/Os, (b) a deferred list containing tenants that have run out of the virtual slice quota, and (c) an idle list that includes tenants that do not issue I/Os within a time window. At the beginning of a new epoch, Flint estimates the available bandwidth of the device and reassigns the bandwidth slices to each tenant. Flint then runs the DRR loop until the current epoch expires. In each loop, we refresh the lists in case their states have changed and iterate tenants over the *active_list* to increase their deficit value. If the deficit exceeds a threshold, we set its state to *Idle*, update the corresponding lists, and reassign quotas (ALG 4 L10–L13).

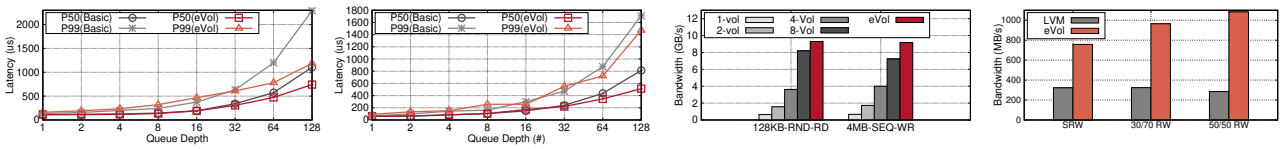
5 Evaluation

Our evaluations aim to answer the following questions:

- How does Flint perform compared with the default EBOF block storage in terms of latency and throughput? (§5.2)
- How does Flint address the three performance inefficiencies of an EBOF volume? (§5.3, §5.4, §5.5)
- What are the considerations when deploying Flint? (§5.6)

5.1 Experimental Methodology

Testbed. We evaluated Flint using a small-scale rack cluster, comprising 2U commodity servers as storage clients, a Dell



(a) 4KB Random Read. **(b) 4KB Sequential Write.** **(c) Bandwidth.** **(d) Congested Bandwidth.**

Figure 7: Performance comparison between eVol and the defacto physical/logical volume on EBOF and LVM. (a)/(b) report the P50 and P99 latency of 4KB random read and 4KB sequential write when increasing the queue depth. (c) shows the maximum bandwidth of 128KB random read and 4MB sequential write comparing eVol and different types of logical volumes. (d) shows the bandwidth of LVM and eVol of 1MB sequential write, 30/70 mixed read/write, and 50/50 mixed read/write, where certain drives are congested.

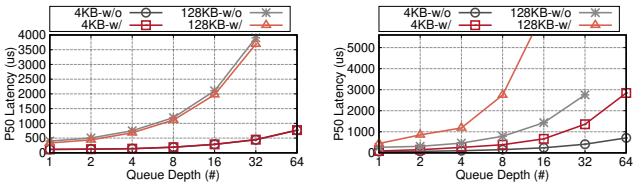
Z9264F-ON ToR switch (w/ 64× 100GbE ports), and the Fungible FS1600 EBOF. Each server is a Dell R7525 box, enclosing two AMD 7302 processors (running at 3.0GHz), 256GB DDR4 memory, a Broadcom 25GbE dual-port NIC, a Mellanox 100GbE dual-port CX6 NIC, and two Samsung NVMe SSDs. All clients run Ubuntu 22.04.

Experiment setup. We build Flint from scratch in about 7600 lines of C++ code. Flint provides eVol management and both sync/async I/O interfaces to the client. We primarily compared with the defacto volume service provided by EBOF. Our experiments first use FIO-based synthetic workloads [4] to explore the benefits of Flint proposed techniques and the capabilities of the shadow view. We then take an object store as a storage application to showcase the overall benefits. We use the YCSB benchmark [20] in this case. We report throughput and average/tail per-I/O latencies as the performance metrics.

5.2 eVol Performance

We first examine the basic performance of an eVol. In terms of small I/Os, when we increase the same queue depth for an I/O stream, eVol achieves the same P50 latency as a physical volume (Figures 7-a and b). This is because the majority of I/Os traverse the fast path (§4.3), yielding one RTT between the client and an EBOF. An eVol places data blocks over all the SSDs in an EBOF. As a result, given the same queue depth, each SSD will receive less amount of I/Os than the physical volume. Thus, the eVol reduces the P99 latency by 48.1% and 13.4% in terms of 4KB random read and 4KB sequential write, respectively. Regarding large I/Os, eVol can harness the massive I/O bandwidth of an EBOF easily because of its flexible data layout (Issue #1 in §2.2). As shown in Figure 7-c, it achieves 9.3/9.2 GB/s when running 128KB random read/4KB sequential write, outperforming a logical volume with 1, 2, 4, and 8 physical volumes by 14.5/13.6×, 5.9/5.3×, 2.6/2.3×, 1.2/1.3×, respectively.

Replication. Flint uses chain replication (§4.4) to achieve high availability. Figure 8 reports the P50 latency of I/O reads and writes with the replication enabled/disabled. We use three replicas in this experiment. For 4KB and 128KB random reads, with replication, they achieve the same performance as the disabled case when varying the queue depth because requests are served by the tail node. However, in terms of writes, since an I/O should pass through the chain from the head to the tail, one would experience a considerable latency increase. For example, on average across all cases, the P50 latency of a 4KB and 128KB write is worsened by 2.9× and 3.5×, respectively. With future EBOF recirculation support (where a request can be reinjected into the I/O processing pipeline), we expect the latency can be further reduced.



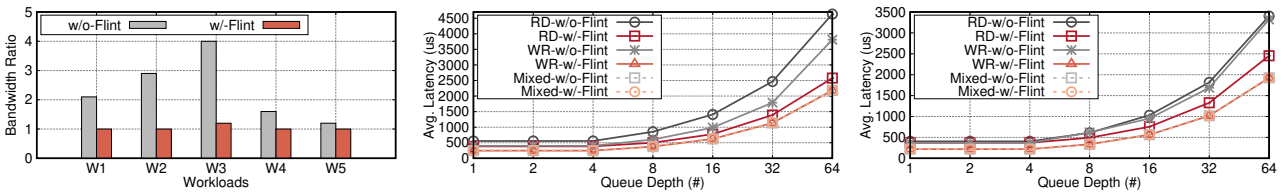
(a) I/O read under replication **(b) I/O write under replication**

Figure 8: Latency varying the I/O queue depth when enabling and disabling the replication.

5.3 Flint Decouples Capacity and Bandwidth Allocation

Next, we evaluate how Flint addresses the second issue (§2.2) via the bandwidth auction technique. We configure five scenarios with two types of contending random I/O streams, where each is issued from different-sized volumes: W1 (16× 4KB reads *v.s.* 8× 128KB reads), W2 (8× 4KB reads *v.s.* 8× 4KB writes), W3 (8× 4KB writes *v.s.* 8× 128KB reads), W3 (8× 4KB reads *v.s.* 8× 4KB writes), W4 (8× 128KB reads *v.s.* 8× 128KB writes), and W5 (8× 4KB 70/30 reads/writes *v.s.* 8× 4KB 30/70 reads/writes). We max out the bandwidth of each stream using a large queue depth (QD).

Figure 9-a reports their aggregated relative bandwidth ratio. For all scenarios, Flint ensures I/O streams with the same demand receive similar bandwidth regardless of their capacity. This is because (a) our arbiter distributes equal bandwidth slices to each active tenant in a deficit round-robin manner; (b) a client can only submit an I/O when its granted slice still has room. Thus, a small volume can still receive more bandwidth if it has more pending I/Os, whereas a large volume cannot take more than its (weighted) equal share. Under contention, no matter how many outstanding I/Os a stream issued, it can only receive more bandwidth when all contending streams operate in a similar phase. When disabling our mechanism, one can observe great unfair allocation. For example, in W3, the 128KB read ones take 4× more than the 4KB write ones.



(a) Bandwidth Allocation. (b) Perf. under a read-congested SSD. (c) Perf. under a write-congested SSD.

Figure 9: We compare two cases that enable (w/) and disable (w/o) the proposed Flint technique to evaluate the multi-tenancy support. (a) presents the achieved bandwidth ratio between two categories of contending I/O streams. (b) and (c) show the average latency of a 32KB read, write, mixed (70% read) I/O stream when running under a read/write-congested SSD.

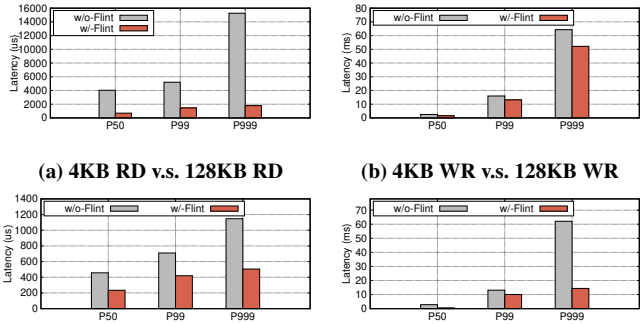
5.4 Flint Mitigates I/O Interference

This experiment demonstrates the effectiveness of the Flint I/O scheduler. We set up two categories of I/O streams on one SSD: the victim ones that issue 4KB read/write I/Os with a small QD; and the background ones that send different types of I/Os with a large QD. Because Flint computes the I/O rank (priority) on the fly and dynamically determines the request issuing order, a low-ranked I/O would then be inserted in the front of the queue, which is dequeued first, mitigating the head-of-line blocking issue. Figure 10 reports the average P50/P99/P999 latency of the victim streams. When 4KB random read co-locating with 128KB random read (Figure 10), Flint helps reduce the P50, P99, and P99 latencies by 4.8×, 2.6×, and 7.5×, respectively. We observe similar trends in the other three cases. Under read/write mixed cases, the latency savings are related to the write cost estimation, which determines the accuracy of computing the write I/O rank. For example, when 4KB writes interleave with 128KB writes, one can achieve 36.4%, 17.7%, and 18.9% lower P50, P99, and P999 latency when enabling Flint, respectively.

5.5 Flint Avoids Busy EBOF SSDs

Flint is conscious of the device running condition due to the shadow view's capabilities. To demonstrate this, we set up an experiment that pollutes an EBOF SSD using 4 128KB random reads/writes. We then create an eVol over the SSD and launch a 32KB read, write, and mixed (70% read) stream. We disable/enable the dynamic extent remapping technique and I/O scheduler and measure the average latency when increasing the QD. As shown in Figures 9-b and c, Flint consistently reduces latency for scenarios because it eschews the congested SSD and directs I/Os to others. For example, under the read/write congestion, it reduces the average latency by 40.1%/29.8%. Therefore, based on §5.4 and §5.5, one can see that Flint is capable of tackling the third issue (§2.2) and achieves effective multi-tenancy.

We also compare the performance of eVol and Logical Volume Management (LVM). Figure 7-d shows their bandwidth under three different workloads, where some SSDs are congested. Compared with the workload oblivious placement in LVM, eVol is aware of the device's runtime condition and places extents in a less-loaded drive, yielding 2.3×, 3.0×, 3.8× throughput improvement, respectively.



(a) 4KB RD v.s. 128KB RD (b) 4KB WR v.s. 128KB WR

(c) 4KB RD v.s. 4KB WR (d) 4KB RD v.s. 128 WR

Figure 10: Latency comparison when consolidating different types of I/O streams. RD/WR=Read/Write.

Primitive	Throughput (MRPS)	P50 (us)	P99 (us)
`view_query`	21.5	24	31
`view_sync`	5.8	38	67

Table 1: The shadow view performance on a storage client.

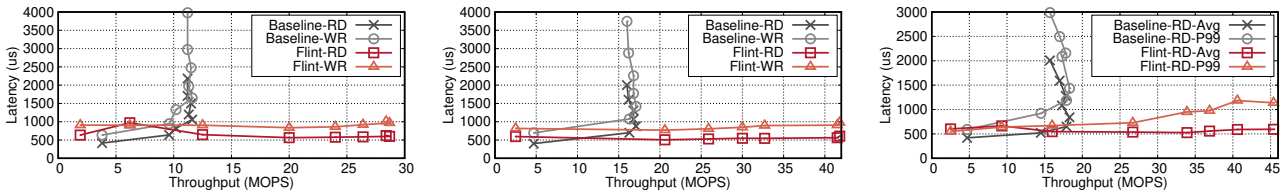
5.6 System deployment

Application integration. Flint is backward-compatible, provides the volume interface, and supports different storage workloads. As a case study, we deploy an object store [12,29] over Flint and see how much benefits it can bring to the application. We use one storage client in this experiment. Figure 11 presents the latency v.s. throughput when running YCSB workloads. When running over an eVol, one can achieve 2.8×, 2.8×, 2.9× throughput improvements, with 66.4(74.6)%, 63.7(73.8%), and 61.9% read(write) latency reduction for YCSB-A, YCSB-B, and YCSB-C, respectively.

Shadow view. `view_query` and `view_sync` are the two major RPC primitives provided by our shadow view service. Flint uses the Protobuf [9] for serialization, incurring some data processing overheads on both the client and arbiter sides. For example, as shown in Table 1, the average/tail latency of a `view_query` is 24/31us, which could be optimized using hardware-assisted serialization techniques [63].

6 Related Work

Block Storage. Researchers have built a plethora of distributed block storage systems in different settings. For example, Petal [41] realizes an early vision of network-attached disks through ATM networks that enable adaptive capac-



(a) YCSB-A. (b) YCSB-B. (c) YCSB-C.

Figure 11: Performance comparison between the EBOF volume (Baseline) and Flint when holding an object store. We use the YCSB benchmark [20] with the default parameters. We report the average latency for A/B and average/P99 latency for C.

ity/bandwidth scaling. Parallax [51] harnesses the characteristics of a virtualized environment to implement advanced storage features and provides a block interface over storage VMs. Strata [21] develops a scalable storage system over network-attached flash using global data address translation and SDN-assisted storage protocol virtualization. Blizzard [53], built atop FDS [59], provides a cloud-scale virtual drive with some optimizations, such as nested striping and out-of-order write commits. Decibel [58] exposes the *dVols* abstraction and employs a shared-nothing data-plane design, allowing efficient sharing of disaggregated storage class memories (SCMs).

Object Storage. It organizes data as objects and uses a global identifier for addressing. NASD (network-attached secure disks) [27] builds an early version of object store and enables scalable disk bandwidth. OceanStore [39] develops a data object infrastructure that spans the globe and provides continuous access to persistent information. Ceph [77] distributes data replication, failure detection, and recovery to semi-autonomous object storage devices that run a specialized local object file system. Haystack [16] is an object store for the Facebook's photos application and changes costly disk metadata operations into in-memory ones. FDS [59] proposes locality-oblivious blobs over full bisection CLOS networks. F4 [57] designs a hierarchical tier architecture and improves the warm object access performance. Pocket [38] is a distributed object store for serverless applications with fast response, auto-scaling, and smart data placement capabilities.

Distributed File Systems. There is a long history of building distributed file systems. xFS [13] proposes a serverless principle and distributes management and data access services over multiple machines. Frangipani [75] takes a similar design but relies on the underlying Petal [41] to improve system scalability. GFS [26] is a fault-tolerant distributed file system targeting large-scale data processing workloads. pNFS [32] extends NFSv4 with a layout driver, an I/O driver, and a file layout retrieval interface such that clients can perform direct access to remote storage. HDFS [73] serves as the storage backend for the Hadoop data processing framework. NOVA [78] optimizes the file system performance on hybrid memory systems (DRAM+NVM) using a log-structured design. Assise [14] materializes the client-local persistent memory module as a linearizable and crash-recoverable cache and improves the performance of distributed file systems.

Programmable Networks. An EBOF is partially motivated by reconfigurable switches [18, 31, 44, 48, 68, 69] and SmartNICs [19, 25, 28, 46, 49, 60–62, 67], which realize some functionalities of its data path (§3.2). Akin to an in-network telemetry system, our shadow view captures EBOF runtime statistics and serves as an assistant building block for others.

Storage Disaggregation. People have explored disaggregated storage extensively given the rising networking bandwidth and fast remote storage protocol [29, 33, 34, 36, 37, 43, 45, 47, 54–56, 64, 79]. Ana Klimovic *et al.* characterize the performance of iSCSI-based disaggregated storage [36]. Reflex [37] develops a customized kernel-bypass data-plane for remote NVMe accesses. i10 [34] develops an efficient in-kernel TCP/IP remote storage based on dedicated end-to-end IO paths and delayed doorbell notifications. LeapIO [43] unifies the address space across X86 CPUs and ARM coprocessors, exposing a virtual NVMe to an unmodified guest VM. Researchers [30] also report the performance characteristics of server-based NVMe-oF boxes. Gimbal [54] designs an efficient multi-tenancy mechanism to share a SmartNIC JBOF via traditional networking techniques. LEED [29] builds a fast and replicated key-value store over an array of SmartNIC-based storage appliances to improve energy efficiency. RackBlox [64] co-designs the software-defined networking and storage stacks, dividing storage functionalities between the programmable switch and the host I/O stack.

7 Conclusion

This paper explores how to use emerging EBOFs efficiently to serve storage applications. The linchpin is a distributed telemetry system (shadow view) that continuously monitors and reports the EBOF running condition. It enhances the static and opaque EBOF I/O processing pipelines with performance monitoring, resource accounting, and central and global coordination capabilities. Using it, we then design and implement an elastic block storage system over EBOFs and demonstrate considerable benefits compared with defacto schemes from performance, utilization, and multi-tenancy perspectives.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Kang Chen, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, and CAREER-2339755.

References

[1] Rocksdb. https://rocksdb.org/.

[2] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2025.

[3] Enterprise and Datacenter Standard Form Factor. https://www.snia.org/forums/cmsi/knowledge/formfactors#EDSFF, 2025.

[4] Flexible I/O Tester. https://github.com/axboe/fio, 2025.

[5] Fungible FS1600 Platform. https://www.servethehome.com/fungible-dpu-based-storage-cluster-hands-on-look/fungible-fs1600-capacity-and-performance/, 2025.

[6] IEEE 802.3ad Link Aggregation Configuration. https://www.ibm.com/docs/en/aix/7.1?topic=aggregation-ieee-8023ad-link-configuration, 2025.

[7] Ingrasys ES2000/ES2100 Platform. https://www.ingrasys.com/es2000, 2025.

[8] NVMe Express TCP Transport Specification. https://nvmexpress.org/specification/tcp-transport-specification/, 2025.

[9] Protocol Buffers. https://protobuf.dev, 2025.

[10] The rapid growth of io_uring. https://lwn.net/Articles/810414/, 2025.

[11] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for {SSD} performance. In *2008 USENIX Annual Technical Conference (ATC'08)*, 2008.

[12] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*, pages 1–14, 2009.

[13] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP'95)*, page 109–126, 1995.

[14] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-Local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[15] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 49–67, 2023.

[16] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, 2010.

[17] Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang. MACAW: A Media Access Protocol for Wireless LAN's. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM'94)*, page 212–225, 1994.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*, page 99–110, 2013.

[19] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, pages 143–154, 2010.

[21] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory. In *12th USENIX Conference on File and Storage Technologies (FAST'14)*, 2014.

[22] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage (Systor'22)*, pages 120–127, 2022.

[23] Dror G Feitelson. Packing schemes for gang scheduling. In *workshop on job scheduling strategies for parallel processing*, pages 89–110, 1996.

[24] Dror G Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing*, 16(4):306–318, 1992.

[25] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 51–66, 2018.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, 2003.

[27] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, page 92–103, 1998.

[28] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, page 916–929, 2023.

[29] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 1012–1027, 2023.

[30] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference (Systor'17)*, 2017.

[31] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, page 33–47, 2023.

[32] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pNFS. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 18–27, 2005.

[33] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 297–312, 2024.

[34] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

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

[36] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, 2016.

[37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 345–359, 2017.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 427–444, 2018.

[39] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna

Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, page 190–201, 2000.

[40] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, page 302–313, 1994.

[41] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, page 84–92, 1996.

[42] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, page 92–103, 1992.

[43] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, page 591–605, 2020.

[44] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[45] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[46] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, page 318–333, 2019.

[47] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 305–323, 2020.

[48] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 795–809, 2017.

[49] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 363–378, 2019.

[50] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. vFair: latency-aware fair storage scheduling via per-IO cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15)*, pages 125–138, 2015.

[51] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual Disks for Virtual Machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys'08)*, page 41–54, 2008.

[52] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*, page 753–766, 2022.

[53] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 257–273, 2014.

[54] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, page 106–122, 2021.

[55] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[56] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage*, 20(3), June 2024.

[57] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 383–398, 2014.

[58] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 17–33, 2017.

[59] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 1–15, 2012.

[60] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 663–679, 2018.

[61] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, page 16–22, 2020.

[62] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 772–787, 2021.

[63] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, page 200–215, 2023.

[64] Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-Mei Hwu, Deming Chen, Sameh Asaad, and Jian Huang. RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, page 182–199, 2023.

[65] Robbert Van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI'04)*, 2004.

[66] Jordi Ros and Wei Kang Tsai. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2, pages 717–726, 2001.

[67] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 740–755, 2021.

[68] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 1–16, 2018.

[69] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 685–699, 2020.

[70] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for Flash-Based SSDs. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 67–78, 2013.

[71] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'95)*, page 231–242, 1995.

[72] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 783–799, 2024.

[73] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[74] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. {MQSim}: A framework for enabling realistic studies

of modern {Multi-Queue}{SSD} devices. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 49–66, 2018.

[75] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*, page 224–237, 1997.

[76] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 449–462, 2020.

[77] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[78] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 323–338, 2016.

[79] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

Pushing the Limits of In-Network Caching for Key-Value Stores

Gyuyeong Kim
Sungshin Women's University

Abstract

We present OrbitCache, a new in-network caching architecture that can cache variable-length items to balance a wide range of key-value workloads. Unlike existing works, OrbitCache does not cache hot items in the switch memory. Instead, we make hot items revisit the switch data plane continuously by exploiting packet recirculation. Our approach keeps cached key-value pairs in the switch data plane while freeing them from item size limitations caused by hardware constraints. We implement an OrbitCache prototype on an Intel Tofino switch. Our experimental results show that OrbitCache can balance highly skewed workloads and is robust to various system conditions.

1 Introduction

Key-value stores are the fundamental building blocks for online services owing to fast access to variable-length data [2, 5, 16, 30]. Workloads are generally read-intensive, and most items are hundreds of bytes [7, 8, 12]. A challenge in scaling key-value stores is mitigating load imbalance among storage servers caused by skewed key popularity (e.g., trending events). The load imbalance leads to overloading hot item servers, causing overall performance degradation [21,27,29].

In-network caching is a promising load-balancing approach that builds a cache in the switch by leveraging programmable switch ASICs like Intel Tofino [3]. The idea is to store hot items in the on-chip switch memory. Requests read the item value by referring to the cache lookup table where the item key is the table index. Compared to server-based caching, in-network caching provides much higher performance without extra cache nodes or accelerators thanks to a Tbps-scale processing throughput.

Unfortunately, existing solutions [21, 32, 34] can cache only tiny items due to hardware constraints. Specifically, the key size is limited to 16 bytes due to the maximum match-key width of the match-action table. A limited number of match-action stages and a small accessible byte size per stage make it difficult for the value size to exceed 128 bytes. These numbers are far from typical workloads where key and value sizes are still small enough to be stored in a single packet, but larger than the limit [7,8,12,33,37]. For example, in 54 Twitter workloads [37], most keys are tens of bytes, and many values are less than 1024 bytes. However, the existing solutions cannot cache even a single item in 42 out of 54 workloads because the keys and values are generally larger than the limits. Between the other 12 workloads, only 2 workloads have a portion of cacheable items larger than 50%. This trend is similar in Facebook workloads as well [12]. In this context, we ask the following question: *how can we cache variable-length items in the programmable switch for balancing typical key-value workloads?*

We present OrbitCache, a new in-network caching architecture that can cache variable-length key-value items in the switch data plane. Our idea is to make hot items visit the switch data plane continuously in the form of cache packets instead of caching the hot items in the switch memory. We efficiently use packet recirculation, a built-in feature of the programmable switch ASIC. This enables a packet to revisit the switch data plane through an internal loopback port without going outside. The switch can cache variable-length key-value pairs without being limited by the hardware constraints owing to circulating cache packets. For cache serving, the switch maintains small request metadata in the switch memory and forwards the cache packet to the client if there is a pending request for the key. To support variable-length keys, we use key hashes for cache lookups and resolve hash collisions at the client by comparing the requested key and the returned key, which is contained in the cache packet.

The key insight behind the idea is that it is hard to overcome the memory access constraint of switch hardware if we stick to the approach of caching items in the switch memory. For example, the idea of using key hashes for variable-length keys is difficult to realize in the existing solutions. This is because the switch does not have enough hardware resources to store keys and values together. In OrbitCache, cache packets contain both keys and values, allowing to handle hash colli-

sions at the client. Meanwhile, we may recirculate requests multiple times to read larger values. However, this limits scalability because the number of in-flight packets in the recirculation port increases proportionately to the number of requests, making a bottleneck. In addition, the switch has one internal recirculation port. OrbitCache avoids a bottleneck in the recirculation port. This is because 1) requests are never recirculated; 2) the number of in-flight cache packets is small and constant; 3) the switch can process multiple cache packets simultaneously with hardware-level parallelism, thus minimizing queueing between cache packets.

Realizing the idea imposes various technical challenges. First, the switch should buffer multiple request metadata for cached items. We design a request table as a circular queue structure using multiple register arrays. Our request table provides isolated data access among different keys. Second, a cache packet should serve multiple requests once fetched, rather than a single request. To achieve this, the switch clones the cache packet with low overhead before forwarding it to the client using the packet replication engine (PRE), a special module of the switch ASIC. Next, we should ensure cache coherence. We design an invalidation-based coherence protocol that can fetch new values and send back a reply to the client simultaneously. Lastly, the switch should deal with dynamic workloads where key popularity changes over time. We design an efficient cache update mechanism in the switch control plane.

We have implemented a OrbitCache prototype on an Intel Tofino switch. To evaluate OrbitCache, we build a testbed consisting of commodity servers. Our experimental results show that OrbitCache provides load balancing for many skewed workloads having diverse item sizes. In addition, OrbitCache is robust to various workload conditions like key access distributions, write ratios, and the number of servers. We also show that OrbitCache can adapt to dynamic workloads where key popularity changes over time.

In summary, this work makes the following contributions.

- We design OrbitCache, an in-network caching architecture where the switch can cache variable-length hot items to balance a wide range of workloads for distributed key-value stores.
- We propose several techniques to address technical challenges when realizing the idea of variable-length in-network caching.
- We implement a OrbitCache prototype on Intel Tofino switches and show the efficiency and robustness of OrbitCache through extensive testbed experiments.

2 Background and Motivation

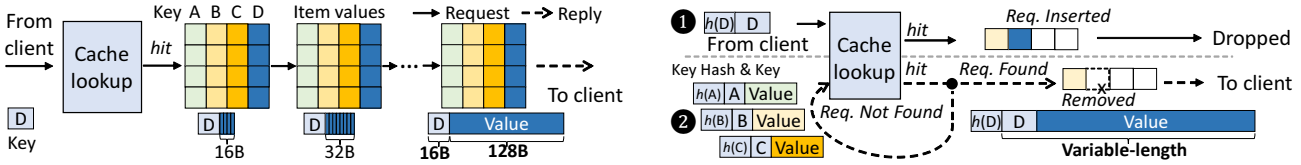
2.1 In-Network Load-Balancing Caches

Balancing key-value stores with a small cache. In distributed key-value stores, balancing imbalanced loads across storage servers caused by different key popularity is a primary challenge. Caching is a powerful technique to address the challenge, which is based on a theoretical result called the *small cache effect*: we can balance loads for N servers (or partitions) by caching the $O(N \log N)$ hottest items, regardless of the number of items [15]. Caching on commodity servers is a natural option to build the cache [15, 16, 29]. However, the performance of a cache node is not sufficient to handle many requests due to the limited throughput of CPUs. Building a high-performance caching layer using multiple replicated cache nodes is expensive and causes poor write performance for cache coherence.

Why in-network caching? Building a load-balancing cache in the switch by leveraging programmable switch ASICs like Intel Tofino [3] is an attractive approach. Unlike server-based caching, this approach can provide high-performance load balancing in a single box without additional nodes. This is because the switch is highly optimized for packet I/O and can process a few billion packets per second, which is an order of magnitude higher than that of the CPU in servers. The switch also provides a low packet processing delay within hundreds of nanoseconds.

Limited cacheable item size. Existing works [21, 32, 34] demonstrate the efficiency of in-network caching, but they limit the cacheable item size. Specifically, they support items of up to 16-byte keys and 128-byte values. In the reconfigurable match table (RMT) switch architecture [10] like Intel Tofino, the switch data plane consists of n match-action stages [3]. Each stage has a static memory and a few ALUs that can perform simple arithmetic operations on k bytes. The existing works store the value of cached items across multiple stages after fragmentation, limiting the maximum value size to $n \times k$ bytes. Unfortunately, $n \times k$ of the current switch is quite small, and the available number of stages for accessing the value is less than n since other non-caching functions also consume match-action stages. The key size is also limited by the maximum match-key width of the match-action table in a single match-action stage. This is because the existing works use a match-action table to implement the cache lookup table where the item key is the match key.

Why is it not enough? Many key-value items are indeed small, but typically exceed the existing size limits. We have analyzed 54 Twitter workloads [37], and observe that existing solutions are insufficient to handle typical workloads. For example, only 3.7% of the workloads have over 80% of keys $\leq$ 16 B. 38.9% of the workloads have over 80% of values $\leq$ 128 B. The existing works can cache less than 10% of items for 85% of the workloads because, to be cacheable, both key and value sizes must be within the limits. Furthermore, they cannot cache even a single item for 77.8% of the workloads. Facebook workloads [12] show similar size distributions. The average key size is 27.1 bytes, and the median value size is 235 bytes. These numbers exceed the size limits of the existing solutions.



(a) The NetCache architecutre [21]

(b) The proposed OrbitCache architecture

Figure 1: Comparison of the high-level idea with the NetCache architecture. *In NetCache, requests read fragmented values stored in the memory across stages. In OrbitCache, clients submit requests, and then circulating cache packets read request metadata. Since both keys and values are in cache packets, hardware constraints do not limit the item size.*

2.2 Variable-Length In-Network Caching

Design rationale. We argue that it is hard to overcome the memory access constraint if we stick to the approach of caching data in the switch memory because the constraint is determined at the time of manufacture. Therefore, we propose OrbitCache, a different approach to in-network caching.

Our high-level idea is to keep cached key-value pairs circulating within the switch data plane through packet recirculation, a built-in feature of the switch that makes the packet revisit the switch data plane. In our approach, cached items read requests rather than the requests read the cached items. Specifically, clients submit requests, and the switch maintains small metadata like the client IP address. *Cache packets*, which are reply packets containing the key and the value of cached items, continue to revisit the data plane via a recirculation port while actively checking pending requests. Note that cache packets do not go outside the switch. If a request for the key is found, the matched cache packet is forwarded to the client. Otherwise, the cache packet is recirculated. In Figure 1, we illustrate the difference between OrbitCache and the NetCache architecture [21], which is the reference architecture of existing in-network caching solutions [21, 32, 34].

Variable-length keys. Our approach enables us to support variable-length key-value pairs. For variable-length keys, one possible solution within the limited match-key width of the match-action table is to use a fixed-size key hash as the match key. To handle potential hash collisions, we should compare the requested key and the key of the value that has been read. To do this, the switch data plane should also store the item keys alongside values. Unfortunately, it is hard to realize in the existing architecture because there are not enough match-action stages to store both keys and values. OrbitCache can realize the idea of using key hashes since the switch maintains cached key-value pairs in the form of circulating reply packets. Therefore, the client can get the correct value from the storage server if the requested and returned keys differ.

Variable-length values. For variable-length values, a possible approach in the existing architecture is recirculating requests. Specifically, requests can access the switch memory multiple times using packet recirculation. However, this approach is not scalable because the number of in-flight packets per second in the recirculation port increases in proportion to the number of requests. For example, if every request is recirculated 7 times to read a 1024-byte value, the effective throughput of the recirculation port is reduced to 1/8 of the bandwidth. Unfortunately, a pipeline in the programmable switch has only one internal recirculation port, while there are tens of regular front ports. This means that the recirculation port limits the performance excessively.

Our recirculation-based caching design avoids the bottleneck in the recirculation port as follows. First, the switch never recirculates requests. Second, only a small, constant number of cache packets are recirculated. The time to recirculate and process a cache packet is a few hundred nanoseconds like normal packets. Therefore, the queueing delay in the recirculation port rarely increases as the request rate grows.

Trade-off. There is a trade-off of using recirculation to achieve variable-length caching. Although we are free from size limitations, we should limit the cache size. This is because, in OrbitCache, requests should wait until cache packets handle them. For a cache packet, the time to read a request is impacted by the other in-flight cache packets. This means that the number of in-flight cache packets in the recirculation port determines the latency for cache serving. Although the switch can process many cache packets simultaneously, the request may wait excessively until being served if there are too many cache packets. The sacrifice of cache size for variable-length caching is backed by the small cache effect. As described in Section 2.1, caching a small number of hot items is enough to balance skewed workloads [15, 27].

Technical challenges. We should address several technical challenges to translate the idea into a working system.

- The switch should maintain multiple requests, especially for the same cached item. If not, many requests for cached items would be forwarded to the server as they cannot be stored in the data plane.
- A cache packet should serve multiple requests once fetched. Otherwise, the switch must fetch the cache packet from the server again, degrading performance.
- We should design a cache coherence mechanism between the switch and storage servers. If not, a request may read a stale value for the requested key.

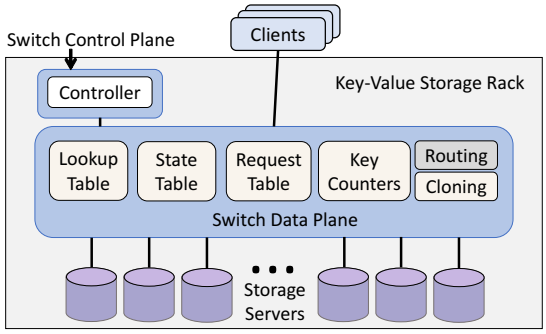


Figure 2: OrbitCache architecture.

- The switch should adapt to key popularity changes.

3 OrbitCache Design

3.1 OrbitCache Architecture

Figure 2 shows the overview of the OrbitCache architecture.

Switch data plane. The switch data plane in OrbitCache consists of several custom tables and modules as follows.

- *Lookup table* is for cache lookups, which is a match-action table that uses a hash of item keys as the match key. The table returns a table index needed to access other tables and modules. Cache entries are managed by the controller in the switch control plane.
- *State table*, implemented as a register array, maintains the validity of the value of cached items. This is needed to prevent read requests from obtaining the stale value when there are pending writes for the requested key.
- *Request table*, implemented with multiple register arrays and registers[1], stores request metadata like client IP address, L4 port number, and request sequence number.
- *Key counters* consist of two registers and one register array. The key popularity counter is a register array that tracks the key popularity for each key. The cache hit counter and the overflow request counter are registers that track the total number of cache hits and the total number of overflow requests for all cached keys. The controller uses these for cache sizing.
- *Cloning module* comprises a few match-action tables for cloning reply packets. Packet cloning is done by the PRE, a hardware module in the switch ASIC [3].

Meanwhile, the switch invokes the custom processing logic only for OrbitCache packets by referring to reserved L4 ports. We use UDP to handle messages for better latency like existing works [21,27,29] while using TCP for top-k item reports in cache updates. For normal packets, the switch only applies the traditional packet forwarding logic.

[1] In P4 language, a register means an indexed register array. However, in this paper, we define a register as a single-slot register and a register array as an indexed register array for better clarification.

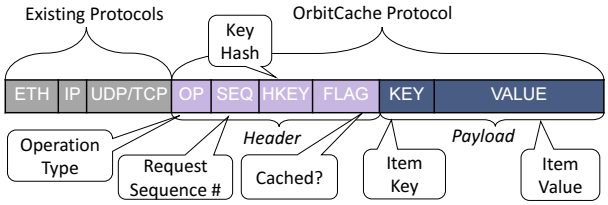


Figure 3: OrbitCache packet format.

Switch control plane. The controller in the switch control plane performs cache updates and switch configurations. It evicts the least popular keys and inserts new hot keys based on server-side periodic top-k reports for uncached keys and the switch-side report for cached keys. Note that while the controller handles key insertion, value fetching is done via the switch data plane. It also handles switch configurations like the rule update for packet forwarding tables.

Clients and servers. Clients should specify the operation type, the item key to request, and the key hash. Storage servers run a server application that acts as a shim layer that translates OrbitCache messages to API calls for key-value stores and vice versa. The server returns replies for read and write requests like regular storage servers. However, in cases of write requests for cached items, the server returns a write reply with the item value as well to fetch the latest value.

3.2 Packet Format

Figure 3 depicts the OrbitCache packet format. The message consists of the header and the payload. The switch only parses the header. The payload of OrbitCache message consists of the key and the value. Our header is 22 bytes. Therefore, OrbitCache supports a key-value pair of up to 1438 bytes for a single packet when we consider a 1500-byte MTU packet where 40 bytes are for L3/L4 headers. For example, the switch can cache an item with a 16-byte key and a 1422-byte value. Our header fields are as follows.

- `OP` (1 byte): the operation type, which can be `R-REQ` (Read request), `W-REQ` (Write request), `R-REP` (Read reply), `W-REP` (Write reply), `F-REQ` (Fetch request), `F-REP` (Fetch reply), and `CRN-REQ` (Correction request).
- `SEQ` (4 bytes): a request ID assigned by the client, which is used for resolving hash collisions.
- `HKEY` (16 bytes): the key hash as cache lookup index.
- `FLAG` (1 byte): a flag field to distinguish write requests for cached items from those for uncached items.

3.3 Basic Request and Reply Processing

In Figure 4, we illustrate the packet processing logic.

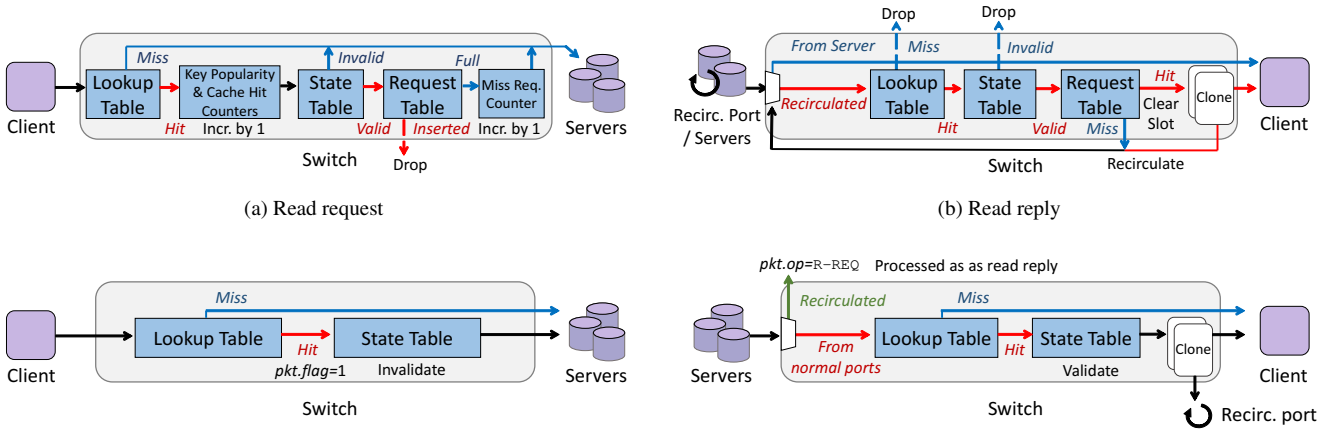


Figure 4: Request processing. *(a) the switch drops the request after inserting request metadata into the request table; (b) If a circulating cache packet reads request metadata, the switch clones the packet so that the original packet is forwarded to the client and the cloned one is recirculated again for further serving; (c) the switch invalidates the item to avoid inconsistent reads if a write request is for a cached item; (d) upon receiving a write reply for a cached item, the switch validates the item. After that, the switch clones the packet. The cloned packet is processed as a read reply after updating the operation type.*

Request generation at clients. Clients should specify the `OP` and `HKEY` fields. The `SEQ` field is increased by one for every request, which is used to resolve hash collisions. The destination storage server is determined by hashing the key.

Read requests. The switch first refers to the lookup table using a key hash to get a table index, which is used to access the slots in other tables. If missed, the switch forwards the packet to the server as the request is for an uncached item. In case of a cache hit, the key popularity counter and the cache hit counter are incremented by one. This key popularity is collected by the controller in the switch control plane periodically for cache updates. Next, the switch checks the validity of the requested key by looking into the state table. The state is binary: valid or invalid. Being invalid means that there are pending write requests for the key. In this case, the switch forwards the read request to the server to avoid reading stale item values. If the key is valid, the switch checks the request table to see whether there is a free slot. The switch puts the request metadata into the table when a free slot is found. Otherwise, the request is destined to the server after the overflow request counter is increased. Request metadata includes the client IP address, L4 port number, and `SEQ` as request IDs. After insertion, the switch drops the packet. This is acceptable since a cache packet will soon service the stored request.

Read replies. When receiving a read reply, the switch first checks to see if the ingress port is the recirculation port. If it is, the reply is a cache packet. Otherwise, it is a reply for an uncached item sent by the server. For cache packets, if the cache misses or the state of the value is invalid, the switch drops the packet. A cache miss for a cache packet means that the controller evicted the key from the lookup table due to a change in key popularity or the cache size. The invalid state indicates a write request with a new value is in progress.

The switch goes through the lookup and state tables, and then looks for pending requests in the request table. If a request is found, the switch forwards the cache packet to the client after updating the header with metadata and removing the metadata from the table slot. To make the cache packet serve more requests, the switch clones the packet before forwarding. The switch forwards the original packet to the client and the cloned one to the recirculation port. The switch recirculates the packet if there are no pending requests.

Write requests. The switch checks whether the requested key is in the cache lookup table. If it is, the value for the key is invalidated to prevent reading the outdated value. In addition, the switch sets the `FLAG` field to 1 to indicate that this request is for a cached item. This makes the storage server append the value in the write reply. Regardless of a cache hit, the switch forwards the request to the storage server to update the value of the key in the server.

Write replies. If the key is cached, the switch validates the value to allow read requests to get the latest value. The switch clones the packet so that the client receives the write reply while the switch has a new cache packet simultaneously. The switch updates the `OP` field to `R-REP` of the cloned packet after the first recirculation since cache packets should be read replies. For a cache miss, the switch forwards the packet to the client since the reply is for an uncached item.

Other types of messages. Fetch messages are for cache updates. Correction messages are for resolving hash collisions. Fetch/correction requests are delivered to the storage servers. The fetch reply is processed as a write reply.

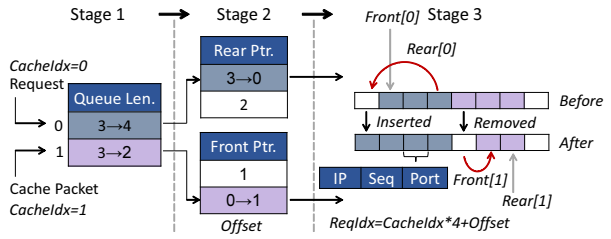


Figure 5: An example of operations in the request table.

3.4 Buffering Request Metadata

Circular queue-based request table. We should store multiple request metadata for cached items because the requests should wait until cache packets serve them. The switch should be able to buffer concurrent read requests for the same cached item. If not, many requests for the cached item would be forwarded to the storage server due to the lack of vacancies. Furthermore, a request for a key should be isolated from requests for other keys since the request may undergo excessive queueing delay due to the different requests.

We design a request table based on a circular queue structure. Our request table provides a logical queue for each cached key, and the queue can be accessed in $O(1)$ by efficient indexing. This provides fast and isolated queue access for a key regardless of other keys. The request table consists of 6 register arrays: the first three arrays are for storing request metadata for each key (i.e., client IP address, request sequence number array, and L4 port number). The other three arrays are for managing queue operations. The queue length array maintains the number of stored request metadata for each key. The front pointer array handles dequeue operations by tracking the first request metadata for each key. The rear pointer arrays perform enqueue operations by keeping track of the last request metadata for each key.

The queue management arrays are indexed using a table index ($CacheIdx$) returned by the cache lookup table. The request metadata arrays are accessed via a request index, which is defined as $ReqIdx = CacheIdx \times S + i$ where S is the maximum queue size per key and i is the offset for ith slot in the logical queue for a key and is given by the pointer arrays. The switch uses three match-action stages for a request table. Specifically, the switch first checks the queue status (Stage 1), and performs en/dequeue operations if the queue is not full/empty (Stage 2). Next, the switch gets/puts metadata from/into the request table (Stage 3).

An example for table operations. Figure 5 shows a simple example of table operations. We assume that the request table can maintain up to 4 requests for each key. We consider enqueueing and dequeueing operations using read requests and cache packets. In stage 2, the rear pointer changes to 0 from 3 since we implement a circular queue. It is also easy to see that the request metadata for different keys does not collide since we partition the metadata arrays using the indexing formula of $ReqIdx = CacheIdx \times S + i$.

3.5 Cache Serving

Handling multiple requests via packet cloning. In OrbitCache, cache packets pass through the switch data plane repeatably to serve pending requests for cached items. One challenge is serving multiple requests with a cache packet fetched only once because the cache packet is forwarded to the client after updating the header using the retrieved metadata. A strawman is to fetch the cache packet from the server again, but this approach is inefficient as the switch cannot serve pending requests for the key until the fetching is completed. Fetching multiple copies of the cache packet may be a solution, but this incurs excessive queueing delay between cache packets. In addition, it is hard to know how many cache packets should be fetched because we cannot predict the number of requests for each cached item in advance.

To address this, we utilize packet cloning, another built-in feature of the programmable switch ASIC besides packet recirculation. Packet cloning is done by the PRE, a hardware module specialized for packet cloning in the switch ASIC. Cloning has low overhead for the following reasons. First, the PRE is located after the ingress pipeline. This means the switch does not repeat the ingress pipeline processing for the cloned packet, not causing extra ingress processing delay. Second, the switch does not copy the entire packet. It only copies the small descriptor pointing to the memory location of the packet and reuses the packet data.

We use multicast to forward the original and clone packets. Through a match-action table using the destination IP address acquired from the request table, the switch gets a multicast group ID that specifies the regular port number directed to the client and the recirculation port number. The switch forwards the original packet to the client and the cloned one to the switch data plane again, making it serve more requests.

3.6 Handling Hash Collisions

The existing works use the item key as the match key of the cache lookup table, which is implemented as a match-action table. Since the match-key width is limited by hardware, we cannot use the key exceeding the size limit. We may use a register array, but the size limitation in the register index is stricter than in the match-action table. Therefore, we use the fixed-sized key hash as the match key. A challenge is how to resolve potential hash collisions. We should handle this because a request may read the wrong value of other keys.

Client-side collision resolution. We handle hash collisions at the client level by maintaining a list of the keys for each request that has not yet received a reply. The list is indexed by $pkt.seq$. Request packets contain both the original key and the key hash since we should use the original key

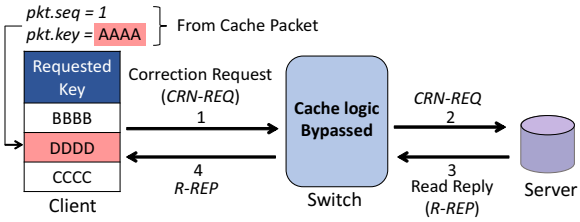


Figure 6: An example of handling hash collisions. *The client gets the correct value by sending a correction request.*

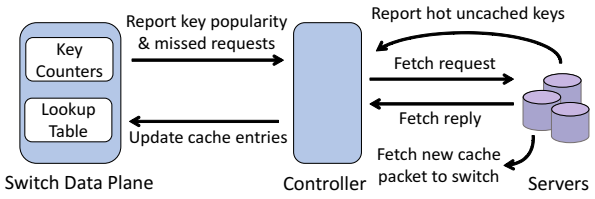


Figure 7: Cache update. *The controller updates the entries of the cache lookup table by tracking workload changes.*

to get the value in servers. Upon receiving read replies, the client checks whether the requested key in the list and the returned key in the reply header are identical. If different, the client sends a correction request to the server. The switch then forwards the request without applying the cache logic so that the client gets the correct item value from the storage server. We use a simple, low-overhead hash function to calculate key hashes. In addition, the complexity of accessing a matched slot of the key list is $O(1)$ as it is an indexed array. Note that a key in the list exists only until the reply arrives.

One issue is that requests for uncached but colliding keys always undergo this process. However, this is rare since our 128-bit key hash provides a low collision probability of $1/2^{128}$. In our experience, we never see a hash collision. There is a corner case when a new hot item uses the table index of the evicted item after cache updates. In this case, the pending requests for the evicted item are served by the new but wrong cache packet. However, they eventually get the correct value through the correction process with 1-RTT latency overhead. Meanwhile, the switch drops the colliding cache packet when hash collisions occur due to write requests. To handle this, the server can send the cache packet as a fetch reply to the switch again if the received write request is with $pkt.flag = 1$ but not in the hot key list.

An example of resolving hash collisions. We plot an example of handling hash collisions in Figure 6 where the requested key is `DDDD` but the returned key is `AAAA`. Upon receiving the reply, the client detects that the retrieved value is wrong by referring to the key list using $pkt.seq = 1$. The client then sends a correction request to the storage server. The switch bypasses the cache logic, and forwards the packet to the server. The storage server returns the value as a read reply, and the client finally gets the correct value for the key `DDDD`. The client removes the key from the list. $pkt.seq$ wraps around if it reaches the maximum value.

3.7 Cache Coherence

To ensure cache coherence between the switch and servers, we design an invalidation-based coherence protocol illustrated in Figure 4 (c) and (d). The switch invalidates the value when handling a write request for a cached item and revalidates the value upon receiving a write reply. Ideally, the outdated cache packets should be replaced with the latest cache packet. However, requests may be returned with the stale value since the outdated packets are still circulating in the switch data plane even if the item value becomes invalid. To address this, the switch drops the cache packet if the item is cached but its value is invalid. Owing to the fact that the cache packet is dropped before accessing the request table, we can prevent read requests from retrieving the stale value until a new cache packet is fetched. The storage server sends a single reply packet, and the switch updates the value and replies to the client simultaneously by cloning the packet.

3.8 Cache Updates

To handle key popularity changes over time, we update cache entries based on periodic popularity reports from the switch and storage servers. We count the popularity of cached keys in the switch data plane using the key popularity counter. The controller keeps track of key popularity by reading the counter periodically. Meanwhile, storage servers periodically report the top-k keys to the controller, which are popular uncached keys. The servers use a count-min sketch with five hash functions to track key popularity in a memory-efficient manner while ensuring accuracy. To reflect the recent status only, we reset all the counters to zero after reporting.

If the cache needs to be updated due to a change in key popularity, the controller deletes the victim key from the cache lookup table and inserts a new popular key instead. The controller then sends a fetch request to the storage server containing the latest value for the key. Upon receiving the request, the server fetches a new cache packet to the switch data plane and replies to the controller. The new popular key inherits the table index (*CacheIdx*) of the evicted key. With this, the pending requests for the evicted key can be handled by the new cache packet and the hash collision resolution mechanism. Note that the hash collision here is due to the same table index, not that the hashes actually collide.

3.9 Handling Practical Requirements

Failure handling. For packet loss, we can use an application-level mechanism. Our controller uses UDP with a timeout-based mechanism to exchange fetch requests/replies. Storage

servers use TCP to report top-k items to the controller. Meanwhile, server failures do not cause a OrbitCache-specific problem. Switch failures result in the loss of cached items, but the cache can be reconstructed quickly by the controller after the switch is recovered because the switch failure is similar to the rapid key popularity changes.

Multi-rack deployment. OrbitCache can be deployed with multiple racks because the ToR switch caches hot items of storage servers belonging to its rack only. For example, assume that we have two racks where ToR switches, $ToR1$ and $ToR2$, are interconnected via a spine switch SPN. When CLI, a client in rack 1, sends a request to SRV, a server in rack 2, the path for an uncached item is $CLI-ToR1-SPN-ToR2-SRV-ToR2-SPN-ToR1-CLI$. If the item is cached, $ToR2$ is the only switch that applies the cache logic. There is no issue even if rack 1 is the replicated rack of rack 2 where the two ToR switches have the same cache entries because the request is handled by $ToR1$. If the client is located in another rack and the other two racks are replicated ones, we can balance requests by implementing load-balancing mechanisms in the ToR switch of the client-side rack.

3.10 Discussion

Multi-packet items. Many key-value items are less than the MTU size. Therefore, existing works in key-value stores generally consider item values up to 1024 bytes [6,17,26,29–31]. However, some items may exceed the MTU size, such as articles and photo objects for some workloads (e.g., Wikipedia and Flickr [8]). To cache multi-packet items, we should fetch multiple cache packets with fragmented values for the same key. To do this, we maintain the number of forwarded cache packets for each item by placing another register array alongside the request table, the *ACKed packet counter*. The initial value of each slot is 1 since most items are single-packet. When fetching item values, the storage server puts the number of packets comprising the item in the `FLAG` field. When handling cache packets, the switch retrieves metadata as usual, but it does not manipulate the slots of all register arrays in the request table if the slot value in the ACKed packet counter is not equal to `FLAG`. Instead, the switch increases the value by one. When the value in the counter equals to `FLAG`, the switch removes metadata as it is ready to be finished.

Write-back caching. FarReach [34] is a recent work that enables write-back caching. Although it still cannot serve many workloads due to the size limitation, it provides high performance regardless of the write ratio. OrbitCache uses write-through caching, and our performance gain decreases as the write ratio grows. The difference between write-through caching and write-back caching is whether a write request for cached items updates the storage server or not. Therefore, OrbitCache can also use write-back caching by letting the switch return write replies upon receiving write requests after updating the cache only, though we need extra modules like snapshot generation.

Multi-pipeline deployment. The multi-terabits performance of the programmable switch ASIC comes from the multi-pipelined architecture. Each pipeline consists of tens of regular ports and one internal recirculation port. Metadata and memory data are not shared between the pipelines. Therefore, the pipelines of client-directed ports and server-directed ports should be the same. Otherwise, cache packets in a pipeline may not read requests since they are in another pipeline. This can be addressed by mapping each pipeline of the ToR switch to a client-directed port on the spine switches. This is especially feasible when the clients and servers are in different racks. We expect this to be addressed more easily with a new programmable switch architecture like MP5 [35] that achieves high performance with a logical single pipeline.

4 Implementation

Client-server application. We develop an open-loop application in C using NVIDIA Messaging Accelerator library (VMA) [4]. VMA bypasses kernel network stacks by intercepting the socket function calls and translating them to native RDMA verbs. The client application measures throughput and latency by generating requests. The time gap between consecutive requests follows an exponential distribution. The server application has multiple threads where each thread is pinned to a disjoint CPU core. To emulate multiple storage servers, we assign a partition per thread so that each thread acts as an independent storage server. We also limit the Rx throughput of each emulated server to 100K RPS to ensure the bottleneck is at servers in our testbed. This technique is used in existing works [21,29,34] as well. In a similar vein, like NetCache [21], we implement a key-value store with TommyDS [1], a high-performance hash table library.

Switch. We implement the switch data plane in $P4_{16}$ [9] for Intel Tofino [3]. We use Intel P4 Studio SDE 9.7.0 to compile the switch data plane. Our prototype uses 9 stages and 6.67% SRAM, 7.38% Match Input Crossbar, 9.29% Hash Bit, and 30.56% ALUs. The request table has a maximum queue size of 8 for each key. The controller is written in Python 3. In our prototype, we use an additional register array for the request table to store the timestamp of the request for latency measurement. The OrbitCache header has 3 extra fields of 1-B `Cached`, 4-B `Latency`, and 1-B `SrvID`. The first two fields are required to separately measure the latency of requests. The last field is to store the server ID as we emulate multiple storage servers as dedicated threads in a physical node.

5 Evaluation

5.1 Methodology

Testbed setup. We build a cluster consisting of 8 nodes, which are connected by an APS BF6064X-T switch with the

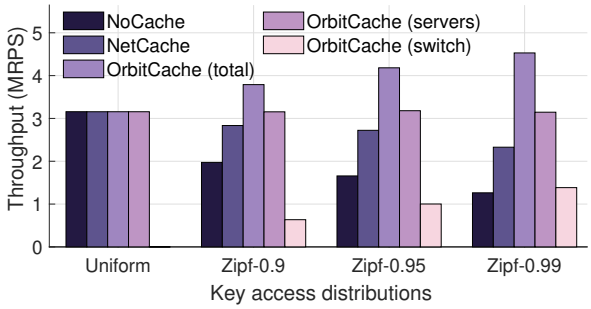


Figure 8: Throughput with different skewness.

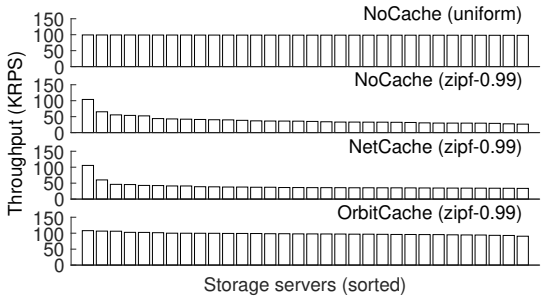


Figure 9: Load on individual storage servers.

Intel Tofino 1 [3]. The servers are equipped with a 10-core CPU (Intel i5-12600K @ 3.7 Ghz, 12 hyperthreads and 4 non-hyperthreads), 32 GB of DDR5 memory, and a 100GbE NVIDIA CX-5 NIC. The servers run Ubuntu 22.04 LTS with Linux kernel 6.5.0. The 4 nodes act as clients and the remaining 4 nodes are used to emulate multiple storage servers.

Compared schemes. We mainly compare our work with NoCache and NetCache [21]. NoCache is a mechanism without cache logic. NetCache is the representative in-network caching architecture. FarReach [34] and DistCache [32] also adopt the NetCache architecture. We implement the core caching logic of NetCache, but our implementation provides items up to 64-byte values across 8 stages with an 8-byte accessible size per stage. We find that the P4 compiler allocates only two cache read tables per stage, even with pragma statements. We suspect that this is due to compiler restrictions on our code. We clarify that this does not mislead the conclusions of our experiments, as there is a negligible difference in the latency required to read 64 bytes and 128 bytes at line rate using the same number of match-action stages.

Workloads. We emulate a single storage rack with 32 storage servers using 8 partitioned threads per node. We basically consider a workload with 10M key-value pairs whose key popularity skewness follows a Zipfian distribution with $\alpha = 0.99$, since it is regarded as typical skewness [7, 13]. Although both key and value sizes impact whether an item is cacheable by NetCache, we use 16-byte keys by default for simplicity, the maximum supported key size by NetCache. Instead, we represent the portion of cacheable items using different ratios between 64 bytes and 1024 bytes values. The 64-byte value represents a cacheable item value of NetCache. The 1024-byte value is a typical-sized item value in many workloads and is considered in many research papers [6, 17, 26, 29–31]. We use a bimodal distribution with 82% 64-byte and 18% 1024-byte values by considering the cacheable item ratio of NetCache for the `Cluster018` workload of Twitter [12]. Most experiments are for read-only workloads as we target read-intensive workloads.

Except for dynamic workloads, we preload the 10K and 128 hottest items for NetCache and OrbitCache, respectively. 128 is a nearly optimal cache size for OrbitCache that provides the best performance gains. Since NetCache can cache only 82% of items in the workload, the actual number of cached items is 8200, which is still larger than the cache size of OrbitCache by 64×. To be fair, we choose 82% of keys among the 10K hottest keys with a uniform distribution. We store the chosen keys as a text file to make experimental results consistent. The controller loads the file and puts the keys to the switch cache. We note that trends are usually consistent, even when using different key samples.

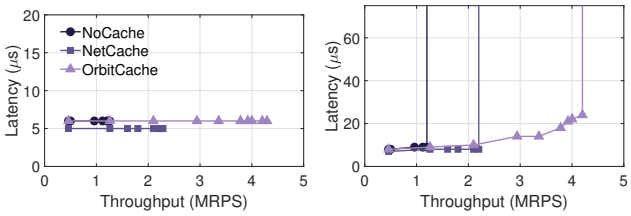
5.2 Main Results

Throughput with different key access distributions. We measure the throughput with different skewness and plot the results in Figure 8. We can see that OrbitCache provides high throughput regardless of skewness, unlike the others whose throughput decreases as the workload becomes more skewed. NetCache does not provide high throughput as expected since many hot items are not cacheable. In the Zipf-0.99 workload, OrbitCache has higher throughput than NoCache and NetCache by 3.59× and 1.95×, respectively. The server throughput in OrbitCache is consistent across the given skewness, and this means that the loads are balanced.

Individual server loads. We plot the load on individual servers in Figure 9. We can see that NoCache and NetCache do not balance loads well. This is due to the lack of caching logic for NoCache and many uncacheable hot items for NetCache. However, OrbitCache can balance the loads since OrbitCache can cache variable-length items.

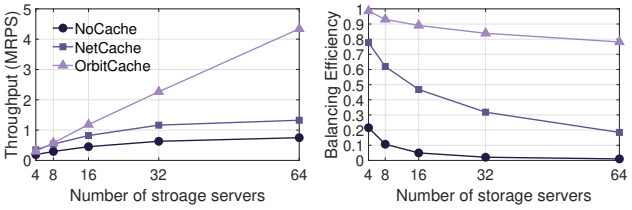
Latency vs. throughput. We measure the latency by varying Tx throughput. Figure 10 shows the median and the 99th percentile latencies as a function of Rx throughput. OrbitCache provides the best throughput but slightly higher latency than NetCache of 1 microsecond. This is because NetCache has a larger cache size, and requests handled by the switch dominate in latency data. In addition, in OrbitCache, requests for cached items should wait until cache packets read them. Although NetCache has slightly better latency, throughput is very limited since it fails to balance loads.

Impact of write ratio. Figure 11 reports the throughput as a function of write ratios. The throughput of OrbitCache



(a) Median (b) 99th percentile

Figure 10: Latency vs. throughput.



(a) Throughput (b) Balancing efficiency

Figure 12: Scalability.

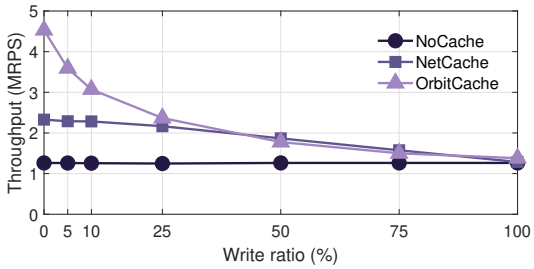


Figure 11: Impact of write ratio.

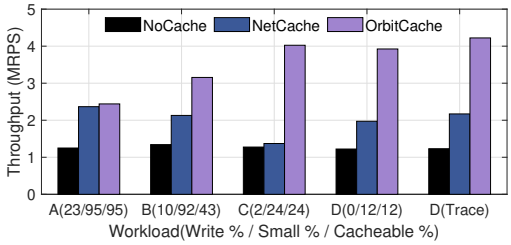


Figure 13: Performance with production workloads.

decreases as the write ratio grows. This is because the switch invalidates the cached key when handling a write request for a cached item to ensure cache coherence. The switch cache provides performance gains for read requests, the throughput is decreased with higher write ratios since read requests for invalid cached items are forwarded to storage servers. With the 100% write ratio, the throughput of OrbitCache converges to NoCache since the cache does not provide any benefit. Meanwhile, similar to OrbitCache, NetCache offers reduced throughput as the write ratio grows since it also invalidates the cached key if there is a write for the key.

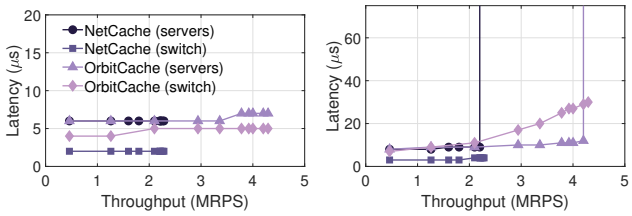
Scalability. We now inspect whether OrbitCache can balance loads of different numbers of servers. In this experiment, we limit the Rx throughput to 50K RPS to ensure that the bottleneck occurs at the storage servers rather than the clients, even when using 64 servers. In Figure 12 (a), we can see that the throughput of OrbitCache is improved almost linearly while the others do not. Figure 12 (b) clarifies that this is because NoCache and NetCache fail to balance imbalanced loads. *Balancing efficiency* is defined as the minimum throughput between the servers divided by the maximum throughput between the servers.

Performance with production workloads. We conduct experiments with several workloads of Twitter [37] to see how OrbitCache works with various production workloads. We pick 5 workloads based on the cacheable item ratio. We assign the workload ID A to D for `Cluster045/016/044/017`. We still use the 16-B keys for simplicity but vary the write ratio, the portion of 64-B values. Unlike the other experiments, the cacheable item ratio is controlled by choosing keys with a uniform distribution independent of the portion of 64-B values. We also use a non-bimodal version of workload D, by referring to the real trace of values in `Cluster/017` to show the fidelity of bimodal distributions in our experiments.

Figure 13 shows the results. Although the difference in throughput varies depending on the used workloads, we can see that OrbitCache shows the best performance for all the workloads. There is a little difference for Workload A. This is because NetCache can cache 95% of items, and the write ratio is relatively high. Workload E is a workload that shows a significant performance gap. This is because only 1% of items are cacheable for NetCache. We can also see that the trend in workloads D and D(Trace) is very similar. The slight throughput difference in workload D(Trace) is because the real trace contains more item values of less than 1024 bytes than the bimodal version. This demonstrates that our bimodal-distributed workloads successfully reflect the characteristics of real-world workloads.

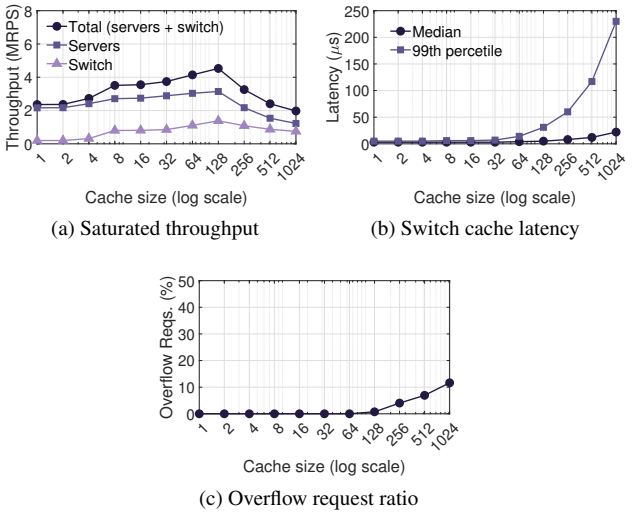
5.3 Deep Dive

Latency breakdown. Figure 14 (a) and (b) plot the median and the 99th percentile latency breakdowns, respectively. In the median latency, OrbitCache shows slightly higher switch latency than NetCache. This is because requests in OrbitCache should wait until circulating cache packets read them, resulting in latency overhead. In Figure 14 (b), we can see that the tail latency of OrbitCache (switch) increases as throughput grows. This is not surprising because we use a circular queue-based request table to maintain request metadata and packet cloning for a cache hit, resulting in additional latency overhead. We believe that this is acceptable since Or-



(a) Median (b) 99th percentile

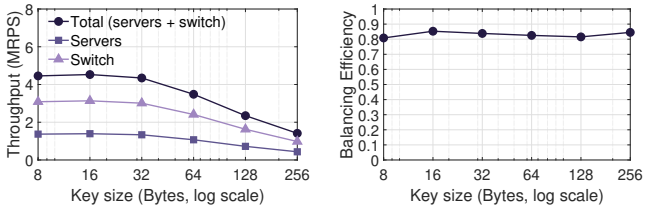
Figure 14: Latency breakdown.



(a) Saturated throughput (b) Switch cache latency

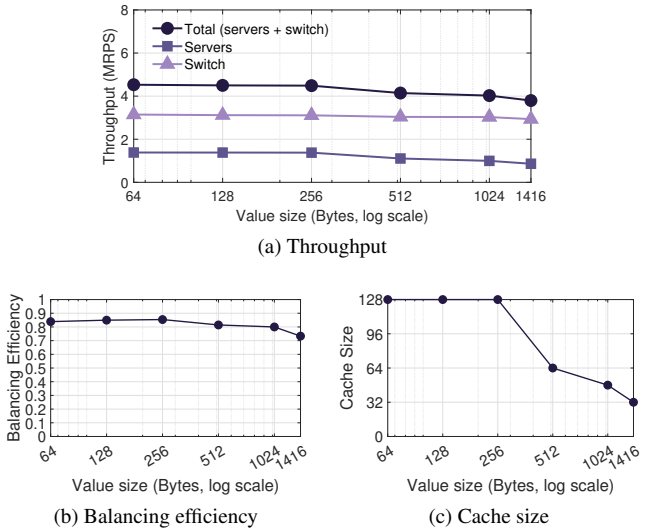
(c) Overflow request ratio

Figure 15: Impact of cache size.



(a) Throughput (b) Balancing efficiency

Figure 16: Impact of key size.



(a) Throughput

(b) Balancing efficiency (c) Cache size

Figure 17: Impact of value size.

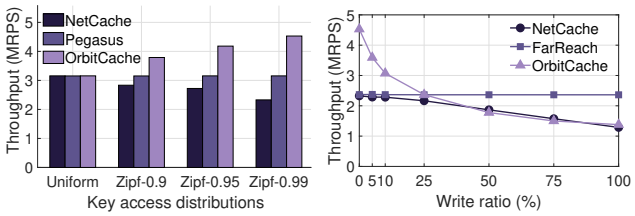
bitCache can provide much higher throughput than the others. In addition, the tail latency of the switch is still tens of microseconds even when the tail latency of servers soars by reaching the saturated throughput. We can decrease the tail latency of the switch by reducing the cache size as well. If the cache is based on a commodity server, the tail latency would be 10-100× longer than the switch cache.

Impact of cache size. Figure 15 plots the throughput breakdown, request latency handled by the switch, and the overflow request ratio of OrbitCache with different cache sizes. We can see that the total throughput increases as the number of cached entries grows. However, the throughput is saturated with around 128 cached items. Similarly, in Figure 15 (b), the tail latency increases quickly after 64-128 cached items. Figure 15 (c) clarifies the reason. From 256 cached items, the overflow request ratio rapidly increases, where the overflow requests indicate the requests for cached items forwarded to the storage servers due to the lack of free slots in the request table. The lack of free slots is caused by the excessive queueing delay between too many cache packets beyond the capability of the switch hardware. This clarifies that the cache size is the key to determining the performance. This also means we should choose an effective cache size between 32 and 128 to balance the trade-off.

Impact of key size. To see the impact of key size, we measure throughput and balancing efficiency using different key sizes. To show the impact of key size more clearly, we use a value of 100% 64-B. Figure 16 (a) shows that throughput decreases as key size increases. This is because the server consumes more computing power when key size is large. Figure 16 (b) shows that balancing efficiency remains high regardless of throughput changes.

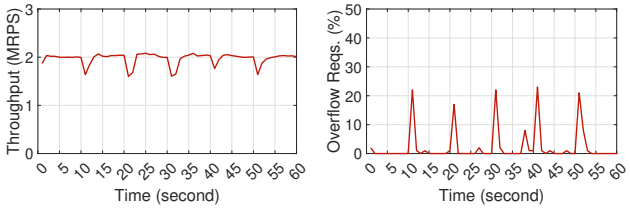
Impact of value size. We measure throughput, balancing efficiency, and the maximum cache size by varying the value size. In this experiment, we use the 100% same value size for all items to inspect how OrbitCache works in the worst case. Note that 16-B key and 1416-B value are the maximum size for a single packet payload with 28-B custom header fields.

Figure 17 (a) shows that OrbitCache can balance various workloads, including a workload where 100% values are MTU-sized. The slight drops in throughput are due to the increased value size. Figure 17 (b) shows the balancing efficiency. We see that OrbitCache generally maintains high balancing efficiency. Figure 17 (c) plots the effective cache size that provides the highest performance gain. We can see that



(a) Impact of skewness (Pegasus) (b) Impact of write ratio (FarReach)

Figure 18: Comparison to Pegasus [27] and FarReach [34].



(a) Throughput (b) Overflow request ratio

Figure 19: Performance with dynamic workloads.

the effective cache size decreases as the value size increases. This is because larger cache packets consume more switch resources, making the effective cache size small. Nevertheless, we can see that OrbitCache can balance the workloads well since we need only a small number of cached items to balance skewed workloads.

Comparison to Pegasus and FarReach. We now compare OrbitCache against Pegasus [27] and FarReach [34]. FarReach enables write-back caching in NetCache. Pegasus [27] is a selective replication solution where the switch replicates hot items across storage servers instead of caching items. We measure throughput using different key access distributions for Pegasus and different write ratios for FarReach. Figure 18 (a) shows the result for Pegasus. We can see that OrbitCache outperforms Pegasus for all the key access distributions. This is because the switch of OrbitCache provides extra throughput, whereas the throughput of Pegasus is limited to the throughput of storage servers. Pegasus is better than NetCache as it provides variable-length items. Figure 18 (b) shows the result for FarReach. OrbitCache outperforms FarReach until 25% of write ratio since FarReach has a limited cacheable item size like NetCache. However, when the write ratio grows, the performance gap with OrbitCache decreases. Over 25% of write ratio, FarReach shows better throughput than OrbitCache. This is because the write latency of FarReach is shorter than OrbitCache as it only updates the value of the item in the switch using write-back caching. OrbitCache updates the value of the switch and the server at the same time.

Handling dynamic workloads. We investigate how OrbitCache reacts to dynamic workloads. Like existing works [21, 29, 34], we use a *hot-in* pattern, which is the most radical workload change. We use 4 storage servers without server emulation and Rx rate limits similar to a previous work [34] to avoid inaccuracy due to the system state change. Every 10 seconds, the popularity of the 128 coldest items and the 128 hottest items is swapped.

Figure 19 (a) shows the throughput for 60 seconds. We can see that the throughput decreases when key popularity changes but recovers within a few seconds. This is because the controller in the switch control plane quickly updates the cache entries and fetches cache packets based on the top-k report of servers and the cache popularity counter of the switch data plane. Figure 19 (b) plots the change in the overflow request ratio over time. We can see that the ratio soars when the key popularity changes. This is because it takes time to fetch items from storage servers.

6 Related work

Storage systems are major target domains of recent in-network computing works (e.g., [11, 14, 18–25, 27, 28, 36, 38–40]). NetCache [21] shows the potential and limitations of switch-based caching. The limitation in the item size is passed down to recent works like FarReach [34] and DistCache [32]. OrbitCache enables variable-length in-network caching by leveraging built-in features of the switch. SwitchKV [29] employs the Openflow switch for cache lookups while items are cached in a cache node. Although this reduces the lookup overhead, it still provides limited performance due to server-based caching.

7 Conclusion

We proposed OrbitCache, an in-network caching architecture that is capable of variable-length caching in programmable switches. The key idea of OrbitCache is to make cached key-value pairs revisit the switch data plane continuously by leveraging packet replication efficiently. Experimental results demonstrated that OrbitCache can balance skewed workloads with diverse conditions. We hope that this work can contribute to the research community by providing insights that utilizing built-in hardware features has great potential to make in-network computing solutions effective.

Acknowledgement

The author would like to thank the shepherd, Xin Jin, and the anonymous reviewers for their insightful comments and feedback. The author also thanks Yeon-sup Lim for providing constructive feedback on the earlier version of the paper. This research was sponsored by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (No. RS-2025-00522990).

References

[1] Tommyds c library. https://www.tommyds.it/, 2018.

[2] Redis key-value store. https://redis.io/, 2023.

[3] Tofino switch. https://github.com/barefootnetworks/Open-Tofino, 2023.

[4] Nvidia messaging accelerator (vma). https://docs.nvidia.com/networking/display/vmav9840, 2024.

[5] Rocksdb: A persistent key-value store for flash and ram storage. https://rocksdb.org/, 2024.

[6] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proc. of ACM SOSP*, page 1–14, New York, NY, USA, 2009. Association for Computing Machinery.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, page 53–64, New York, NY, USA, 2012.

[8] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. Scaling out to a single-node 80gbps memcached server with 40terabytes of memory. In *Proc. of USENIX HotStorage*, pages 8–12, USA, 2015.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proc. of ACM SIGCOMM*, pages 99–110, 2013.

[11] Tommaso Caiazzi, Mariano Scazzariello, and Marco Chiesa. Millions of low-latency state insertions on asic switches. *Proc. of ACM CoNEXT*, 1, nov 2023.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proc. of USENIX FAST*, Santa Clara, CA, February 2020.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of ACM SoCC*, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[14] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, aug 2020.

[15] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proc. of ACM SoCC*, New York, NY, USA, 2011.

[16] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[17] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. Leed: A low-power, fast persistent key-value store on smartnic jbofs. In *Proc. of ACM SIGCOMM*, page 1012–1027, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. P4db - the case for in-network oltp. In *Proc. of ACM SIGMOD*, page 1375–1389, 2022.

[19] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. *Proc. VLDB Endow.*, 14(9):1626–1639, may 2021.

[20] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proc. of USENIX NSDI*, pages 35–49, Renton, WA, April 2018.

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

[22] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proc. of ACM SIGCOMM*, page 90–106, 2020.

[23] Gyuyeong Kim. Holistic in-network acceleration for heavy-tailed storage workloads. *IEEE Access*, 11:77416–77428, 2023.

[24] Gyuyeong Kim. Netclone: Fast, scalable, and dynamic request cloning for microsecond-scale rpcs. In *Proc. of ACM SIGCOMM*, page 195–207, September 2023.

[25] Gyuyeong Kim and Wonjun Lee. In-network leaderless replication for distributed data stores. *Proc. VLDB Endow.*, 15(7):1337–1349, Mar. 2022.

[26] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proc. of ACM SOSP*, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proc. of USENIX OSDI*, pages 387–406, November 2020.

[28] Junru Li, Youyou Lu, Yiming Zhang, Qing Wang, Zhuo Cheng, Keji Huang, and Jiwu Shu. Switchtx: Scalable in-network coordination for distributed transaction processing. *Proc. VLDB Endow.*, 15(11):2881–2894, jul 2022.

[29] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *Proc. of USENIX NSDI*, pages 31–44, USA, 2016.

[30] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proc, of USENIX NSDI*, page 429–444, USA, 2014.

[31] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proc. of ACM ASPLOS*, page 795–809, New York, NY, USA, 2017.

[32] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. of USENIX FAST*, pages 143–157, Boston, MA, February 2019.

[33] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proc. of USENIX NSDI*, pages 385–398, Berkeley, CA, USA, 2013.

[34] Siyuan Sheng, Huancheng Puyang, Qun Huang, Lu Tang, and Patrick P. C. Lee. FarReach: Write-back caching in programmable switches. In *Proc. of USENIX ATC*, pages 571–584, Boston, MA, July 2023. USENIX Association.

[35] Vishal Shrivastav. Stateful multi-pipelined programmable switches. In *Proc. of ACM SIGCOMM*, page 663–676, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with In-Network cache coherence. In *Proc. of USENIX FAST*, pages 277–292, February 2021.

[37] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *Proc. of USENIX OSDI*, pages 191–208. USENIX Association, November 2020.

[38] Parham Yassini, Khaled Diab, Saeed Mahloujifar, and Mohamed Hefeeda. Horus: Granular In-Network task scheduler for cloud datacenters. In *Proc. of USENIX NSDI*, pages 1–22, Santa Clara, CA, April 2024.

[39] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proc. of ACM SIGCOMM*, page 126–138, 2020.

[40] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.

CellReplay: Towards accurate record-and-replay for cellular networks

William Sentosa[†], Balakrishnan Chandrasekaran[‡], P. Brighten Godfrey[†•], Haitham Hassanieh[◇]
[†]*University of Illinois Urbana-Champaign,* [‡]*VU Amsterdam,* [•]*Broadcom,* [◇]*EPFL*

Abstract

The inherent variability of real-world cellular networks makes it hard to evaluate, reproduce, and debug the performance of networked applications running on these networks. A common approach is to record and replay a trace of observed cellular network performance. However, we show that the state-of-the-art record-and-replay technique produces empirically inaccurate results that can cause evaluation bias. This paper presents the design and implementation of CellReplay, a tool that records the time-varying performance of a live cellular network into traces using preset workloads and faithfully replays the observed performance for other workloads through an emulated network interface. The key challenge in achieving high accuracy is to replay varying network behavior in a way that captures its sensitivity to the workload. CellReplay records network behavior under two predefined workloads simultaneously and interpolates upon replay for other workloads. Across various challenging network conditions, our evaluation shows that real-world networked applications (e.g., web browsing or video streaming) running on CellReplay achieve similar performance (e.g., page load time or bitrate selection) to their live network counterparts, with significantly reduced error compared to the prior method.

1 Introduction

Cellular network performance, including bandwidth and latency, can vary significantly due to factors such as wireless interference, environmental obstructions, and handovers, especially in mobile environments [12, 13, 22, 23, 34]. The gold standard for evaluating application and protocol performance on cellular networks is, hence, to test them directly on *live* cellular networks. However, live testing is time-consuming, as experiments must be conducted across many different network conditions and can produce different results—due to different signal strengths, types of wireless service (e.g., 5G millimeter wave, 5G low-band, and 4G), kinds of interference, rates of mobility, physical locations, etc. Repeating each experiment multiple times is crucial to ensure statistically reliable results given the performance variability in cellular networks. In addition to being time-consuming, experiments are often difficult to reproduce. A lack of control over the environment makes it infeasible, for instance, to compare the effects of a protocol change under identical network conditions.

Thus, researchers and app developers often turn to simulation or emulation for much of their evaluation, hoping to replicate a representative environment that yields performance similar to a live network. Simulators and emulators, such as `ns3` [28] or Linux's `netem-tc`, offer various options for configuring delay, jitter, bandwidth, packet loss, and more. While they can adjust these parameters to emulate specific and realistic conditions, properly tuning them to accurately represent the dynamic behavior of real-world cellular networks remains a challenging and open problem.

A more realistic approach is to *record* network performance traces (e.g., latency, bandwidth, or packet loss) over time using predefined workloads (e.g., RTT probing) on a real-world network and *replay* those traces in an emulated network for the tested apps. This method allows for recording different traces under various conditions (e.g., locations) and testing multiple apps using such recorded traces. Record-and-replay emulation was pioneered by Noble et al. [27]. More recently, the Mahimahi network emulator [25] can also replay recorded cellular network traces and has been instrumental in the design and evaluation of several notable networked systems and protocols (e.g., [5,14,18,20,24,26,30,31,36,37,39,41,43,45,46]).

However, we found that Mahimahi can produce inaccurate results compared to real-world tests in important cases, particularly for latency-sensitive and bursty workloads. For instance, in our evaluation, we observed an average bias of approximately 17.1% in web page load times (PLTs) when comparing Mahimahi emulation to running the application in the same commercial cellular environment where the traces were recorded. This error is a persistent underestimation of the PLT rather than just random variation. This issue affects other applications as well and the error may even be greater. For example, we observed a 49% error for 250 KB file downloads when Mahimahi emulated a commercial Verizon 5G as shown in §5.6.

Thus, despite record-and-replay emulation being practical and widely used, it does not support high-fidelity testing of networked systems and protocols. Minimizing emulation error is crucial, particularly for wireless protocol and application research, where record-and-replay emulations are often the most feasible evaluation platform. These errors could affect any evaluation and may even alter its conclusions, as we demonstrated in the ABR algorithms use case (§5.9). Therefore, we asked: What is causing this emulation error? And, is there a way to fix it to faithfully record and replay real-world cellular network performance?

Our first contribution is to study how the record-and-replay method used by Mahimahi can result in persistent bias (§3). Mahimahi records *packet delivery opportunities* by continuously saturating the link with packets (a "saturator" workload)

and noting when packets arrive at the endpoint. It then replays this trace as a schedule for when the link can deliver packets after delaying those packets using a *fixed* propagation delay, for any workload.

However, we found that this method causes two fundamental issues. First, it fails to fully capture network base latency changes, which are prevalent in cellular networks. In fact, our measurements show that Mahimahi underestimates RTT by 13.25% and 16.88% across two operators. Second, the available bandwidth that a cellular network provides to an end-to-end connection depends significantly on that connection's workload[1]. For example, in our measurement using Verizon 5G, a long train with 100 back-to-back packets experiences 2.6 times higher delivery rate than a short train with 10 packets. In such cases, Mahimahi's saturator (i.e., heavy traffic) would see a higher rate than what shorter traffic should experience. This dependency between cellular network available bandwidth and workload poses a fundamental challenge for record-and-replay because the whole point is to record *one* trace (which is necessarily running one workload) and replay that trace under a *variety* of applications for testing. If available bandwidth depends on the workload, is faithful record-and-replay feasible?

Our second contribution is to address these fundamental problems in a record-and-replay system called *CellReplay*. To solve the workload-dependence problem, one obvious approach would be to record performance under every possible workload. However, this is impractical and degenerates into simply testing every application directly on the live network, which is what record-and-replay emulation is trying to avoid. In other words, we can only record a *limited* number of different workloads. Another option would be to build a white-box emulation of providers' underlying resource allocation policies; but these are proprietary and vary across providers, so we seek a black-box method based on end-to-end observations.

The approach we take is to record just two representative workloads (*light* and *heavy*) simultaneously, chosen at extremes on the range of traffic patterns, and then interpolate between them during replay to achieve high accuracy across a wide range of workloads. During the recording phase, we use two phones: one running a heavy saturator workload and the other running a light workload. The light workload is calibrated to capture RTTs and light-workload bandwidth, but is not too light as to capture the network's transition from light to heavy bandwidth allocations. During replay, the emulator applies delay using the RTT trace and initially releases packets according to the light trace. It then splices in the heavy trace during longer packet sequences before eventually returning to the light trace after an idle period. This technique addresses the two key problems with Mahimahi's approach mentioned above, namely capturing (1) dynamic RTTs and (2) bandwidth that depends on workload.

We implemented CellReplay using an architecture similar to Mahimahi—an emulated network interface that can be used by unmodified applications. Using randomized trials, we evaluated CellReplay's accuracy by comparing the application performance when running under CellReplay emulation to the live networks. We tested two commercial providers' 5G mid-band and low-band deployments, and covered multiple network conditions, including non-ideal conditions (e.g., in a crowded library) and mobility (e.g., driving). We evaluated two real-world application traffic patterns: randomized file downloads and web page loads with HTTP/1.1 and HTTP/2. These applications cover a variety of workloads, ranging from periodic small to heavy flows in file downloads to complex interleaved traffic from web page loads. Additionally, we used CellReplay to evaluate the startup phase of multiple adaptive bitrate (ABR) implementations for 4K video streaming.

We find that CellReplay substantially reduces emulation error. In web page load tests, CellReplay reduces emulation error from 17.1% with Mahimahi to 6.7%, representing a 60.8% improvement. For randomized file download tests, CellReplay lowers mean file download time errors from 7.9%-49% with Mahimahi to just 0.2%-22.4%. Moreover, CellReplay achieves lower error when replicating application performance under non-ideal network conditions, such as inside a basement (15.22% error in Mahimahi vs. 5.87% in CellReplay) and a crowded library (22.51% vs. 8.47%), and during user mobility, such as walking (14.48% vs. 4.13%) and driving (13.15% vs. 6.97%). Finally, we demonstrate CellReplay's usefulness in evaluating ABR algorithms, as it preserves the relative ordering of ABR performance and avoids the biases observed in Mahimahi. We discuss challenges and future directions for improvement in §6. We release CellReplay alongside with its recorded traces as an open source at https://github.com/williamsentosa95/cellreplay

2 Background and related work

2.1 Cellular network record-and-replay

The goal of record-and-replay network emulation (within the scope of this paper) is to emulate the end-to-end network performance of an application communicating between two endpoints, ensuring performance similar to that of a live network counterpart. During the recording phase, user equipment (UE) and the server send traffic according to a predefined workload (e.g., sending packets beyond the link bottleneck rate), while observed performance metrics (e.g., throughput) are logged. This workload should be *independent* of the tested applications, allowing us to record traces *once* and reuse them for multiple applications, regardless of whether they are UDP- or TCP-based. During replay, this trace is consumed by an emu-

[1]To be clear, this effect is not due to queuing behavior of a traditional link with constant throughput and latency; nor is it caused by variations in wireless physical channel quality over time. Although commercial providers' internal policies are proprietary and opaque to us, the effect could be explained by a resource allocation policy (e.g. [9]) or a carrier aggregation policy (as observed in [40]) at the Radio Access Network (RAN) allocating some bandwidth for the client, but observing the client's injected packets and dynamically modifying that provided bandwidth.

lated network interface. Real applications (e.g., a web browser and web server) can connect through this interface, and traffic between the endpoints will experience artificial network conditions (e.g., time-varying latency and bandwidth) as if they were communicating over a cellular network, even though they reside on the same physical host. Our goal is for any metrics of interest—including transport-level and application-level metrics such as flow completion time or web PLT—to closely match those of the live network.

Record-and-replay can be applied to any type of network, but our interest here is on cellular networks. Their performance can be time-varying, vendor-dependent, and environment-dependent, making it difficult to generate conditions—whether in simulators, emulators with hand-picked or even calibrated [42] parameters, or testbeds—that match real-world complexity. Thus, record-and-replay is especially useful in such environments, but it is also challenging to execute well.

Note that record-and-replay deals with end-to-end conditions and does not require any link- or physical-layer information or support from network operators. Like past work, we do not need to determine which hops along the path cause certain performance effects. This means that the observed performance, and its replay, may result from a combination of sources (e.g., the 5G RAN, service provider core, or the Internet to a remote endpoint). However, major performance variations are expected to originate from the cellular network [22]. We sometimes refer to the observed performance as coming from a cellular link, the path, or simply the network; all terms are equivalent for our purposes.

2.2 Related work

Network emulators. Popular network emulators, such as NetEm [16] and dummynet [32], can emulate cellular networks. Google Chrome also provides configuration profiles with fixed latency and bandwidth for cellular networks, such as "Fast" and "Slow" 3G [3]. Pantheon [42] provides calibrated emulators based on parameters like fixed propagation delay, bottleneck link rate, isochronicity, etc. These configurations are tuned to match packet traces collected from a path (including cellular network) using various congestion control protocols. iBox [7] extends this by incorporating cross-traffic. However, fixed parameters, by definition, do not capture time-varying effects, which are common in cellular networks.

Record-and-replay network emulation. Noble et al. pioneered the concept of recording the end-to-end network characteristics of a wireless network and replaying them in an emulated network in 1997 [27]. However, it is designed to emulate WaveLAN, which differs fundamentally from modern cellular networks. More recently, NemFi [21] was introduced as a record-and-replay emulator for WiFi. NemFi's design is specific to WiFi (e.g., emulating frame aggregation) and it is not readily applicable to emulating cellular network paths.

Mahimahi. In 2015, Netravali et al. demonstrated a framework for recording and replaying HTTP traffic [25] called Mahimahi, which also included a network emulator derived from CellSim [38] to replay time-varying uplink and downlink rates in cellular networks. Mahimahi has since become the state-of-the-art record-and-replay emulator for cellular networks and is widely used to evaluate various networked applications.

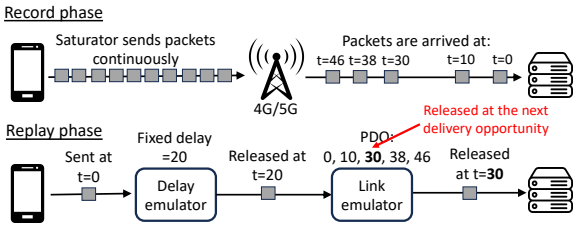


Figure 1: *An overview of Mahimahi's record-and-replay to emulate cellular uplink.*

We detail Mahimahi's record-and-replay approach[2], as it serves as an important reference for this paper. Fig. 1 illustrates the process for the uplink only, as the same approach applies to the downlink. Mahimahi records time-varying link rates using a *Saturator*, which saturates both the uplink and downlink with MTU-sized packets (e.g., 1500 bytes) to ensure that the base station always has packets to deliver. The endpoint then records the arrival time of each packet. During emulation, Mahimahi treats each arrival timestamp as an opportunity to deliver a packet. A sequence of such timestamps constitutes a *packet delivery opportunity (PDO) trace.* Each PDO entry represents an opportunity to deliver an MTU-sized amount of data, which can be either a single MTU-sized packet or multiple smaller packets whose combined sizes add up to the MTU. If no packets are queued for delivery when the PDO occurs, the opportunity is lost. Mahimahi also emulates the RTT delays on a cellular link, albeit using a *fixed* propagation delay. That delay is determined by measuring the minimum packet RTT (e.g., via ICMP ping) and halving that value.

3 Live record-and-replay is hard

Why is record-and-replay challenging? Also, why does the current state-of-the-art method (i.e., Mahimahi) fail to accurately replicate the performance of networked applications on a cellular network? Below, we answer both questions using measurements and insights from real cellular networks.

The measurements in this section were collected from two commercial cellular networks: T-Mobile 5G mid-band and Verizon 5G low-band, using a Samsung Galaxy S22 (SGS) phone tethered to a laptop. The laptop, equipped with an Intel i7 CPU and 16 GB RAM, ran Ubuntu 20.04 and served as our

[2]This method was introduced by CellSim. Mahimahi also provides traces recorded in CellSim's approach, which have been beneficial and used in past work (e.g., [20]). Due to its usefulness, other work has collected newer cellular network traces following CellSim's approach (e.g., [19, 26]) and replayed them in Mahimahi. For simplicity, throughout the rest of this paper, we refer to this record-and-replay method as Mahimahi.

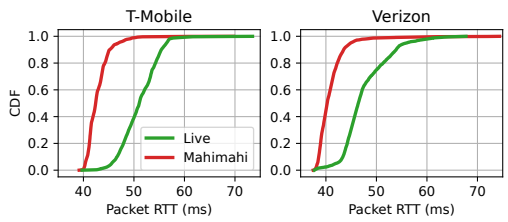


Figure 2: *Packet RTT CDF of Mahimahi emulation and live 5G networks. Mahimahi underestimates the RTT and fails to capture the RTT distribution.*

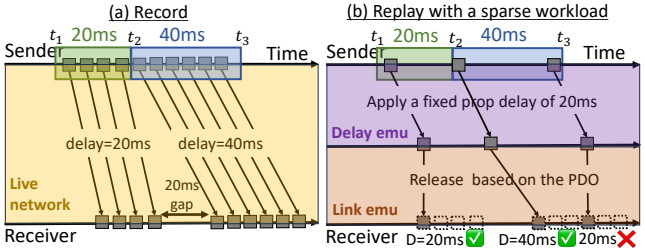


Figure 3: *Mahimahi PDO approach fails to capture the base delay changes for sparse workload.*

client. The server was located within close proximity (<10 miles) of the client. We confirmed that all results remained consistent when using a different phone model (Google Pixel 5).

3.1 Variability in base RTT

The base RTT is defined as the round-trip time (RTT) of a packet from the client to the server when there is no self-inflicted congestion. In cellular networks, this RTT is expected to be variable, as packets frequently experience delays (jitter) due to link-layer retransmissions, channel contention, base station scheduling, and device mobility. Emulating this variability is critical for testing latency-sensitive applications such as VR/AR and remote driving.

Mahimahi also emulates delay variability based on packet delivery traces. Despite using a fixed propagation delay, it must hold the packet until it sees a PDO before releasing it (Fig. 1). However, our measurements suggest that it fails to fully capture the base RTT variability. To quantify this error, we compared the packet RTT reported in live experiment with that of Mahimahi.

Specifically, we conducted repeated packet RTT tests and Mahimahi recordings individually over live networks, following the randomized trial approach (§5.3). The packet RTT test involves a client sending a 1400-byte UDP packet (roughly an MTU-size) every 50 ms to our echo server and noting each packet's RTT. We repeated this test 10 times, and both the RTT test and Mahimahi recording session lasting 60 seconds. Next, we ran the exact same packet RTT test under Mahimahi's emulated interface, using the recorded trace and setting the propagation delay to half of the minimum RTT from the live packet RTT tests.

Figure 2 shows the cumulative distribution function (CDF) of packet RTTs on the live network and Mahimahi replay. It indicates that Mahimahi underestimates packet RTT (by 16.88% and 13.25% at the median for T-Mobile and Verizon, respectively), and its distribution differs from that of the live network (as seen in the shape of the CDF curve). This suggests that simply increasing Mahimahi's fixed propagation delay (i.e., shifting the CDF curve to the right) does not capture the variability. Note that this experiment was performed under stationary conditions with a strong signal, where network performance is more stable. In a mobile scenario, where packet RTT can vary more or even change, Mahimahi's fixed propagation delay approach may perform even worse.

This is because PDOs, in principle, only *partially* capture base delay changes. Figure 3 illustrates a case where the packet base delay changes at t_2 from 20 ms to 40 ms, and Mahimahi fails to apply the correct delay for a certain packet. Note that base delay changes may occasionally occur in live cellular networks due to factors such as increased retransmission delays caused by a weakened radio signal. In this illustration, during the recording phase, the four packets delivered from t_1 to t_2 experience a 20 ms base delay, while packets from t_2 to t_3 experience a 40 ms base delay. The receiver indeed perceives a delay change since, after receiving four packets, it does not receive any packets for 20 ms before receiving the next set. This 20 ms "blackout" period is also reflected in the PDO trace during the replay.

A PDO blackout means no packet delivery. Any packets scheduled for delivery during this period will be delayed until the next available opportunity. As a result, only packets arriving during the blackout period (relative to the link emulator) will experience a delay, while others will *not*. However, sparse workloads, such as those in Fig. 3b, may have packets arriving outside the blackout period and thus not experiencing any delay.

Conclusion: Fixed propagation delay and PDOs are insufficient to model cellular network delay variability. Therefore, we need to record packet RTTs over time through probing during the recording phase and apply time-varying delays during the replay.

3.2 Performance depends on workload

Recall that Mahimahi uses a Saturator to ensure that the network always has packets ready to send, and so any available PDOs will be consumed and recorded. Then, a subset of those PDOs is used when replaying any given workload. An underlying assumption is that the same PDOs would have been available for the replayed workload. However, our experiments on live cellular networks show that *the network substantially changes the PDOs it provides depending on the workload*. We reached this conclusion upon observing that short flows consistently experience lower bandwidth than longer flows.

To demonstrate this, we conducted live experiments in which our server periodically sent *packet trains* to a client. Each train consists of N back-to-back UDP packets, each

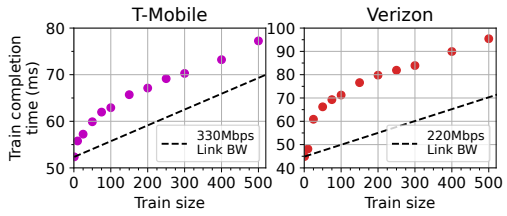


Figure 4: *Mean train completion time (TCT) of different sized trains (N), and TCT if trains were delivered according to the Saturator's observed bandwidth.*

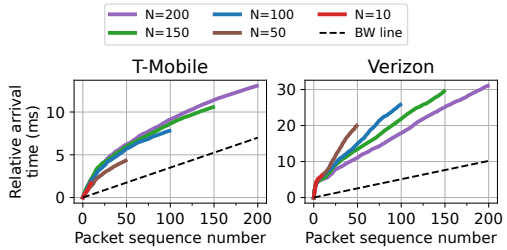


Figure 5: *Mean relative arrival time for each packet of different sized (N) trains.*

1400 bytes, followed by a 100 ms gap—long enough to clear out any packets from previous trains). We refer to N as the train size. Each packet in a train is tagged with a train number and a sequence number reflecting its order within a train $(P_0, \ldots, P_{N-1})$. The client then records each packet's arrival time. Additionally, the client sends a 100-byte ACK back to the server upon receiving the last packet of a train (P_{N-1}). The *train completion time* (TCT) is defined as the time at which the server receives the ACK minus the time it sent P_0. We performed this test with different train sizes following our randomized trial approach (§5.3). Each test lasted 5 seconds, and there are 12 tests with different train sizes (1, 10, 25, 50, ..., 500) in one trial. We repeated this trial 50 times. Finally, we also recorded network performance with the Saturator, as Mahimahi would.

Figure 4 shows the mean TCT for each train size on T-Mobile and Verizon 5G. The dashed black line represents the TCT if the link had a fixed bandwidth equal to that observed by the Saturator (equivalent to the mean TCT with Mahimahi replaying the Saturator). If network performance were independent of workload, then the mean bandwidth would remain the same for all train sizes, and the mean TCT would follow a linear function of the amount of data being delivered, i.e., a linear function of N. In particular, it should coincide with the bandwidth observed by Saturator. However, the observed TCTs do not conform to a straight line and generally do not match the Saturator line, with TCTs being up to 11.5% higher than the Saturator in T-Mobile and 35.8% higher in Verizon. This indicates that *the service experienced by the train workloads is consistently and significantly different than the service experienced by the Saturator's heavy workload.*

To better understand these observations, we examined the arrival times of packets within each train, revealing the PDO patterns recorded by different trains. For each packet $P_i \in \{P_0, \ldots, P_{N-1}\}$ in each train, we calculated its *relative arrival time* as $t(P_i) - t(P_0)$, where $t(\cdot)$ denotes the receiver's observed arrival time of a packet. We present the mean relative arrival time as a function of packet sequence number i for different train sizes N in Figure 5.

These results confirm that the link's rate (or PDO) depends on the workload. The network provides a lower delivery rate for the first few packets in any train (note that in these plots, a higher slope indicates a lower delivery rate). As the train progresses, the delivery rate increases, and the slope begins to approach that of the Saturator. We also repeated the same packet train tests in the opposite (uplink) direction but found that the link's rate remains uniform regardless of N (i.e., it follows a straight line). We suspect that the rate-workload dependence in the downlink results from the operator's proprietary packet scheduling implementation. For example, cellular network packet scheduling may depend on historical application traffic and the current queue depth when scheduling packets [9]. This rate-workload dependence was observed across all conditions tested in §5, including both peak and off-peak hours.

Interestingly, T-Mobile and Verizon have dramatically different implementations. T-Mobile's delivery rate approaches the Saturator's rate (slope) as i increases, mostly regardless of N. Verizon's delivery rate, on the other hand, asymptotes to significantly different values depending on N, with larger N approaching the Saturator's heavy-workload delivery rate more closely. Additionally, in T-Mobile, the first 50 packets of trains with $N > 50$ are delivered more slowly than the 50 packets of the train with $N = 50$, whereas Verizon shows an inverted behavior. This further complicates record-and-replay, as we aim for a general and relatively accurate approach across different operators and locations.

We also found a more minor way in which performance depends on workload: the RTT of a packet varies with packet length by an amount not explained by throughput. For instance, based on our measurements on Verizon, the RTT of a 100-byte packet is 6.8 ms faster than that of a 1400-byte packet, even though the difference in serialization time at the bottleneck link rate (60 Mbps) should have been only ≈ 0.17 ms. This outcome aligns with findings from the prior latency study on 5G [12] and may be attributed to the additional time required for reassembling larger data chunks. Due to space constraints, we omit detailed results.

Conclusion: Cellular network performance can depend significantly on the workload. Our observations indicate that cellular providers allocate delivery rates (i.e., bandwidth or PDOs) differently for *light* and *heavy* workloads. The Saturator forces the link into its heaviest workload mode, which generally increases the available bandwidth. Consequently, using Saturator's PDOs for a lighter workload can result in consistent bias (flows complete faster than they should). This suggests that diverse workloads are needed to capture differ-

ent PDOs, but choosing the right representative workloads remains challenging.

4 CellReplay

4.1 Design overview

At a high level, we want to solve the problems of capturing time-varying base RTT (§3.1) and workload-dependent performance (§3.2) in both record and replay.

We begin with the latter problem (§3.2). To achieve highly accurate emulation, an obvious solution is to record performance under different workloads. However, recording every possible workload is impractical and degenerates into simply testing the apps directly on the live network. We also aim for the recorded workload to be independent of the tested apps (§2.1). Therefore, we can only record a *limited* number of different workloads.

From §3, we observe that short and continuous traffic are handled differently, while medium-length flows exhibit performance somewhere between the extremes. Inspired by this observation, our key approach is to record two workloads chosen at the extreme points on the spectrum of traffic patterns: (1) *Packet train probing* to capture link PDOs under short and bursty load (light PDOs), and (2) *Saturator* to capture PDOs under heavy continuous load (heavy PDOs). These workloads capture the essential behavior of link rate differentiation under light and heavy flows. We used two phones to record both traces simultaneously and show in our evaluation that interference between them is limited in practice.

During replay, we leverage both PDOs to match the provided workload. When the application under test begins sending packets, we initially release the first sequence of packets according to the light PDOs and then transition to heavy PDOs as the packet sequence lengthens. After a certain gap in the workload, we return to the light PDO trace.

Returning to the problem of time-varying RTT (§3.1), we design the packet trains to avoid inflating queues, so that it gives us a good measurement of base RTT. The packet train probing serves a dual purpose: to record changing base RTTs (for any workload) and PDOs for shorter packet sequences.

Finally, the effectiveness of the above design depends on parameter choices. For example, a too-small train will not capture the network's light workload behavior completely, forcing us to go to the heavy PDO trace too soon; if the trains are too large, we cannot sample frequently as network performance may then resemble that of a heavy workload, and there is a risk of inflating base RTT measurements due to congestion. Thus, before recording, we conduct a calibration phase to determine train size, train gaps, and other parameters that will yield the least error.

In summary, CellReplay has *three* components. When recording network traces in a specific environment, we first perform an automated **calibration** of parameters in that environment, and then start **recording** live traces by running packet train probing and Saturator in parallel. These traces

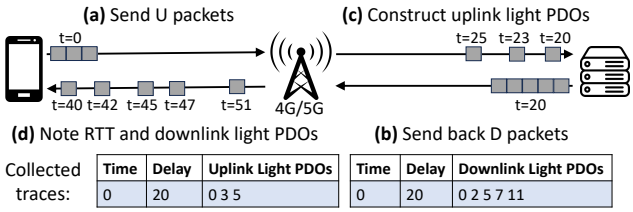


Figure 6: *Packet train workload alongside with its recorded base delay and light PDO traces for uplink and downlink.*

are then used to emulate the network during **replay**. The following subsections detail each of these components: record (§4.2) and replay (§4.3), before returning to calibration (§4.4), which is best understood after seeing the rest of the design.

4.2 Recording network traces

There are three time-series metrics we want to record: (1) base delay, (2) light PDOs, and (3) heavy PDOs.

Base delay and light PDOs. The base delay trace should reflect the network's round-trip time (RTT) without any queueing delays introduced by the workload itself. Ideally, this trace would be captured by periodically measuring the RTTs of small packets. The one-way base delay can be estimated by halving the RTT.

Light PDOs can be captured by periodically sending a limited number of back-to-back packets, i.e., a *packet train*, in both the uplink and the downlink. The number of packets should be small enough to capture the network's light workload behavior and ideally some of the transition to moderate workload, without pushing the network into heavy workload mode. In particular, the train should be short enough to avoid "warming up" the network for the following train. As a result, both the base delay trace and light PDOs share similar requirements. We can collect both simultaneously using a packet train probing workload on a single device. This workload uses MTU-sized packets, as a significant amount of traffic is still required to capture the transition point between light and heavy modes.

Figure 6 provides an example of how this process works. In every G ms, (a) the client sends U back-to-back MTU-sized packets to the server. Upon receiving the first packet of the train, (b) the server sends back D back-to-back MTU-sized packets. The server also (c) records each packet's arrival within that train and uses it to calculate the uplink light PDOs as the arrival time of each packet minus the arrival time of the first packet (since, during replay, the base delay will be added). When the client receives the corresponding downlink train, (d) it infers the current base RTT as the receipt time of the first downlink packet minus the send time of the first uplink packet (within that train). It then calculates the downlink light PDOs based on packet arrival times, just as the server did.

Heavy PDOs. The heavy PDOs are collected using a Saturator (similar to Mahimahi) that saturates the link with packets beyond its bottleneck rate, effectively "requesting" the link to remain in max bandwidth mode. In practice, we developed our

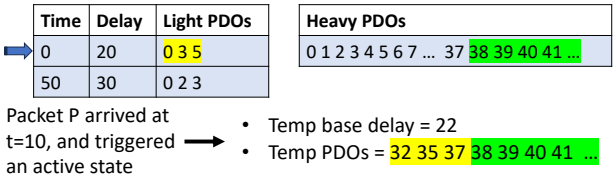


Figure 7: *Temporary base delay and PDOs constructed when a CellReplay first enters an active state.*

own Saturator tool, which sends MTU-sized packets in both the uplink and downlink at fixed upload and download rates, eliminating the need for two phones as in Mahimahi's Saturator [38]. We overestimated (by 25%) the max link bandwidth measured using an existing bandwidth test application like iperf or speedtest. We confirmed that the reported throughput from our Saturator is similar to UDP iperf.

However, running both Saturator and packet train probing on a single device is not feasible, as the Saturator will overload the queue, leading to two issues: inflating the base delay measurement and keeping the link in maximum bandwidth state. One solution is to run these workloads in separate trials, which may be permissible under stationary conditions but is less ideal under mobility. Alternatively, we chose to perform these workloads on separate identical phones placed in close proximity. This is possible since most (if not all) cellular network providers employ user-separated queues [38] such that the Saturator traffic will not inflate the packet train probing measurement results. Beyond the known separation of queues, we confirmed that light vs. heavy bandwidth allocation is also separated on both Verizon and T-Mobile: when one phone runs the Saturator, the other phone running packet train probing still experiences light-workload service.

Note that the two-phone method is not without limitations. The phones may not always connect to the same base station all the time, especially in a mobile environment where handoffs could occur slightly differently. We leave this discrepancy for future work.

4.3 Replaying network traces

CellReplay takes input traces of base delay, light PDOs, and heavy PDOs over time to emulate network performance in a virtual interface. At a high level, CellReplay first applies a base delay to each packet based on the delay trace, adjusts the delay for any latency offset from packet-size calibration (§4.4), and then releases packets according to either the light or heavy PDOs.

In more detail, CellReplay operates in two states: active and inactive. Initially, CellReplay is in the inactive state until it receives a packet at some time t, relative to the start of the emulation. This event triggers CellReplay to enter the active state, which involves preparation as shown in Figure 7. CellReplay searches for the most recent base delay ($DELAY$) and light PDOs ($LightPDO$) where the timestamp is $\leq t$. Since the trace is sampled per G, linear interpolation is used to assign $DELAY$ to packets arriving between two samples. CellReplay then saves $DELAY$ and constructs temporary PDOs ($TempPDO$) by adding every PDO entry in $LightPDO$ with $t + DELAY$. It then concatenates these with the suffix of the heavy PDOs, starting from $t + DELAY + max(LightPDO) + 1$. The system is now done entering the active state.

As long as the system remains in the active state, packets are initially delayed by $DELAY$ plus a size-based delay compensation $comp(size(P))$. As discussed in §3.2, base delay may depend on packet size; the specific adjustment $comp(\cdot)$ is determined during calibration. $DELAY$ and $TempPDO$ remain unchanged unless the system enters an inactive state[3]. After a packet is delayed, it is either placed in a PDO queue or dropped if the queue exceeds B bytes. Packets are dequeued according to the time schedule in $TempPDO$ using byte-wise dequeueing. This process mirrors Mahimahi's PDO replay, with CellReplay using the temporary (concatenated) PDO trace. As a result, early packets in the active state will experience light PDOs, while later packets will experience heavy PDOs. Once F milliseconds pass without any packets in the PDO queue, CellReplay returns to the inactive state. Any future arriving packet will then trigger the procedure to reenter the active state, as described above.

Table 1: *Parameters set in calibration phase (§4.4).*

Parameter	*Definition*
U	Number of packets per uplink train
D	Number of packets per downlink train
G_{min}	Lower bound of gap between trains (milliseconds)
F	Fallback timer to return to inactive state
$comp(s)$	Delay compensation for s-byte packets
B	Bottleneck buffer size in bytes

4.4 Parameter Calibration

We describe how to select values for the parameters in Table 1. The parameters U, D, G_{min}, F, and $comp(s)$ are exclusive to CellReplay and are calibrated in every new environment before recording traces. This process is automated. B is a standard network emulation parameter, derived using a classical max-min approach [11]. For details, see §A.3.

Setting U and D. We profile the network to determine a *packet train size* that provides the best overall approximation of the network across other sizes. We first conduct randomized experiments with different packet train sizes (the same as §3.2) using a fixed train gap that is conservatively large enough to ensure the link returns to its light-workload state.

[3]Replaying base RTT changes during the active period could lead to double-counting, as PDO traces already capture some RTT changes in the form of gaps between PDOs. While this might seem counterintuitive to our goal of emulating time-varying base delay, the issue discussed in §3.1 only arises when there is a gap in the workload packets that empties the queue. However, in such cases, CellReplay has an opportunity to re-enter the inactive state and select another base RTT. Although there is a slight chance that the workload gap is shorter than the inactive-state timer F, F is typically small (e.g., 5 milliseconds), so we did not find this to be a major problem in practice.

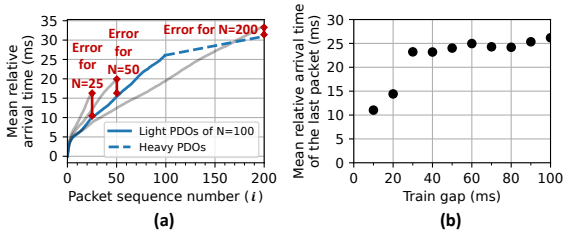


Figure 8: *(a) Estimating error if a 100-sized train is used to estimate {25, 50, 200}-sized trains. (b) G_{min} is chosen as the smallest train gap that results in performance similar to a large train gap (i.e.,* 30 ms*).*

In our implementation, we use a 100 ms gap, and the set of train sizes we consider is $\{5, 25, 50, 75, \ldots, X\}$ where X is chosen such that the resulting mean sending rate (including gaps between trains) is half of the bottleneck throughput.

After running for 10 trials, we compute $R_N(i)$ for each train size, which represents the mean relative arrival time of the i-th packet in an N-packet train. The relative arrival time is the packet's arrival time minus that of the first packet in its train. R_N essentially represents the mean light PDOs of an N-sized train. We further define $R^*_N(i)$ as the estimated mean arrival time of the i-th packet *in replay mode*, assuming we choose to record trains of size N. More specifically, recall that during replay, we follow light PDOs before splicing in the heavy PDOs; therefore, $R^*_N(i) = R_N(i)$ for $i \leq N$, and otherwise, $R^*_N(i) = R_N(N) + heavy(i-N)$, where $heavy(x)$ is the delivery delay of x packets based on the mean throughput of the heavy workload.

The purpose of $R^*_N(i)$ is to help us calculate the estimation error of $R_N(N)$ (i.e., the mean relative arrival time of the *last* packet of an N-packet train) for every other train size N that we have tested. Let L be the train size used to estimate the error for other trains. Fig. 8(a) shows the estimation error when using $L = 100$. The blue line is $R^*_{100}(i)$, which represents the PDOs based on the concatenation of mean light and heavy PDOs. If we use that to estimate $R_N(N)$ of *other* trains $N \in \{25, 50, 200\}$, the prediction will result in some error (shown in red). For each train size L, we compute the mean error over *all* tested train sizes, and our chosen train size is the L that yields the smallest mean error. We conduct the entire procedure in two directions (uplink and downlink) separately to choose the train lengths U and D.

Finding G_{min}. If the train gap G is too small, the link will not have enough time to reset to its light-workload mode before the next train arrives. We typically set $G = 50$ ms for stationary conditions and $G = 100$ ms for mobile conditions. However, in certain environments, this may not be large enough. We thus aim to find G_{min}, the smallest value at which the link has enough time to return to its light-workload mode, ensuring that our chosen G is at least that value.

We conduct another randomized experiment, this time testing different train gaps. Again, we send sequences of trains; however, in this case, we fix the train length at our chosen value (U or D, for uplink or downlink, respectively, in separate experiments) and vary the gap g. We begin with a conservatively large gap (as in the previous experiment) and test gaps of decreasing size; in our implementation, $g \in 100, 90, 80, ..., 10$ ms. Let $r_{last}(g)$ denote the mean relative arrival time of the last packet in trains with gap g. Intuitively, as g decreases to a too-small size, the link will begin staying in its heavy-workload mode, causing $r_{last}(g)$ to decrease. We set G_{min} as the smallest g for which $r_{last}(g)$ is within 20% of its value with the conservatively large gap, i.e., $r_{last}(100ms)$. In the example of Fig. 8(b), CellReplay selects $G_{min} = 30$ ms.

Inferring F. Recall that F determines how long CellReplay's emulated link remains idle in the active state before transitioning back to the inactive state. We derive F using the same data collected to select G_{min}, which involves calculating the difference between G_{min} and the time required for the queue to clear, which is observable as $r_{last}(G_{min})$. For details, see §A.2.

Inferring *comp*(). We profile how RTT is affected by packet size by sending randomly sized packets between {100, 200, ..., 1400} bytes every 50 ms to a receiver that responds with a 100-byte ACK. We then measure the RTT difference for a packet size of s compared to the RTT of 1400-byte packets and model this difference as $comp(s)$. We describe this in more detail in §A.1.

5 Evaluation

Our goal is to evaluate the accuracy of CellReplay's emulation in replicating application performance compared to its live network counterpart. We also compare CellReplay with Mahimahi [25]. We implemented CellReplay record tool in Java and Python 3 to send and receive UDP packets. We extended the Mahimahi shell to support CellReplay replay, allowing unmodified applications to run inside the shell and experience the emulated network conditions induced by CellReplay. For more details, refer to §B.

The evaluation includes experiments that test CellReplay's accuracy across (1) different networked applications, including web browsing and random file transfers using TCP, (2) different cellular providers and technologies, including T-Mobile, Verizon, 5G mid-band, and 5G low-band, and (3) different environmental conditions, such as good signal strength, weak signal strength, crowded areas, and various mobility levels (stationary, walking, and driving). For full details on the environments and their calibration parameters, refer to §C. Finally, we present a use case of using CellReplay and Mahimahi to evaluate ABR algorithms.

5.1 Experimental setup

We designed two test setups: a *live network* and an *emulation* test setup, as shown in Figure 9. The *live network* test setup was used for running application tests on the live network. During the tests, we tethered a laptop to phones connected

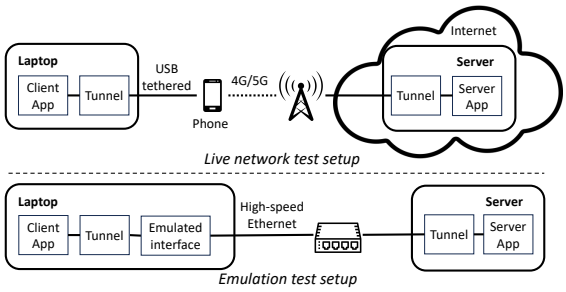


Figure 9: *Live network and emulation application test setup.*

to 5G or 4G networks. The client application (e.g., a web browser) and the application server (e.g., a web server) communicate via a UDP tunnel (based on [42]). We deployed our server (i.e., the remote endpoint) in the same geographical area–within 10 miles of the phones–to minimize the network path length and, consequently, reduce the likelihood of experiencing congestion over long paths. We also used a similar setup, albeit without a tunnel, to record the cellular network traces using UDP traffic. However, instead of a single phone, we used two identical phones to separately perform packet train probing and the Saturator workload.

We used the *emulation* test setup to test applications under an emulated network interface that employed either CellReplay's or Mahimahi's replay approach. Although this setup is similar to the live network test, we made two modifications: (1) we replaced the USB-tethered interface with a direct high-speed Ethernet connection to our server via a single switch, and (2) we ran the tunnel inside either CellReplay's or Mahimahi's replay network-emulation shell, which emulates the network using recorded traces. The same client and server devices were used for record and replay.

End-point specifications. We used two Samsung Galaxy S22 (SGS) and two Google Pixel 5 (Pixel) phones for testing. We had an unlimited data plan from both T-Mobile and Verizon. Since we observed no performance difference between the SGS and Pixel across different operators, we connected our SGS devices to T-Mobile and our Pixel devices to Verizon for convenience. The laptop in our setups featured an Intel Core i7 CPU, 16 GB RAM, and a 512 GB SSD, running Ubuntu 20.04.

5.2 Applications under Test

We tested CellReplay's fidelity using two real-world applications: (a) web-page loads, which exhibit complex traffic patterns and represent the most popular application type on mobile devices, and (b) random file downloads, which have simpler traffic patterns but involve large-sized flows.

Web page load test. Our client ran a Chromium browser to load a web page from our server. To generate the list of web pages for our testing, we began by randomly selecting 200 internal and landing pages from the Hispar list [6]. Using Chromium, we loaded all pages and recorded each page's total size of the compressed web objects. We then sorted the pages based on this total size and selected five pages each for landing and internal pages corresponding to the 10^{th}, 30^{th}, 50^{th}, 70^{th}, and 90^{th} percentiles of the distribution. We use 'L-ID' and 'I-ID' to refer to an individual landing and internal page, respectively. 'ID' is a number indicating the order based on page size, where lower values represent smaller page sizes. Refer to §D for the exact list of pages and their page composition).

We utilized Mahimahi's HTTP record-and-replay framework [25] to replay these pages, ensuring that the fetched content for a given page remained consistent across all trials. First, we recorded all HTTP requests and responses using `mitmproxy` [2], following Mahimahi's record format, while loading a web page using a headless Chromium browser. Next, we used Mahimahi's ReplayShell to serve the responses over the HTTP/1.1 (using Apache2) or HTTP/2 (using h2o from the [47] extension).

In our test setup (Fig. 9), ReplayShell was deployed on our server, while the laptop ran a headless Chromium browser to repeatedly fetch a web page. We cleared the browser and DNS caches before every web-page fetch, and we used TCP Cubic, the default TCP implementation on Linux, for this test. Lastly, we measured *page-load time (PLT)*, based on the `onLoad` event [29], as the application performance metric.

Random file download test. For this test, we implemented a client that sends download requests (each fewer than 10 bytes) to a server. Each request selects a file uniformly at random from a list of files of varying sizes, and the server delivers the corresponding file data. After completing the transfer, the client then sleeps for 50 ms before requesting another file. We implemented the client and server in Python and used TCP Sockets. We measured the *file download time* as the turnaround time between when the client sends its request and when it receives the requested file.

Adaptive video streaming over HTTP. We ran a client-server setup from [4, 20], which includes multiple adaptive bitrate (ABR) streaming implementations. However, instead of using the 2K video from that setup, our server hosts a 250-second-long AVC 4K video (`indoor soccer` from [35]) with a 4-second chunk duration encoded with 12 bitrates of [100, 200, 375, 550, 750, 1000, 1500, 3000, 5800, 7500, 12000, 17000]. We used a Chromium browser to run the video player.

5.3 Methodology

Randomized trials. When evaluating a networked system's performance on the CellReplay emulated interface versus the live network, we must account for cellular network variability. Thus, all our experiments used *randomized trials*. Each experiment comprised multiple trials, with each trial consisting of multiple tests and recording sessions. A test involved running an application (e.g., loading a web page) on a live network, while a recording session (e.g., CellReplay record) gathered a trace of the live network. The tests and recording sessions ran for the same duration, and we randomized the sequence

of tests and sessions within each trial. We then compared the application's performance (across multiple tests) on the live network to that on the emulated network to calculate the emulation accuracy.

Calculating the emulation accuracy. We quantified emulation error using the normalized difference between two distributions of application performance: one from the live network and the other from the emulation. To measure the difference between these distributions, we used Earth Mover's Distance (EMD) [33], defined as: $EMD(L,T) = \int_{-\infty}^{+\infty} |L(x) - T(x)|\,dx$, where L and T are the cumulative distribution functions (CDFs) of the observed application performance on the live and emulated networks, respectively. A lower EMD value indicates a high-fidelity emulation, meaning the performance distributions are more similar. Finally, we calculated the *emulation distribution error* by dividing the EMD with the mean (performance) value from the live network tests.

Other details. In each new network environment (or network operator), we first performed a calibration. To record CellReplay traces, we used one phone (*primary*) for packet train probing and the other (*secondary*) for running the Saturator. The *primary* phone was also used for calibration, testing applications on the live network, and recording Mahimahi traces using our own Saturator. For propagation delay, we provided Mahimahi with the minimum base delay observed from the packet train workload. During data collection, both phones (UEs) were held together at a close distance ($\approx$ 12 inches). We also confirmed that both probing devices were attached to the same cell in all tests, except in the driving case.

5.4 Microbenchmarks

We first evaluated how accurately CellReplay emulates the time-varying base RTT and the non-uniform delivery rate in a cellular network. We conducted packet RTT and packet train tests (refer to §3.1 and §3.2) as separate experiments. Each experiment consisted of 30 randomized trials, with each trial including two record sessions (CellReplay and Mahimahi) alongside packet RTT and packet train tests. The sessions (as well as tests) lasted for 10 s. All tests were performed under "good" network conditions, i.e., UEs were held stationary near a window.

First, CellReplay accurately records RTT changes over time, with the CDF of packet RTTs closely overlapping with the live network (Figure 10). As expected, Mahimahi persistently underestimates RTT (e.g., median RTT is underestimated by 16.88% on T-Mobile and 13.25% on Verizon) and produces an RTT distribution that deviates from the live network. Second, CellReplay captures the non-linearity in train completion times as train size increases more accurately than Mahimahi, thanks to its light and heavy PDOs approach. The packet train experiment demonstrates CellReplay 's interpolation error when using a pre-defined train size (e.g., 75 packets for T-Mobile) for workloads ranging from 1 to 200 packets. For trains longer than 75 packets on T-Mobile and 100 packets on Verizon, CellReplay achieves lower interpolation errors than Mahimahi's single (heavy) PDO approach, reducing errors from 26.68% to 6.44% on T-Mobile and from 43.24% to 7.74% on Verizon for a train size of 200. CellReplay 's interpolation errors for longer trains highlight an opportunity for future improvement.

When running these experiments, the primary phone recorded the base delay while the secondary continuously sent packets; the results, hence, suggest that the two-phone setup had minimal interference.

5.5 Web browsing test

We evaluated CellReplay's accuracy for the web browsing by fetching both landing and internal pages (listed in Table 3) using HTTP/1.1 and HTTP/2. As before, we conducted these tests under "good" network conditions, with the UEs placed near a window. For each operator, we conducted four experiments, with each randomized experiment consisting of 10 trials. Each trial included fetching five pages using the live network and two recording sessions–one for CellReplay and the other for Mahimahi. Each test and recording session lasted for 60 s. In total, we spent approximately 9.33 h conducting these experiments.

CellReplay outperforms Mahimahi across all page load tests (Figure 11). It achieves an emulation distribution error between 1.2%-17.7%, with a mean error of 6.7%. In contrast, Mahimahi has errors ranging from 4.5% to 42.6%, with a mean error of 17.1%. On average, CellReplay reduces emulation error by 60.8% for web-browsing apps compared to Mahimahi. Page load traffic is typically dominated by small object transfers, which are sensitive to RTT and categorized as *light* traffic. Mahimahi's underestimation of RTT and overestimation of PDOs from the Saturator lead to significant errors for small flows (see §5.6). All Mahimahi's errors stem from underestimating *all* PLTs, with a mean PLT of 2637 ms compared to 2918 ms in real networks. Meanwhile, CellReplay accurately captures the network RTT and provides a better representation of real available bandwidth for these transfers, although some error still persists.

Below, we examine two dimensions to highlight CellReplay's edge over Mahimahi.

Across different operators. CellReplay maintains low mean errors for both operators: 6.4% on T-Mobile and 7.1% on Verizon. Mahimahi has a mean error of 13.2% on T-Mobile and an even higher 21% on Verizon. Mahimahi's poor performance on Verizon is expected: Verizon tends to assign small bandwidth for small file transfers (see §5.6), and page loads primarily involve fetching small objects.

Across different protocols. CellReplay performs well on both HTTP/1.1 and HTTP/2, with mean errors of 5.8% and 7.7%, respectively In contrast, Mahimahi shows poorer accuracy, with a mean error of 12.6% on HTTP/1.1 and an even higher 21.6% on HTTP/2. The multiplexing behav-

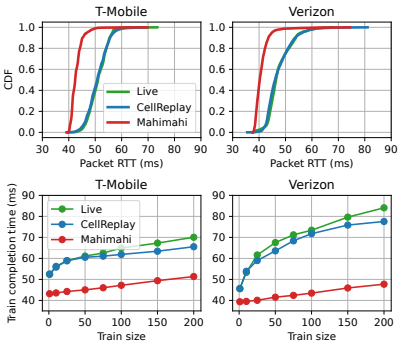


Figure 10: *The CDF of RTTs (above) and mean TCTs (below) from the packet RTT and packet train tests, respectively, run on both live network and emulations.*

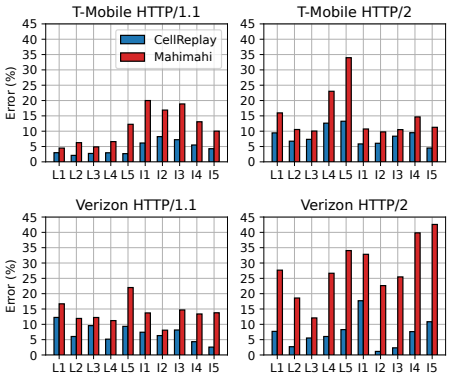


Figure 11: *Emulation distribution error across different web-page load tests with HTTP/1.1 and HTTP/2.*

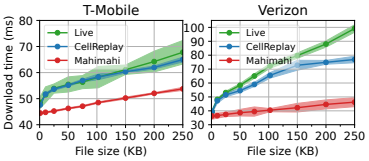


Figure 12: *Mean small file download times along with its 95% CI.*

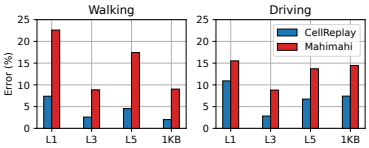


Figure 13: *Emulation distribution error of three web PLTs and 1KB file download time under mobility.*

ior of HTTP/2 may lead to a more complex traffic pattern compared to HTTP/1.1, leading to more frequent medium-sized flows. As a result, CellReplay often uses an interpolated PDO, increasing the error. For Mahimahi, the issue is even more pronounced, as it always applies the heavy rate, significantly underestimating flow completion times. Consequently, both methods experience higher errors for HTTP/2 than for HTTP/1.1.

5.6 Random file downloads test

To further evaluate CellReplay's accuracy, we conducted randomized file download tests with small-sized files (1 KB to 250 KB) and medium-sized files (1 MB and 10 MB) on both T-Mobile and Verizon. Unlike the web-page loads, download tests are not affected by non-network-related computations (e.g., JavaScript parsing). Additionally, unlike the packet train test, which sends fixed train sizes over time, the download test is fully random: the client selects each file randomly from a predefined list. This live experiment consisted of 20 randomized trials, each including: two test workloads (small and medium file downloads) and two recording sessions (Mahimahi and CellReplay). Each test and recording session ran for 60 s, resulting in a total experiment duration of 80 min per network operator. All tests were conducted under "good" network conditions.

Small-sized files. Figure 12 shows the file download times for sizes between 1 KB and 250 KB. Consistent with the observation in §5.4, download times do not conform to a straight line. Mahimahi significantly underestimates (mean) download times, with errors ranging from 8.4% to 20.7% on T-Mobile and 7.9% to 49% on Verizon. CellReplay manages to capture and emulate the non-uniform bandwidth availability with its light and heavy PDOs, resulting in significantly lower mean download time errors: 0.5%-3.5% on T-Mobile and 0.2%-22.4% on Verizon. CellReplay 's error increases with larger file sizes (e.g., 250 KB on Verizon) as it must interpolate once packet sequence lengths exceed a certain threshold.

Medium-sized files. CellReplay achieves a mean distribution error of 9.14% for 1 MB downloads and 6.54% for 10 MB downloads across both providers. In comparison, Mahimahi produces significantly higher errors of 23.35% and 17.06%, respectively, for the same file sizes. Surprisingly, Mahimahi's error increases further for 10 MB files, despite expectations that large file transfers would be dominated by heavy PDOs (i.e., the Saturator rate). We suspect that Mahimahi's inaccuracy stems from a combination of base RTT underestimation and available bandwidth overestimation. Refer to §E.3 for the full results, including mean file download times for 1 MB and 10 MB files and their respective distribution emulation errors.

5.7 Interpolation effectiveness

We evaluated the impact of CellReplay's PDO interpolation on emulation accuracy by comparing it to two variants: CellReplay with only light PDOs (CellReplay-light) and CellReplay with only heavy PDOs (CellReplay-heavy) for web page loading and file downloads. CellReplay-heavy resembles Mahimahi but incorporates variable base delay and delay compensation based on packet size. CellReplay-light does not transition to heavy PDOs when the light PDOs end; instead, it restarts from the beginning. We used T-Mobile and Verizon traces, as in §5.5 and §5.6, and reported the average error across both operators.

Figure 14 presents the results. For web browsing, CellReplay-light performs similarly to CellReplay with interpolation (i.e, our CellReplay) and beats CellReplay-heavy, as web browsing is a mostly light workload. For the 1 KB download, all three versions perform similarly since RTT dominates performance and bandwidth is less critical. As the workload increases to 10 KB, 100 KB, and 1 MB, bandwidth becomes a more significant component of download time, but CellReplay-heavy's emulation of bandwidth is inaccurate in this regime, leading it to have higher errors. However, at 10 MB, CellReplay-heavy's error decreases as the workload starts to resemble the Saturator. Meanwhile, CellReplay-light excels for smaller downloads (10 KB and 100 KB) but struggles with larger files (1 MB and 10 MB). This suggests that

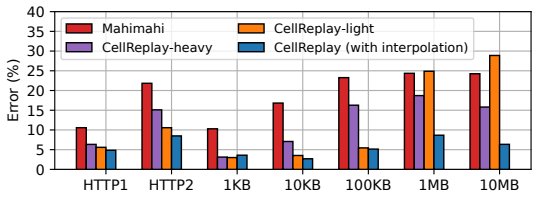


Figure 14: *Emulation distribution error of CellReplay with and without light and heavy PDOs interpolation.*

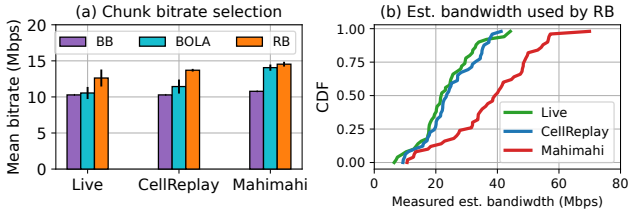


Figure 15: *(a) Mean ABR bitrate selection result. Mahimahi shows positive bias to BOLA. (b) Reported estimated bandwidth from RB.*

neither light nor heavy PDOs alone can accurately emulate the characteristics of the wireless channel.

Notably, the variable base delay and delay compensation in CellReplay-heavy reduce error to 11.67%, compared to Mahimahi's 18.77%. The error is further reduced by incorporating PDO interpolation, as in our full CellReplay system, bringing the error down to 5.68%. These results confirm that interpolating between light and heavy PDOs significantly improves emulation accuracy.

5.8 Mobility and other network conditions

We also evaluated CellReplay's accuracy under low and moderate mobility. For low mobility, a user walked through an office corridor in a loop while carrying the UE (connected to Verizon). For moderate mobility, the user drove around a university campus in a loop with the UE (connected to T-Mobile). We followed a fixed path while walking or driving in a loop to ensure consistent comparisons across trials. On average, a loop took 75 s while walking and 220 s while driving.

To reduce the number of tests, we performed web page load tests using only three HTTP/1.1 landing pages – L1, L3, and L5, which represent small, medium, and large web pages (§5.2), respectively. We also tested repeated 1 KB file downloads. To minimize variance, we limited each trial to two live tests and two recordings. We conducted two separate experiments: the first, with 1 KB file download and L3, and the second, with L1 and L5. We conducted 10 trials for `walking` experiments and 5 trials for `driving` experiments. Calibration was performed only once at the starting point. In total, we spent 1.67 h walking and 2.45 h driving around the loop.

Figure 13 shows the emulation distribution error for CellReplay and Mahimahi. Network conditions are more variable under `driving`, which may also introduce packet drop, a factor that CellReplay does not capture[4]. We indeed confirmed that `driving` triggered at least two handovers[5], and a few packets were dropped during certain periods of recording. Consequently, CellReplay's accuracy suffers more under `driving` than `walking`. However, despite this limitations, CellReplay still provides a noticeable improvement (1.8 times) over Mahimahi in both mobility scenarios. This improvement stems from CellReplay's ability to capture more performance variability than Mahimahi, as indicated by the shape of the CDF curve, which is closer to the live network's curve compared to Mahimahi. Refer to §E.2 for the CDF curve for 1KB download and L3 page load times.

Additionally, we tested CellReplay in a basement and a crowded library. In both conditions, CellReplay produced significantly lower error compared to Mahimahi, with an emulation distribution error of 5.74% (vs. 15.22%) in the basement and 8.47% (vs. 22.51%) in the crowded library. Refer to E.1 for more details.

5.9 Use case: evaluating ABR algorithms

Finally, we demonstrate a common application for record-and-replay emulation: evaluating ABR algorithms for 4K video streaming. ABR algorithms are reactive, meaning their bitrate selection for video chunk downloads can be influenced directly (rate-based) or indirectly (buffer-based) by observed network performance. Thus, accurately emulating network conditions is crucial to avoid bias in algorithm evaluation.

We compared three ABR algorithms[6] from [4] running on Verizon 5G under "good" network conditions using CellReplay and Mahimahi emulation: (1) Buffer-based (BB) and (2) BOLA, which make bitrate decisions based on buffer occupancy, and (3) Rate-based (RB), which estimates throughput from past chunk download times to select the next bitrate. Given the high download bandwidth reported by Saturator (270 Mbps), all three algorithms eventually select the highest bitrate (17 Mbps) and do not experience rebuffering. However, their startup phase [17] behavior differs, so we focused on bitrate selection for the first 10 chunks as our QoE metric. We conducted 10 randomized trials, each with three test workloads (streaming video using three ABRs) and two recording sessions (Mahimahi and CellReplay). Each test and recording session lasted for 30 s.

Figure 15a presents the mean bitrate results. As expected, Mahimahi reports a higher bitrate than Live-5G across all three ABRs, with an average overestimation of 17.73%, compared to 5.89% on CellReplay. This aligns with our findings that Mahimahi tends to overestimate network performance, leading to inflated application performance. But, more importantly, this also affects protocol evaluation. Mahimahi's

[4]This refers to the drop in IP packets. CellReplay can still capture the effects of the handover process through the increase in the base latency and PDO blackout and emulate it accordingly.

[5]There were a brief period (< 1 second) when the two devices were connected to different base stations due to the handovers were not done at the same time.

[6]This setup also includes MPC [44] and Pensieve [20], but these were not tested, as MPC requires modifying the hardcoded MPC table, and Pensieve requires retraining.

results suggest that RB and BOLA are the best protocols, with BOLA significantly outperforming BB by 30.43%. However, in reality in this environment, BOLA performs significantly worse than RB and is similar to BB. CellReplay 's results are much more aligned with the live network. This discrepancy is largely due to Mahimahi's overestimation of available bandwidth, particularly for smaller initial chunks, causing ABRs to quickly converge to the highest bitrate. In Figure 15b, we present the measured bandwidth estimation from RB, which shows that Mahimahi overestimates bandwidth by 44.38% in the median, compared to 7.96% on CellReplay. Unlike Mahimahi, CellReplay does not suffer from this bias and provides more accurate relative performance of ABRs due to its dual light and heavy rate approach.

6 Discussion and future work

We discuss CellReplay's use cases, limitations, and plans for future improvements.

Use cases. As shown in §5.9, CellReplay can be used to evaluate new applications and protocols on cellular networks and provide more accurate emulation of real network performance compared to the state-of-the-art approach. CellReplay is superior for latency-sensitive applications, as it can emulate base delay variability, and applications with variable flow sizes, as it can emulate the bandwidth-workload dependency. Adaptive applications (e.g., ABR) that react to network measurement will also receive more accurate performance results with CellReplay.

We also provide traces that researchers and developers can use for testing on CellReplay. While recording traces with CellReplay requires a bit more effort than Mahimahi due to the use of two phones, this is a minor issue, as once a diverse set of traces is recorded, users can easily replay them without the phones (just as in Mahimahi). Since CellReplay's implementation is based on Mahimahi's shell, unmodified applications can easily use CellReplay emulated interface.

Inaccuracies in CellReplay. While we have made significant progress in faithfully replaying cellular performance, CellReplay involves several simplifications and assumptions: (1) CellReplay does not record and replay random packet losses (although it drops packets when the queue overflows and can be manually configured for a set random drop rate). We notice, however, that cellular links under stationary conditions are robust to random packet drops (e.g., due to packet corruption) due to link-layer retransmission [22]. However, packet drops are more frequent during handovers in mobility [15]. (2) CellReplay uses fixed calibration parameters before each recording session. A more adaptive selection of parameters could help when network conditions change during recording. (3) CellReplay's two-phone setup has some weaknesses. Under mobility, both phones may connect to different base stations and report different performances. Moreover, although we did not observe major interference, greater interference may occur with other providers and conditions. In the future, we will be able to use a single phone with Dual-SIM Dual-Active (DSDA) modem [1] for recording, as DSDA allows simultaneous traffic transmission across two SIMs. Each of these areas represents an opportunity for future improvement. We note, however, that our evaluation results account for the errors caused by these inaccuracies.

Improving CellReplay interpolation accuracy. A straightforward approach is to gather additional data points for interpolation. However, since we need to record data simultaneously (e.g., while walking or driving), we are limited to running only a few workloads with a few phones. Another viable approach is to leverage ML to model complex, workload-dependent network performance and providers' resource allocation policies. We can train an ML model based on recorded workload and performance traces to predict network performance (e.g., PDOs) for a given test workload. However, this approach may require extensive data collection to capture RAN scheduling behavior, increasing the recording effort and making it time-consuming.

Adding more cellular network specific features. CellReplay could be improved by explicitly emulating cellular network-specific features, especially those that affect application performance. These include radio resource control (RRC) delays, handover, and other relevant factors.

Other limitations. CellReplay probes UDP traffic to record network traces, meaning it cannot capture the effects of network discrimination based on IP protocol types, such as from TCP middlebox intervention [8].

7 Conclusion

This paper exposes the difficulty of accurate record-and-replay emulation and presents CellReplay, which more faithfully captures real-world cellular network performance characteristics. CellReplay's approach of dual-workload recording and interpolated replay provides the community with a more accurate platform for evaluating research in cellular environments. We also hope this work inspires the community to explore future designs that can make record-and-replay emulation even more faithful to live deployments.

Acknowledgements: We sincerely thank our shepherd, Kyle Jamieson, and the anonymous reviewers for their valuable suggestions. We also thank Keith Winstein for helpful discussions on Mahimahi and CellSim. Additionally, we thank Qinjun Jiang, Sam Yuan, and Pradnyan Khodke, who helped drive the car while the author conducted experiments. This project was supported by gifts from T-Mobile and Cisco, and a grant from the IBM-Illinois Discovery Accelerator Institute.

References

[1] Two birds, one stone: Unleashing the full potential for simultaneous 5g cellular connections, thanks to our new qualcomm dsda gen 2 with dual data.

https://www.qualcomm.com/news/onq/2023/05/unleashing-full-potential-for-simultaneous-5g-cellular-connections-qualcomm-dsda-gen-2-with-dual-data, 2023. [Last accessed on Feb 17, 2025].

[2] mitmproxy. https://mitmproxy.org, 2024. [Last accessed on May 8, 2024].

[3] Network features reference. https://developer.chrome.com/docs/devtools/network/reference#throttling-profile, 2024. [Last accessed on May 8, 2024].

[4] Pensieve. https://github.com/hongzimao/pensieve, 2024. [Last accessed on Sept 17, 2024].

[5] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Wanna make your tcp scheme great for cellular networks? let machines do it for you! *IEEE Journal on Selected Areas in Communications*, 39(1):265–279, 2020.

[6] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M Maggs. On landing and internal web pages: The strange case of jekyll and hyde in web performance measurement. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[7] Sachin Ashok, Shubham Tiwari, Nagarajan Natarajan, Venkata N Padmanabhan, and Sundararajan Sellamanickam. Data-driven network path simulation with ibox. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–26, 2022.

[8] Arjun Balasingam, Manu Bansal, Rakesh Misra, Kanthi Nagaraj, Rahul Tandra, Sachin Katti, and Aaron Schulman. Detecting if lte is the bottleneck with bursttracker. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2019.

[9] Francesco Capozzi, Giuseppe Piro, Luigi Alfredo Grieco, Gennaro Boggia, and Pietro Camarda. Downlink packet scheduling in lte cellular networks: Key design issues and a survey. *IEEE communications surveys & tutorials*, 15(2):678–700, 2012.

[10] Stanley CF Chan, KM Chan, Ke Liu, and Jack YB Lee. On queue length and link buffer size estimation in 3g/4g mobile data networks. *IEEE Transactions on Mobile Computing*, 13(6):1298–1311, 2013.

[11] Mark Claypool, Robert Kinicki, Mingzhe Li, James Nichols, and Huahui Wu. Inferring queue sizes in access networks by active measurement. In *Passive and Active Network Measurement: 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004. Proceedings 5*, pages 227–236. Springer, 2004.

[12] Rostand AK Fezeu, Eman Ramadan, Wei Ye, Benjamin Minneci, Jack Xie, Arvind Narayanan, Ahmad Hassan, Feng Qian, Zhi-Li Zhang, Jaideep Chandrashekar, et al. An in-depth measurement analysis of 5g mmwave phy latency and its impact on end-to-end delay. In *International Conference on Passive and Active Network Measurement*, pages 284–312. Springer, 2023.

[13] Moinak Ghoshal, Imran Khan, Z Jonny Kong, Phuc Dinh, Jiayi Meng, Y Charlie Hu, and Dimitrios Koutsonikolas. Performance of cellular networks on the wheels. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, pages 678–695, 2023.

[14] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. {ABC}: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, 2020.

[15] Ahmad Hassan, Arvind Narayanan, Anlan Zhang, Wei Ye, Ruiyang Zhu, Shuowei Jin, Jason Carpenter, Z Morley Mao, Feng Qian, and Zhi-Li Zhang. Vivisecting mobility management in 5g cellular networks. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 86–100, 2022.

[16] Stephen Hemminger et al. Network emulation with netem. In *Linux conf au*, volume 5, page 2005, 2005.

[17] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198, 2014.

[18] Nikhil Kansal, Murali Ramanujam, and Ravi Netravali. Alohamora: Reviving HTTP/2 push and preload by adapting policies on the fly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 269–287. USENIX Association, April 2021.

[19] Gerui Lv, Qinghua Wu, Yanmei Liu, Zhenyu Li, Qingyue Tan, Furong Yang, Wentao Chen, Yunfei Ma, Hongyu Guo, Ying Chen, et al. Chorus: Coordinating mobile multipath scheduling and adaptive video streaming. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 246–262, 2024.

[20] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[21] Abhishek Kumar Mishra, Sara Ayoubi, Giulio Grassi, and Renata Teixeira. Nemfi: Record-and-replay to emulate wifi. *ACM SIGCOMM Computer Communication Review*, 51(3):2–8, 2021.

[22] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, pages 894–905, 2020.

[23] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.

[24] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. Vesper: Measuring Time-to-Interactivity for web pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 217–231, Renton, WA, April 2018. USENIX Association.

[25] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[26] Yunzhe Ni, Zhilong Zheng, Xianshang Lin, Fengyu Gao, Xuan Zeng, Yirui Liu, Tao Xu, Hua Wang, Zhidong Zhang, Senlang Du, et al. Cellfusion: Multipath vehicle-to-cloud video streaming with network coding in the wild. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 668–683, 2023.

[27] Brian D Noble, Mahadev Satyanarayanan, Giao T Nguyen, and Randy H Katz. Trace-based mobile network emulation. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 51–61, 1997.

[28] NS-3. NS-3 Network Simulator. https://www.nsnam.org/, 2023. [Last accessed: September 20, 2023].

[29] Jan Odvarko. Har 1.2 spec, 2007.

[30] Murali Ramanujam, Helen Chen, Shaghayegh Mardani, and Ravi Netravali. Floo: automatic, lightweight memoization for faster mobile apps. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 168–182, 2022.

[31] Devdeep Ray, Jack Kosaian, KV Rashmi, and Srinivasan Seshan. Vantage: optimizing video upload for time-shifted viewing of social live streams. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 380–393. 2019.

[32] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[33] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. A metric for distributions with applications to image databases. In *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*, pages 59–66. IEEE, 1998.

[34] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, and Bruce Maggs. DChannel: Accelerating mobile applications with parallel high-bandwidth and low-latency channels. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 419–436, Boston, MA, April 2023. USENIX Association.

[35] Babak Taraghi, Hadi Amirpour, and Christian Timmerer. Multi-codec ultra high definition 8k mpeg-dash dataset. In *Proceedings of the 13th ACM Multimedia Systems Conference*, pages 216–220, 2022.

[36] Bo Wang, Mingwei Xu, Fengyuan Ren, and Jianping Wu. Improving robustness of dash against unpredictable network variations. *IEEE Transactions on Multimedia*, 24:323–337, 2021.

[37] Shibo Wang, Shusen Yang, Hailiang Li, Xiaodan Zhang, Chen Zhou, Chenren Xu, Feng Qian, Nanbin Wang, and Zongben Xu. Salientvr: saliency-driven mobile 360-degree video streaming with gaze information. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 542–555, 2022.

[38] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, Lombard, IL, April 2013. USENIX Association.

[39] Yaxiong Xie and Kyle Jamieson. Ng-scope: Fine-grained telemetry for nextg cellular networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–26, 2022.

[40] Yaxiong Xie, Fan Yi, and Kyle Jamieson. Pbe-cc: Congestion control via endpoint-centric, physical-layer bandwidth measurements. In *Proceedings of the Annual*

conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pages 451–464, 2020.

[41] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[42] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.

[43] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, 2018. using network trace for video.

[44] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[45] Danfu Yuan, Yuanhong Zhang, Weizhan Zhang, Xuncheng Liu, Haipeng Du, and Qinghua Zheng. Prior: deep reinforced adaptive video streaming with attention-based throughput prediction. In *Proceedings of the 32nd Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 36–42, 2022.

[46] Bo Zhang, Thiago Teixeira, and Yuriy Reznik. Performance of low-latency http-based streaming players. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 356–362, 2021.

[47] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, and Klaus Wehrle. Is the web ready for http/2 server push? In *Proceedings of the 14th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2018.

A Calibration details

A.1 Inferring $comp()$: derivation

We examine the impact of packet size on RTT by sending packets with sizes sampled uniformly at random from {100, 200, ..., 1400} bytes every 50 ms to a receiver that replies to each with a 100-byte ACK. After running for a configurable time (5 minutes for our case), we calculated the mean RTT for each packet size x, denoted as μ_x. The compensation delay for a packet size s, or $comp(s)$, is computed as follows. If $s < 100$, $comp(s) = \mu_{100} - \mu_{1400}$. Otherwise, $comp(s) = (1-\alpha)\mu_i + \alpha\mu_j - \mu_{1400}$ where i and j are the tested packet sizes immediately less than and greater than s, respectively. The parameter α is set to linearly interpolate between the two observed means, i.e., $\alpha = (s-i)/(j-i)$. The compensation is always relative to 1400-byte packets because that is the size used to measure base RTT. This procedure is repeated separately for uplink and downlink measurements, with sender and receiver roles swapped.

A.2 Inferring F: derivation

Assume that internally, the bottleneck link remains in its heavy-workload state while it has queued packets to send, and then once it remains idle for some time F, it returns to the light-workload state. With the gap of G_{min}, this state change is just barely reached. Thus, we have $S + G_{min} \approx Q + F$, where S is the time the sender takes to send the train, G_{min} is the time it waits before beginning the next train, and Q is the time for the link to entirely clear the packet train out of its queue. Since S and G_{min} are both timed in user space, S is close to zero. Furthermore, because we have assumed the link of interest is the bottleneck, Q is observable at the receiver as the time between receiving the first and last packets of the train, i.e., $r_{last}(G_{min})$. Thus, we can estimate $F = G_{min} - r_{last}(G_{min})$. In Figure 8b, the G_{min} is 30 ms, and its $r_{last}(g_{min})$ is 23.23 ms. Thus, we compute $F = 6.77$ ms (and round it to 7 ms when setting the parameter).

A.3 Inferring B.

We use a classical max-min approach [11] to infer the bottleneck buffer size using the difference between the minimum and maximum RTT of packets under heavy load. Specifically, we run iperf and monitor it with tcpdump. The buffer size is calculated as $(RTT_{max} - RTT_{min}) \cdot C$, where C is the observed link capacity [10].

B CellReplay implementation

CellReplay's record client is implemented in Java to ease porting into Android. Meanwhile, the record server is implemented in Python 3. The client accepts workload configurations as user inputs, which are determined from the automated calibration performed separately. These configurations are then sent to the CellReplay server. For packet train workloads, the user must input: the number of packets per uplink and downlink train, and the gap between trains. Saturator workload requires the maximum upload and download bandwidth, which can be determined using other bandwidth probing tools (e.g., speedtest.com or iperf). Both client and server send 1400-byte UDP packets via a socket.

CellReplay's replay is built on top of the Mahimahi shell, extending all of Mahimahi's core functionality. This includes

Table 2: *All tested conditions that includes stationary, walking, and driving scenarios. We tested T-Mobile (TM) and Verizon (VZ) networks under 5G mid band (MB) and low band (LB). We also showed the used CellReplay configurations.*

Name	Description	Op.	Net. type	CellReplay config
stationary-good	UEs were in an office near a window	TM	5G MB	U=25, D=75, G=50, F=5
		VZ	5G LB	U=10, D=100, G=50 F=7
stationary-crowded	UEs were in a crowded library during rush hours	VZ	5G LB	U=25, D=100, G=50, F=7
stationary-weak	UEs were in a basement of a building with no window	TM	5G LB	U=25, D=75, G=50, F=5
walking	User was walking through an office corridor in a loop while holding the UEs	VZ	5G LB	U=10, D=50, G=100, F=7
driving	UEs was with the user driving in a loop around the university area	TM	5G MB	U=10, D=75, G=100, F=5

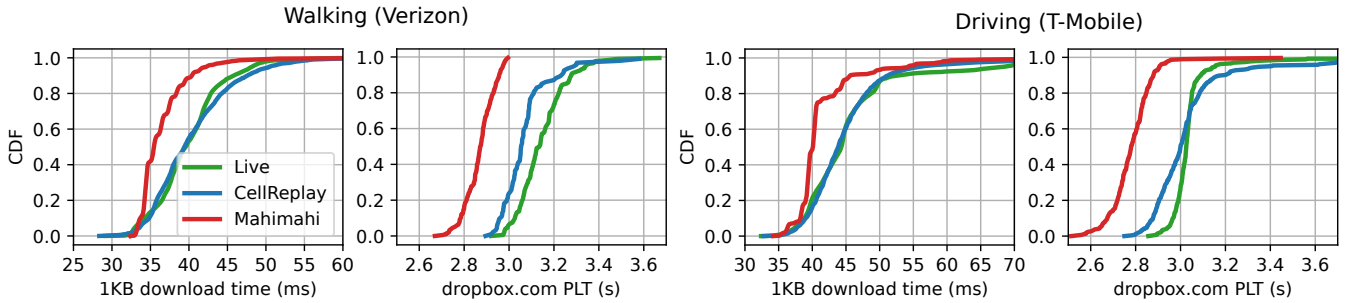


Figure 16: *CellReplay manages to capture more application performance variability under mobility compared to Mahimahi. We cut the CDF graph as the tail is too long.*

the ability to run unmodified applications, or the option to nest the shell with another Mahimahi shell, such as the HTTP ReplayShell. The CellReplay shell accepts the base delay, light PDO trace, and heavy PDO trace as inputs. Similar to the Mahimahi network emulator shell, CellReplay controls a virtual network device (TUN) and captures all IP datagrams from an unmodified application running inside the shell. It then delays each packet before sending it to another interface, such as loopback or Ethernet.

C Experimental conditions

The environments used for evaluation are listed in Table 2.

D Tested web pages list

The list of web pages used for testing and its details on page composition is detailed in Table 3

E More evaluation results

E.1 Non-ideal network conditions

We evaluated CellReplay's accuracy under two challenging network conditions. In this first condition, labeled as '`stationary-weak`', UEs connected to T-Mobile were placed in a windowless basement inside a building. In the second condition, labeled as '`stationary-crowded`', we placed the UEs (connected to Verizon) in a library during crowded or rush hours. Both web-page loads and file downloads were tested under these conditions. To reduce the number of tests and minimize variance, we only selected pages corresponding

Table 3: *List of web pages used for testing along with details on the page composition, including the number of objects ("#objs."), and the mean ("avg. sz.") and total ("tot. sz.") compressed object sizes (both in KB). "PT" indicates page type.*

PT	*ID*	*URL*	*#objs.*	*avg. sz.*	*tot. sz.*
Landing	L1	bing.com	2	205.53	411.06
	L2	microsoft.com	36	24.66	887.64
	L3	dropbox.com	76	22.15	1683.49
	L4	glassdoor.com	64	43.23	2766.46
	L5	discord.com	37	172.86	6395.86
Internal	I1	en.wikipedia.org/wiki/Naivety	19	13.51	256.64
	I2	box.com/about-us	48	14.39	690.95
	I3	etsy.com/payments	50	28.92	1446.22
	I4	youtube.com/user/ESPN	44	56.88	2502.53
	I5	colubrina.tumblr.com	69	116.57	8043.10

to the 10th, 50th, and 90th percentiles of the HTTP/1.1 landing pages. We tested random file downloads with sizes ranging from 1 KB to 10 MB. The experiment was split into two parts: one for web-page loads and the other for file downloads. Each part consisted of 10 randomized trials. In the first part, each trial included loading three pages and two record sessions, while the second part had one random file download test and two record sessions per trial. Every test and record session lasted 60 s. The total experiment time under each network condition was 80 minutes.

Per Figure 17, across the two network conditions and

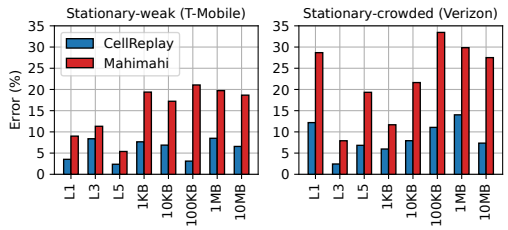


Figure 17: *The web PLT and file download emulation distribution error when testing under non-ideal stationary cases.*

Table 4: *Mean file download time (in ms) of medium-sized files with its emulation distribution error.*

		1MB	***10MB***
T-Mobile	Live	142.52	1056
	CellReplay	135.88 (7.82%)	978.18 (8.19%)
	Mahimahi	117.68 (20.17%)	847.45 (20.44%)
Verizon	Live	155.18	1069.25
	CellReplay	135.68 (10.46%)	974.98 (4.88%)
	Mahimahi	106.63 (26.5%)	762.3 (23.67%)

providers, CellReplay has an average distribution error of 5.96% for web-page loads and 7.9% for file downloads. Mahimahi, in contrast, has mean distribution errors of 13.63% and 22.02% for the same applications, respectively. Even under challenging conditions (`stationary-weak` and `stationary-crowded`), CellReplay still offers a respectable error rate and outperforms Mahimahi.

E.2 Experiments under mobility

Figure 16 demonstrates CellReplay's ability to capture application performance under mobility scenarios more accurately than Mahimahi.

E.3 Medium-sized file download test

Table 4 shows the mean file download time alongside its distribution emulation error for CellReplay and Mahimahi, compared to the live networks.

F Ethics

We note that CellReplay cannot be used to collect other users' packets; it only collects traces of its own user. Moreover, traces collected by CellReplay do not contain any private information. Hence, we believe our work does not raise any ethical concerns.

Large Network UWB Localization: Algorithms and Implementation

Nakul Garg[1], *Irtaza Shahid*[1], *Ramanujan K Sheshadri*[2], *Karthikeyan Sundaresan*[3], *Nirupam Roy*[1]
[1]*University of Maryland, College Park,* [2]*Nokia Bell Labs,* [3]*Georgia Institute of Technology*

Abstract

Localization of networked nodes is an essential problem in emerging applications, including first-responder navigation, automated manufacturing lines, vehicular and drone navigation, asset tracking, Internet of Things, and 5G communication networks. In this paper, we present *Locate3D*, a novel system for peer-to-peer node localization and orientation estimation in large networks. Unlike traditional range-only methods, *Locate3D* introduces angle-of-arrival (AoA) data as an added network topology constraint. The system solves three key challenges: it uses angles to reduce the number of measurements required by 4× and jointly uses range and angle data for location estimation. We develop a spanning-tree approach for fast location updates, and to ensure the output graphs are rigid and uniquely realizable, even in occluded or weakly connected areas. *Locate3D* cuts down latency by up to 75% without compromising accuracy, surpassing standard range-only solutions. It has a 0.86 meter median localization error for building-scale multi-floor networks (32 nodes, 0 anchors) and 12.09 meters for large-scale networks (100,000 nodes, 15 anchors).

1 Introduction

A swarm of connected nodes is the underlying architecture for many emerging applications. Miniaturized sensing modules are scattered like seeds [37, 79] or carried by insects [7, 8, 38] to scale a vast region for fine-grained sensor networks toward border protection [10], animal migration tracking [44], or precision agriculture [23, 24, 81]. Flocks of drones can localize each other to fly in formation for charting inaccessible regions [22, 34, 71] or for airshows [2]. A smart network of tags can enable city-scale tracking of deliveries or missing objects in real-time [52]. Future of the cellular [21, 68] and vehicular networks [4] can pave the path toward autonomy [86, 95] and road safety [90]. Localization of the nodes in such large networks is an essential requirement and at the same time a challenging problem when the number of nodes is large. In this paper, we present *Locate3D*, a system that is grounded on both theoretical and practical foundations, providing a reliable framework for fast peer-to-peer localization as well as orientation of nodes on a large network.

Multidimensional scaling (MDS) has been the essence of nearly all large-scale localization algorithms for wireless networks. The primary reason MDS is adapted so widely is its ability to reconstruct the relative map of the network nodes with little infrastructural support, even in the form of the anchor nodes with known locations. MDS-based algorithms are fundamentally centralized ranging-based systems that consider inter-node distances in the Euclidean space to optimize for sensor locations. It, however, suited the capacity of the mobile nodes that can use the time of flight of the signal or RSSI-based model to estimate ranges without additional hardware requirements or computational complexity. However, recent mobile nodes have evolved not only to accurately sense the ranges but are also equipped with multi-dimensional antenna arrays to estimate reliable angles of the nodes. For instance, around 20 years after being released for commercial applications in 2002 [19], off-the-shelf UWB sensors can now sense the angle of the received signal [65, 66]. Unfortunately, the entire class of the MDS-based network localization algorithms can not take advantage of this new-found capability of the mobile nodes. Intuitively the angle information of the peer nodes can serve as additional constraints of the network topology leading to faster convergence to the location estimation. Moreover, the range and angles are estimated simultaneously per exchange of signals between nodes without incurring additional measurement latency. This convenient information is wasted as the MDS objective function can not jointly optimize on the Euclidean distance and angle plane. We develop a new network localization algorithm with a redefined objective function to include range and angle together for optimization to bridge this gap.

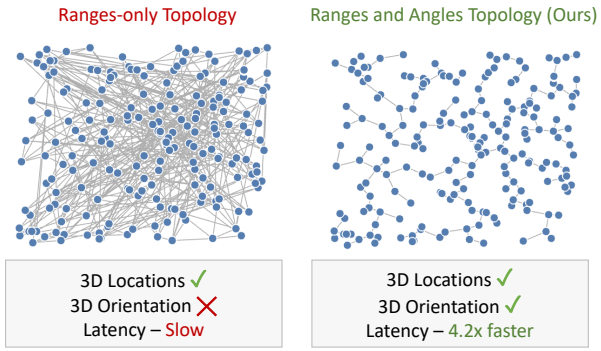


Figure 1: *Locate3D*'s approach to include both angle and edge constraints for faster and efficient localization.

The scalability of a network localization solution depends on several practical factors beyond the correctness of the theoretical formulations. The level of dependency on the infrastructure is the most crucial of them. In addition, a dynamic topology of mobile nodes requires short update latency of location estimations. Like any peer-to-peer localization system, *Locate3D* requires at least four anchor nodes for the unambiguous global 3D location of the nodes. However, the relative locations of the entire topology are correct with no assumption on the anchors. We enable our algorithms to incorporate any number of available anchor nodes and other infrastructural opportunities to create virtual anchors that enhance overall localization accuracy. In the proposed system, the joint range and angle-based optimization reduce the measurement and initial topology estimation latency, then a spanning-tree-based optimal edge selection expedites updates on locations after initial estimation, and finally, a graph rigidity-based solution makes the estimation robust to local occlusions or poor peer-to-peer connectivity.

A class of solutions for network localization resort to multimodal data to improve accuracy and reduce the update latency of the system. While effective in a small number of nodes and within restricted environmental conditions, the multimodality assumption limits the scalability of the system. It is infeasible to maintain homogeneous data quality with thousands of nodes spread across a large geographical region. For instance, some recent papers [51] use Visual Inertial Odometry (VIO) - a camera-based solution to track orientation - to improve localization accuracy. We argue that an unimodal solution is ideal for large networks in terms of consistency and ease of practical deployment and maintenance. It makes our solution equally applicable to the networks of resource-constrained low-power nodes. The core localization algorithm is, however, applicable to any modality that can measure the peer-to-peer range and relative angles, and naturally, the location accuracy is defined by the accuracy of the measurements. For instantiating the algorithm in a prototype and realistic large-scale simulations, we consider off-the-shelf Ultra-WideBand (UWB) wireless sensing nodes.

This paper strives to improve the accuracy of 3D localization of UWB-enabled nodes on a large single-modality network as shown in Figure 1. To this end, we have made the following three specific contributions at the current stage of this project:

- Developed a novel 3D network localization algorithm that jointly incorporates range and angle information for topology estimation. This algorithm achieves a 75% latency improvement and a median localization accuracy of 12.09 meters for a 100,000-node network spanning several kilometers, using only 15 static anchors. In a 32-node, three-floor building-scale experiment, the algorithm demonstrates a localization accuracy of 0.86 meters.
- Designed a system with a supporting algorithm that addresses challenges of occlusions and limited sensor field of view. The system includes an optimal spanning tree estimation algorithm and a decomposition technique that segments non-rigid graphs into smaller rigid subgraphs, considering both range and angle constraints.
- Implemented a working prototype with UWB-enabled nodes and evaluated its performance in real-world scenarios, including room-scale and multi-floor building experiments. Demonstrated the system's scalability through a city-scale simulation using real-world UWB measurement traces.

2 System Design

Our system, *Locate3D*, uses pair-wise Ultra-Wideband (UWB) RF measurements, specifically range and angle-of-arrival (AoA) information, to determine the precise 3D positions of interconnected nodes in relation to one another. The system is designed using four key components:

Joint range-angle localization: We develop an optimization method that uses range and angle measurements as constraints to build a network of localized nodes. Incorporating angles ensures an accurate location estimation with fewer number of edges compared to a range-only method, effectively reducing the latency up to 4.2×.

Reference frame transformation: Each node's angle measurement is a composite value of both the relative orientation (determined by the antenna angle with respect to the peer's antenna) and the AoA. *Locate3D* efficiently decomposes this combined measurement to transform all node axes to a common frame of reference.

Scaling to large networks: We develop an edge selection strategy that ensures the formulation of the optimal spanning tree, facilitating the interconnection of large topologies. Large networks often contain flexible or unconstrained edges, which can result in isolated but structurally rigid subgraphs. We aim to identify and resolve these subgraphs separately.

Opportunistic integration with infrastructure: *Locate3D* can leverage any pre-existing infrastructure, incorporating it as anchor points to refine localization accuracy. We further extend our system's capability by introducing the concept of virtual anchors. This mechanism temporarily designates specific mobile nodes as anchors and adjusts edge weights, disseminating the accuracy throughout the network. Below, we discuss the individual algorithmic contributions in detail.

2.1 Joint Range-Angle Localization

We formulate the localization problem as an optimization problem that aims to minimize the difference between measured pairwise distances and the distances corresponding to the estimated coordinates. There is a plethora of work done in network localization using range measurements only.

SMACOF [15, 33], non-metric MDS etc [9, 61, 78]. Some recent works also account for missing edges [72, 73] and NLOS cases [14, 17, 82]. However, none of the works have jointly incorporated AoA information in the network.

Problem formulation: Consider a topology of N nodes (mobile devices), with unknown locations $X = [(x_1, y_1, z_1), \cdots (x_N, y_N, z_N)]$. Suppose the nodes measure the range between each other, $\hat{r_{ij}}$ where (i, j) is the pair nodes i and j in the topology. We aim to solve for X while minimizing the cost function:

$$\min_X \sum_{i,j} w_{ij}(\hat{r_{ij}} - r_{ij}(X))^2 \tag{1}$$

where, w_{ij} is the weight assigned to edge between nodes i and j and $r_{ij}(X)$ is the euclidean distance between them given by Equation 2. We set $w_{ij} = 0$ for missing edges.

$$r_{ij}(X) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \tag{2}$$

Adding angles in the topology: Incorporating angles into the network topology offers the benefit of reducing the necessary number of edges, which subsequently decreases latency. This approach, however, is not as straightforward due to the highly non-convex and discontinuous nature of angular loss functions when simply added with the range-based loss function.
Existing works [11,50,64] which use L1 and L2 losses directly on this loss function show that the objective function has many local minima.
We can compute the angular loss using the function $f(X)$, expressed as:

$$f(X) = \left[\hat{\theta_{ij}} - \arctan\left[\frac{y_i - y_j}{x_i - x_j}\right]\right]^2 \tag{3}$$

This function encapsulates the difference between the measured angles, denoted by $\hat{\theta_{ij}}$, and the angles calculated from the estimated coordinates. The non-convex nature of $f(X)$ occurs due to the arctangent and the least square operation on angles. This results in multiple local minimums, making it highly prone to generating inaccurate topologies. The primary contributor to this issue is the restrictive interval $[-\pi/2, \pi/2]$ that the arctangent function operates in, failing to account for points in the left quadrants of the plane, thereby resulting in the same angles for coordinates (x, y) and $(-x, -y)$. To address this, we first use the 2-argument arctangent, a variant of the arctangent function that considers both x and y inputs and adjusts for the signs, hence returning angles within the interval $[-\pi, \pi]$.
Unfortunately, the transformation is not enough, as the gradient consists of non-differential making it prone to getting ensnared in local minimums. To overcome this, we apply another transformation to the loss function. We take the negative cosine of the angles, creating a smoother, continuous, and differentiable function restricted within the $[0, 1]$ range. This transformation of the new loss function, is defined as:

$$f(X) = 1 - cos(\hat{\theta_{ij}} - arctan2(y_i - y_j, x_i - x_j)) \tag{4}$$

Finally, we combine the range and angular loss functions to efficiently integrate angles in the network topology optimization formulating a joint optimization problem.

$$\min_X \left[\frac{\sum_{i,j} w^r_{ij}(\hat{r_{ij}} - r_{ij}(X))^2}{\sum_{i,j} \hat{r_{ij}}} + \sum_{i,j} w^\theta_{ij}(1 - cos(\hat{\theta_{ij}} - \theta_{ij}(X)))\right] \tag{5}$$

where, $\theta_{ij}(X) = arctan2(y_i - y_j, x_i - x_j)$. Note that we scale the range loss function with $(\sum_i \sum_j \hat{r_{ij}})$. The range loss can be much higher compared to the angular loss, which is in the range $[0, 1]$, and can dominate the overall gradient/overshadow the angular loss. Hence, we normalize the range loss function before adding the angular loss part.

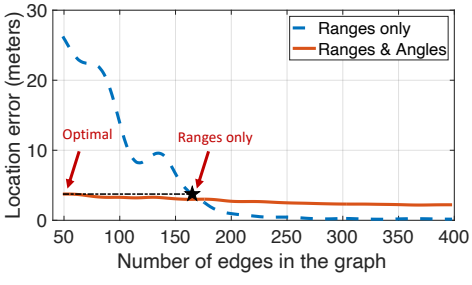


Figure 2: Comparative analysis of constraints: Incorporating angles reduces the number of edges required to attain the same level of accuracy as the "Ranges only" approach.

We conducted a simulation to evaluate the effectiveness of integrating angular measurements with distance constraints in topology optimization. The simulation involved a 50-node network within a $100m \times 100m$ area, incrementally adding edges under two constraint scenarios: ranges only, and both ranges and angles. Figure 2 shows that using only ranges required more than $3\times$ the number of edges to achieve accuracy comparable to our method combining ranges and angles. However, as more edges were added, the range-only approach eventually surpassed the range-angle method in accuracy. This can be attributed to the use of raw angles, which may include noisy measurements, and the random selection of edges rather than optimal spanning tree edges. Despite this, the method promises latency improvement over range-only methods without compromising accuracy even for a sub-optimal set of edges. We address the challenges related to noisy angle measurements and edge selection in Section 2.2, where we present techniques for filtering erroneous angles and constructing optimal spanning trees.

2.2 Scaling to Large networks

In this section, we detail how *Locate3D* can localize large node networks efficiently. To adapt to city-scale networks, we address three significant challenges linked with large networks.

• *Optimal edge selection*: Identifying the optimal edges to sample in the graph topology is vital to forming a unique, rigid structure while optimizing latency and accuracy.

• *Handling erroneous angles*: While in theory, all measured edges should have angle data, the limited FOV of commercial antenna arrays can sometimes result in inaccurate angle information. Correcting these errors is vital as they can adversely affect the graph's rigidity constraints.

• *Rigidity decomposition*: Large networks frequently consist of flexible or unconstrained edges which leads to smaller but rigid subgraphs. We need to identify and solve these subgraphs individually.

Next, we will elaborate on how *Locate3D* mitigates these challenges to scale to larger networks, introducing techniques that could reduce *Locate3D*'s latency by up to 4.2×.

2.2.1 Optimal Edge Selection

During any measurement time iteration, *Locate3D* compiles a list of optimal edges that are to be measured within the graph. These edges are chosen based on three criteria - (1) They form a rigid and unique topology, eliminating flexibility or ambiguity. (2) They are minimal in number to decrease overall latency. (3) They are feasible, meaning the edges fall within the necessary range and angle FoV.

In a topology comprising n nodes, a maximum of $n(n-1)$ edges are potentially available. However, acquiring data for each possible edge isn't viable as it is time-consuming and increases exponentially as n increases. Hence, rather than overconstraining the topology, we can create the topology with significantly fewer edges. According to Laman's condition [42], for a system that doesn't utilize angles, the minimum number of edges equals to $2n-3$ and $3n-4$ for 2D and 3D setups respectively. Interestingly, our approach necessitates only a *single* edge per node to constrain it, amounting to only $n-1$ edges in total. This efficiency stems from the fact that a single edge contains three constraints - range, azimuth angle, and elevation angle. Moreover, any random $n-1$ edges may not always be enough, as some edges could be redundant. Rather we need *well-distributed* $n-1$ edges to make the topology rigid and accurate. This renders our problem closely analogous to the *Minimum Spanning Tree problem* found in graph theory [28].

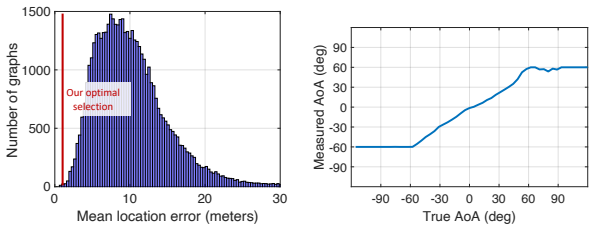
Minimum spanning tree: The minimum spanning tree (MST) is a subset of the graph's edges connecting all nodes with the least total edge weight. In our system, we utilize Kruskal's algorithm, a greedy algorithm that arranges the graph's edges in increasing order of their weights and adds edges to the MST as long as they do not form a cycle, thus guaranteeing the minimum combined edge weights.

Edge weight calculation: Our weight assignment algorithm minimizes localization error in the MST by utilizing prior knowledge of the network topology from the previous iteration, including estimated range (r) and angle (θ) values between nodes. The *area of uncertainty*, denoted as A, is calculated using sector area, $A = \pi(\alpha\sigma_r)^2\frac{\alpha\sigma_\theta}{2\pi}$, where σ_r and σ_θ are the standard deviations of the range and angle estimates, respectively, and α is a factor that scales the standard deviations based on LOS ($\alpha = 1$) or NLOS ($\alpha > 1$) conditions. The edge weight, w, is then computed as $w = A$, representing the potential location uncertainty based on the estimated range, angle, and their associated error distributions. Edges with shorter distances and smaller standard deviations inherently have a smaller area of uncertainty, and edges with available angle information are favored over those restricted to range data only. Figure 3a demonstrates that our algorithm's optimal spanning tree ranks within the top 1% of all possible spanning trees in terms of localization error. Algorithm 1 uses these estimated range and angle values to compute edge weights and form the MST, iteratively selecting the lowest-weight edges without introducing cycles. Actual range and angle measurements are taken for the selected edges once the MST is formed. During the cold start phase, when no prior knowledge is available, all nodes measure their neighbors, similar to traditional systems like [51, 94], resulting in increased latency for the first iteration. Subsequent iterations efficiently select edges for measurement by leveraging the prior knowledge of the network topology.

Algorithm 1 Optimal Edge Selection Algorithm

Input: $G(V,E)$, $\hat{r}$, $\hat{\theta}$, σ_r, σ_θ, α **Output:** $M(V_{MST}, E_{MST})$

```
procedure
    for each E in G do
        Retrieve r̂, θ̂, σ_r, σ_θ, and α for current E
        Assign weight w(E) = π(ασ_r)^2 ασ_θ/2π
    end for
    G_sorted(V,E) = Sort(G(V,E), w(E))
    Initialize M(V_MST, E_MST) as empty
    for each V,E in G_sorted do
        if adding V,E to M does not form a cycle then
            Add V,E to M
        end if
    end for
    return M(V_MST, E_MST)
end procedure
```



(a) (b)

Figure 3: (a) Histogram of localization errors for all spanning trees. (b) Reported angles by COTS UWB sensor.

2.2.2 Handling Erroneous Angles

In practical settings, nodes may not consistently measure each other's ranges or angles due to limited sensing range or angular FOV. This is due to the limited FOV of antenna arrays present in commonly used UWB modules like Decawave [65, 66] and NXP [54]. These modules typically restrict the AoA field of view (FOV) to between 90° and 120° to maintain a respectable accuracy in measured AoA, primarily because the AoA estimation algorithm, which depends on the phase difference of the incoming signal, can result in significant errors when the AoA approaches the broadside of the antenna. Figure 3b shows the reported angles by an off-the-shelf UWB sensor [54]. To address this, *Locate3D* employs two strategies for handling erroneous angles. First, we filter and flag angles as erroneous if they fall outside the range of $|FOV_{angle} - tolerance|$, effectively removing angles close to the broadside. Second, when available, *Locate3D* tracks the rotation of each node using its inertial sensors. Utilizing this rotation data, we can determine if any two nodes are within each other's FOV and flag the angles as valid or erroneous accordingly. In scenarios where some nodes lack inertial sensors or AoA estimation capability, *Locate3D* can still perform localization using the filtered angle data from the nodes equipped with these capabilities.

2.2.3 Rigidity Decomposition

While previous sections operate under the presumption of network topology rigidity, ensuring a unique realization of topology with given constraints, this isn't always the case. Notably, the optimal spanning tree derived earlier doesn't inherently ensure rigidity. Even fully connected graphs can remain non-rigid due to absent constraints, as illustrated in Figure 4 where spanning trees lack rigidity due to missing angle constraints. To address this, we develop a rigidity-based graph decomposition technique that segments the graph into smaller *rigid subgraphs*. These subgraphs can then be solved with refined edge weights ensuring that the topology is both rigid and uniquely realizable.

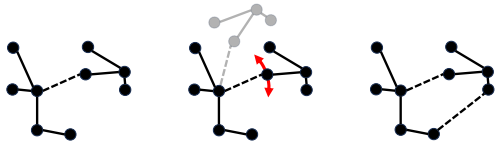


Figure 4: Different spanning trees representing rigid and non-rigid graphs. Solid lines indicate both range+angle edges, and dashed lines indicate range-only edges. (a) A connected but non-rigid graph due to missing angle information in an edge. (b) The subgraph is free to rotate. (c) Adding a range measurement makes the graph rigid.

Quantifying rigidity: The Rigidity of a topology determines whether it is possible to uniquely determine the location of all nodes in the topology with respect to other nodes. This leaves us with some trivial transformations of topology like translations and rotations in the space. The rigidity can be quantified using the degree of freedom (DoF) of the graph. Each node has three DoFs, representing movements in the x, y, and z directions. Thus, a graph with n nodes inherently has $3n$ DoFs. As constraints are imposed, the available DoFs decrease. A graph is deemed rigid if its DoFs are ≤ 3, with these residual DoFs accounting for the whole-graph translational motions.

Eigenanalysis of rigidity matrix: To identify independent and unconstrained motions that the topology can perform without violating the constraints, we perform the eigenvector analysis of the topology. To capture the constraints in a compact matrix form, we first introduce the rigidity matrix, R. Every row of R denotes a unique constraint equation, which could be a distance or angle constraint between nodes. Each column of R corresponds to the x, y and z coordinates for each node in the graph. Thus, for a graph having n nodes, the matrix expands to have $3n$ columns.
Each entry r_{ij} in this matrix is determined by the partial derivatives of the constraint equations 2 and 3, as given by:

$$r_{ij} = \begin{cases} \frac{\partial d_{ij}}{\partial x_m} \\ \frac{\partial \theta_{ij}}{\partial y_m} \\ 0 \end{cases} \begin{array}{l} \text{if edge } ij \text{ is a distance constraint} \\ \text{if edge } ij \text{ is an angle constraint} \\ \text{otherwise} \end{array}$$

$$R = \begin{bmatrix} \frac{\partial d_{1,2}}{\partial x_1} & \frac{\partial d_{1,2}}{\partial y_1} & \cdots & \frac{\partial d_{1,2}}{\partial x_n} & \frac{\partial d_{1,2}}{\partial y_n} \\ \frac{\partial d_{2,3}}{\partial x_1} & \frac{\partial d_{2,3}}{\partial y_1} & \cdots & \frac{\partial d_{2,3}}{\partial x_n} & \frac{\partial d_{2,3}}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial \theta_{1,2}}{\partial x_1} & \frac{\partial \theta_{1,2}}{\partial y_1} & \cdots & \frac{\partial \theta_{1,2}}{\partial x_n} & \frac{\partial \theta_{1,2}}{\partial y_n} \\ \frac{\partial \theta_{2,3}}{\partial x_1} & \frac{\partial \theta_{2,3}}{\partial y_1} & \cdots & \frac{\partial \theta_{2,3}}{\partial x_n} & \frac{\partial \theta_{2,3}}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \end{bmatrix}$$

Let $\lambda_1, \lambda_2, \ldots, \lambda_m$ be the eigenvalues of the matrix $M = RR^T$, where M is the symmetric and positive semi-definite matrix of R. Topology is uniquely rigid if the number of zero eigenvalues of M is equal to $3n-3$. A number of zero eigenvalues, or the degree of freedom, can be represented as:

$$DoF = \sum_{i=1}^{m} \delta(\lambda_i), \quad \text{where } \delta(\lambda) = \begin{cases} 1 & \text{if } |\lambda| < \varepsilon \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Here, ε is a small positive number close to zero, chosen to account for numerical inaccuracies (e.g., due to floating-point representation limits in computers). If $|\lambda_i| < \varepsilon$, it is considered a zero eigenvalue. Figure 5 illustrates the eigenvectors corresponding to the three zero eigenvalues λ_1, λ_2, and λ_3 showing the motion of each node.

Constructing subgraphs: By examining the zero eigenvalues, we gain insights into the *displacement vectors* of each node. For a fully rigid topology, these vectors maintain consistent magnitudes and directions across nodes. In contrast, non-rigid graphs display varied displacements.

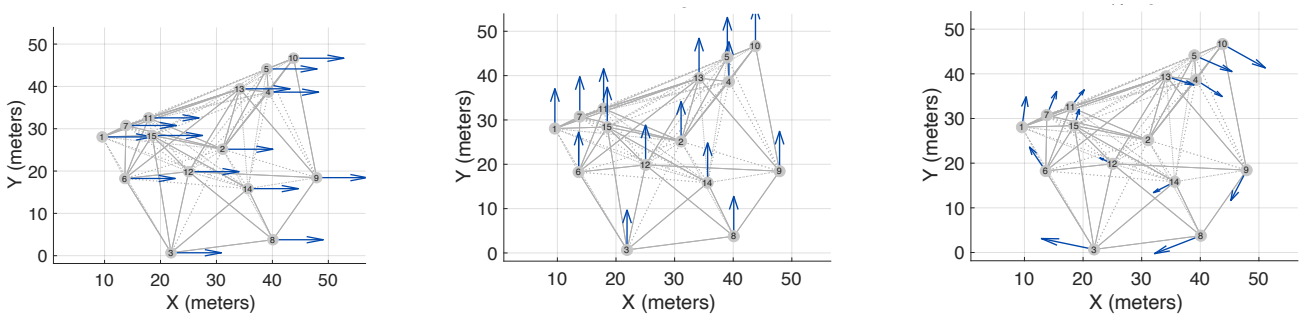


Figure 5: The displacements corresponding to zero eigenvalues represent the translational and rotational motions that the nodes can undergo without violating any constraint.

Our core intuition is that displacement vectors can be used for decomposition: nodes that have identical displacement vectors in terms of magnitude and direction inherently form a rigid subset. This means they can only undergo collective motion. To extract and group these subsets, we first reshape the eigenvector into a 3×n matrix to extract the displacement vectors of each node. Each row within this matrix signifies the displacement of a node across the three axes. Next, we identify the unique displacement vectors in this matrix where every unique vector signals the presence of a distinct "rigid subgraph" within the primary graph. By grouping the nodes corresponding to unique displacement vectors, we find the vertices belonging to the associated "rigid subgraph".

Critical edges list: Beyond recognizing rigid subgraphs, our approach also pinpoints the edges connecting the separate subgraphs. We term these edges as 'critical edges', as these edges are crucial for rigidity, serving as the connectors between independent rigid substructures. For enhanced analysis in subsequent iterations, we also prioritize these critical edges based on their latest computed locations. This prioritized list of edges underlines those edges that are more likely to come into proximity with one another, thus enhancing the overall scalability and rigidity of the graph.

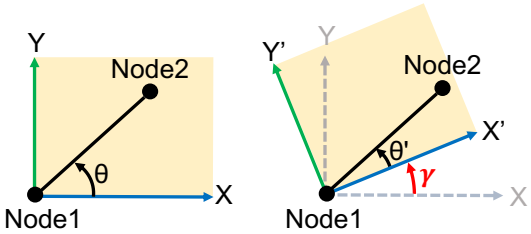


Figure 6: (a) When Node 1 is aligned with the global frame of reference, it reports that Node 2 is positioned at angle θ. (b) When Node 1 is rotated by γ relative to the global frame of reference in a 3D space, it reports a distinctly different angle θ′, for the same node.

2.3 Reference Frame Transformation

In the above section, we assume that all nodes share a common frame of reference. Here, we relax this assumption and address the challenges arising from differing local frames of reference for each node. Every node measures angles based on two factors: its relative orientation (defined by its antenna angle concerning another node's antenna) and the AoA. The challenge lies in the fact that these angle measurements are in the node's local frame of reference, which moves according to the node's orientation within a broader, global context. Consider the 2D example in Figure 6. In scenario (a), Node 1 identifies Node 2 at an angle θ. But in scenario (b), after Node 1 turns by an angle γ, it sees Node 2 at a different angle θ′. This change in reported angles is due to Node 1's rotation, even though the positions of the nodes didn't change. This issue gets complex in 3D, where nodes can rotate in any direction. Our goal is to deconstruct these angle measurements from orientation offsets and align all nodes to a singular, shared frame of reference.

2.3.1 Estimating Orientation Offsets

Every node's orientation in a 3D space is defined by a set of three rotation angles. Specifically, they depict the node's rotations about its X, Y, and Z axes and are recognized as the Roll(α), Pitch(β) and Yaw(γ) [89].

To estimate a node's orientation, we leverage a fundamental observation: When the orientations of nodes are in the same global frame, the measured azimuth AoA (θ) and elevation AoA (ϕ) are complementary, implying they obey the condition: $\theta_{ij} - \theta_{ji} = \pi$ and $\phi_{ij} = -\phi_{ji}$ (Figure 7). Using this insight, we formulate an optimization problem with the goal of estimating the rotation offsets (α, β, γ). The objective is to minimize the deviation from the above-mentioned constraints:

$$\min_{\alpha,\beta,\gamma} \sum_{i,j} (\hat{\theta}_{ij} - \hat{\theta}_{ji} - \pi) + (\hat{\phi}_{ij} + \hat{\phi}_{ji})$$

In this equation, $\hat{\theta}_{ij}$ and $\hat{\phi}_{ij}$ denote the rotated azimuth and elevation angles after the node has undergone rotation by α_i, β_i, and γ_i. Solving for this optimization provides us with the required orientation offsets to align all nodes in the system with the global frame of reference. The residual angles, $\hat{\theta}$ and $\hat{\phi}$, thus correspond to the AoAs in the global frame.

Next, we break down the steps in the optimization problem, detailing how we compute $\hat{\theta}$ and $\hat{\phi}$ through the rotation of local frames. The rotation in a 3D context can be decomposed into three unique rotations around the x, y, and z basis axes.

Step1 - Determine the unit vector based on the estimated

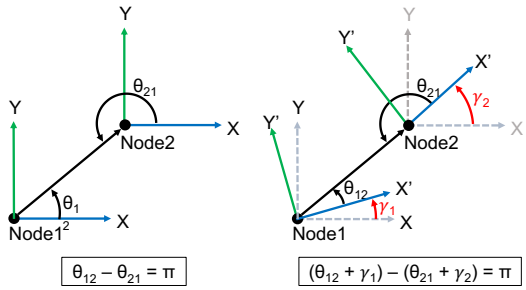


Figure 7: The relation between AoAs reported by two nodes (a) when nodes are perfectly aligned, and (b) when nodes are oriented with angle γ relative to the global frame of reference.

azimuth and elevation angles:

$$U = \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{bmatrix} = \begin{bmatrix} cos(\phi) * cos(\theta) \\ cos(\phi) * sin(\theta) \\ sin(\phi) \end{bmatrix}$$

Step2 - Apply the rotation matrix [76] to adjust the unit vector using the computed roll, pitch, and yaw angles (α, β, and γ). This results in a new unit vector, V, which reflects the node's orientation post-adjustment.

$$V = R_z(\gamma)R_y(\beta)R_x(\alpha)U$$

$$V = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & -\cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & -\cos\alpha \end{bmatrix} U$$

Step3 - Compute the azimuth and elevation angles from the rotated vector V to obtain the corrected AoA measurements.

$$\begin{bmatrix} \hat{\theta} \\ \hat{\phi} \end{bmatrix} = \begin{bmatrix} tan^{-1}(y/z) \\ tan^{-1}(z/\sqrt{x^2+y^2+z^2}) \end{bmatrix}$$

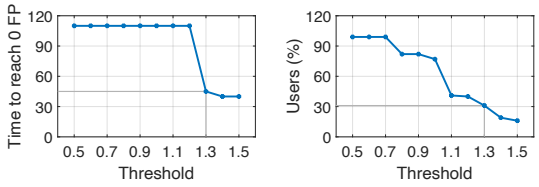
If we already have roll and pitch data from another sensor (e.g., IMU), the error function can become simpler. Instead of considering all three rotation angles, we can solve for yaw (γ). When the optimization is initiated with $\alpha_1 = \beta_1 = \gamma_1 = 0$, the first node sets the reference frame, and thus the orientations of all subsequent nodes are calculated relative to this frame.

3 Opportunistic Anchor Integration

Locate3D is fundamentally designed to function without the necessity for anchors. However, when anchors are available in the infrastructure, *Locate3D* can seamlessly integrate them to refine location and orientation results and transition from relative to global coordinates. This capability offers an edge over range-only systems, which strictly require at least three anchors for global coordinates, which is a challenging requirement in highly mobile environments. We categorize anchors into the following two types:

(a) Static anchors: They are conventional infrastructure anchors with known locations. These anchors have higher edge weightage, which increases their likelihood of being selected in MST over mobile volatile edges. Their impact on the accuracy is evaluated in a later section with varying anchor densities.

(b) Virtual anchors: As existing infrastructure cameras are available in many applications, we explore the potential to perform accurate localization in the vicinity and assign users within the camera's field-of-view as *virtual anchors*. A virtual anchor is a user with a high-confidence location obtained by an external modality, such as an infrastructure camera. It is important to note that while cameras can provide valuable localization information, they have limitations, such as reduced accuracy in low-light conditions.



(a) Time taken (in seconds) (b) Percent of users

Figure 8: (a) Time to reach '0 False positives'. (b) Percentage of users registered for varying trajectory matching thresholds.

User registration: Our registration technique uses correlation in user trajectories to incorporate human motion dynamics, such as varying walking speeds and stationary periods. The system benefits from *heading direction*, *cosine similarity of motion*, and speed analysis. We observe a scenario where some users are equipped with our node, while others are not, all within the camera's FoV. Our primary objective is to accurately register the system users in the camera's FoV while avoiding misclassifications. To address this, we adopt a stringent approach to trajectory similarity, aiming for zero false positives (FP). As depicted in Figure 8, keeping a higher threshold for registration effectively achieves zero FPs. By windowing 45 seconds of motion data, we register approximately 30% of visible users. It is important to note that while the availability of anchors can improve the accuracy, *Locate3D* does not rely on them for operation. We evaluate the system's performance under various anchor densities, to show its adaptability to different infrastructure setups.

4 Implementation

To this end, we implement a prototype of *Locate3D* and perform experiments in room-level, building-level, and city-level scenarios.

Node prototype: We developed prototype mobile nodes of *Locate3D*, as shown in Figure 11a to evaluate the performance. Each node consists of a Raspberry Pi 3 [63] for computation, an Intel Realsense T261 [36] to collect visual inertial odometry data. We use the NXP SR150 UWB evaluation boards [54] to collect the range, azimuth, and elevation AoA data. We collect the UWB measurements at a rate of 20Hz. All the nodes are time-synced using NTP.

Software: All the prototype nodes are part of a network and communicate with each other using a UDP socket. We

collect and store the UWB range, AoA, and VIO data from all nodes using Python scripts running on Raspberry Pi. We then process all measurements offline using Matlab where our algorithms are implemented.

5 Evaluation

We evaluate *Locate3D*'s performance in three key scenarios: room-scale, building-scale, and city-scale. In each scenario, we compare *Locate3D* with state-of-the-art baselines and benchmark localization accuracy, system latency, and the impact of various conditions such as line-of-sight (LOS) and non-line-of-sight (NLOS), stationary and moving nodes, lighting conditions, and varying numbers of infrastructure anchors. To assess *Locate3D*'s performance, we have collected over 4 hours of real-world data using 32 nodes and employed a motion capture system [3] as a ground truth setup to evaluate the centimeter-level localization accuracy and sub-degree-level orientation accuracy. We compare *Locate3D* with two distinct baseline systems: an infrastructure-free system, Capella [51], that uses UWB+VIO for peer-to-peer 3D localization, and an infrastructure-dependent system, ULoc [94], that uses UWB anchors and angular information for parallel node triangulation. Additionally, we simulate a large city-scale network by stitching real-world measurements to evaluate *Locate3D*'s scalability for networks with up to 100,000 nodes.

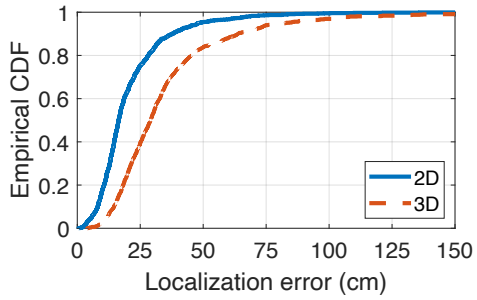


Figure 9: Room-scale evaluation: CDF of localization errors.

5.1 Localization Evaluation

5.1.1 Room-Scale Evaluation

We evaluate *Locate3D*'s performance in room-scale scenarios by conducting experiments in indoor rooms with dimensions of 6m $\times$ 8m and 4m $\times$ 20m, containing various furniture and equipment. We compare *Locate3D*'s performance with Cappella [51], a collaborative localization system that uses VIO for 3D tracking and inter-node UWB ranges to stitch individual trajectories using RBPF [80]. This evaluation involves 6 users walking naturally along random paths, emulating an AR/VR application, with additional users not participating in the system present to create a crowded and dynamic environment (Figure 11b). No anchors are used in these experiments as both *Locate3D* and Cappella are infrastructure-free. We use a Vicon motion capture system [83] as the ground truth, which provides sub-millimeter accuracy at a frame rate of 200Hz. The Vicon system consists of 12 Vantage V8 cameras mounted on the ceiling to track the ground truth locations and orientations of all nodes using active IR light and reflective markers attached to the nodes. We use 9 markers per node to ensure reliable tracking in all orientations. Figure 9 shows the CDF of the absolute localization error in 2D and 3D estimated by *Locate3D*, with median errors 18*cm* and 30*cm*, respectively, and 90th percentile errors of 38*cm* and 68*cm*.

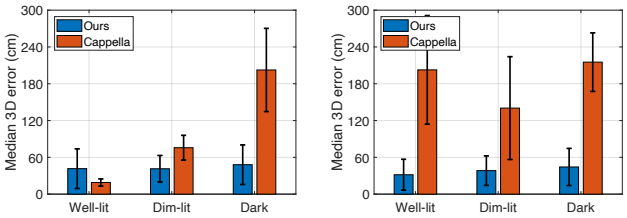


Figure 10: 3D Localization performance of *Locate3D* compared to baseline system - Cappella [51] - which uses visual odometry along with UWB. Results show performance in different lighting conditions for (a) Moving nodes and (b) Static nodes.

Effect of lighting: We evaluate *Locate3D*'s performance under varying lighting conditions to demonstrate its robustness to changes in illumination. As shown in Figure 10a, *Locate3D*'s performance remains consistent across different lighting conditions, while Cappella's performance degrades as indoor illumination decreases. The light intensity values for Well-lit, Dim-lit, and Dark conditions were 400, 150, 50 lux respectively which were controlled by analog light dimmers. This highlights *Locate3D*'s advantage of relying on RF-based peer-to-peer measurements, which are unaffected by visual conditions, in contrast to Cappella's reliance on VIO and visual features of the environment.

Effect of motion: Figure 10b shows that when nodes are static and the baseline system lacks a VIO trajectory tail, it fails to estimate 3D locations accurately. In contrast, *Locate3D* does not rely on odometry and can estimate the complete 6DOF location and orientation for all nodes, regardless of whether they are static or mobile.

5.1.2 Building-Scale Evaluation

We evaluate *Locate3D*'s performance in a multi-level building scenario using 32 nodes across multiple floors, as shown in Figure 11c, 11d. The experiments include a combination of LOS and NLOS conditions, with users walking in the building while holding nodes that capture images of calibrated AprilTags [87] placed on the walls for ground truth reference (Figure 11f). *Locate3D* achieves a median 3D localization error of 0.86 meters and an average orientation error of 4.5 degrees without using any infrastructure anchors. Figure 11e shows a snapshot of the estimated topology. To validate



(a) (b) (c) (d) (e) (f)

Figure 11: (a) Prototype built using Raspberry Pi, UWB sensor, Intel Realsense, IR markers for ground truth and a battery pack. (b) Room-scale evaluation. (c) Building-scale evaluation (d) 3D Lidar scan of the building for reference (not used in computation) (e) Snapshot of estimated locations and MST. (f) AprilTags [87] captured by nodes for ground truth (not used in computation).

the accuracy of the AprilTag-based ground truth system, we compared it against the Motion capture setup and found a 3D localization error of 0.11 meters. These results show *Locate3D*'s ability to provide accurate localization and orientation estimation in larger-scale, multi-level building environments without relying on infrastructure anchors. In section 5.2, we also do an ablation study to understand the contribution of each submodule to the system's latency and accuracy improvements.

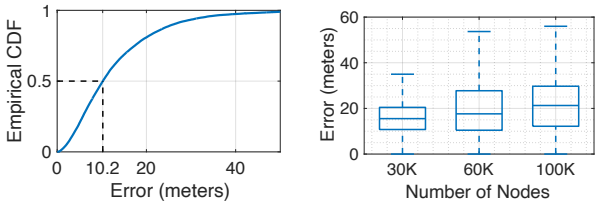


Figure 12: City-scale analysis: (a) CDF of localization errors of 30k nodes, 3800$m \times 3800m$ area, and 15 anchors. (b) Errors for 30k, 60k, and 100k nodes in 1500$m \times 1500m$ area and 1 anchor.

5.1.3 City-Scale Evaluation (Simulation)

We evaluate the performance of *Locate3D* in city-scale scenarios by simulating an environment with real-world collected data stitched in the simulation. The simulation incorporates real-world peer-to-peer measurements collected using 32 nodes in various environments, including indoor, outdoor, LOS, and NLOS scenarios. The dataset consists of 20,000 measurements covering ranges from 0.1 to 50 meters and AoA spanning the full 360 degrees from both static and mobile users. To emulate large-scale city networks with up to 100,000 nodes, we employ a data-driven approach by stitching together real-world measurements to preserve the noise properties. The simulation environment includes obstacles that mimic real-world urban settings, and the corresponding measurements are selected from the appropriate subset of the dataset (e.g., nodes 20m apart, 30° and 200° azimuth and elevation AoAs and NLOS edge) to maintain the integrity of the simulation. During the simulation process, we strictly adhere to the real-world measurements, ensuring that the nodes experience realistic noise and variations in range and AoA based on the specific distance and angle of the interaction. This enables us to evaluate *Locate3D*'s performance in large-scale networks while maintaining a strong connection to real-world conditions. Figure 12 shows the CDF of localization errors for 30k nodes using 15 anchors in a 3800$m \times 3800m$ area and the localization errors for 30k, 60k, and 100k nodes in a 1500$m \times 1500m$ area using 1 anchor (for global translation only). *Locate3D* achieves a median localization error of 12.09 meters for large-scale networks (100,000 nodes with 15 anchors) and 21 meters with 1 anchor.

We further evaluate *Locate3D* in a wide-area simulation of New York City, spanning approximately 22000$m \times 3200m$. We use 100k nodes with ground truth locations from the NYC transportation dataset [59], representing a Tuesday evening scenario. Figure 13 shows the results for configurations with 1 and 5 anchors. *Locate3D* achieved median localization errors of 82.31m and 34.19m for 1 and 5 anchors, respectively. The improved accuracy with more anchors demonstrates the system's ability to propagate location information effectively across wide-area environments.

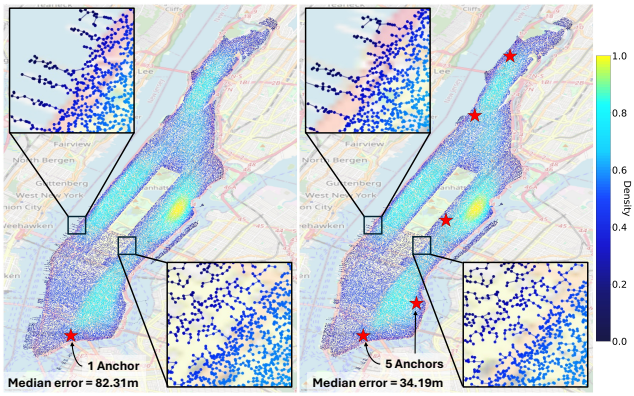
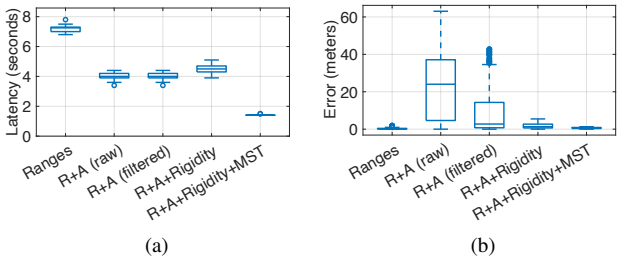


Figure 13: New York City wide-area simulation results: CDF of localization errors for 100,000 nodes using 1 (left figure) and 5 (right figure) anchors in a 22000$m \times 3200m$ area.

5.2 Latency Evaluation

To evaluate the effectiveness of each submodule in latency improvement, we conduct an experimental ablation study in the building-scale scenario. The results are shown in Figure 14. Starting with a baseline system that uses only range information, we observe that the introduction of

angle information (Range+Angle) significantly reduces the latency. However, the raw angle data leads to an increase in localization error due to the erroneous angles, as discussed in the section 2.2.2. To address this issue, we apply our filtering technique to fix erroneous angles which improves the accuracy while maintaining the latency reduction achieved by the introduction of angle information. The incorporation of rigidity constraints (R+A+Rigidity) further enhances the localization accuracy at a slight increase in latency compared to the filtered range and angle system. This is due to the addition of the critical edges that ensure a unique and unambiguous realization of the graph, as discussed in section 2.2.3. Finally, the optimal edge selection algorithm (R+A+MST) yields the most significant latency reduction while achieving localization accuracy comparable to the baseline system. This algorithm selects the optimal set of edges to minimize the overall latency while maintaining the graph's rigidity and uniqueness.



(a) (b)

Figure 14: Impact of submodules on (a) latency and (b) accuracy.

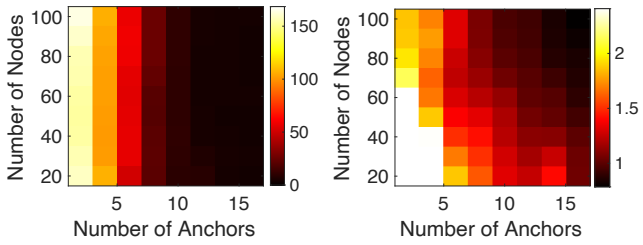
5.3 Micro-Benchmarks

5.3.1 Comparison with Infrastructure-Based Systems

To compare *Locate3D* with infrastructure-based systems, we evaluate the accuracy and scalability of *Locate3D* with a baseline infrastructure-based system ULoc [94], while varying the area, number of nodes, and number of anchors Our results show that *Locate3D* outperforms infrastructure-based systems in two key scenarios:

Sparse anchor deployments: *Locate3D*'s peer-to-peer network enables localization even when nodes are beyond the range of anchors, while ULoc fails in such cases. Figure 15 shows that both ULoc and *Locate3D* achieve meter-level errors when the number of anchors is high. However, as the number of anchors decreases, ULoc's average localization error increases due to nodes being out of range. Figure 16a demonstrates that with a fixed number of anchors and increasing area space, *Locate3D* can localize almost all nodes using its spanning tree, while the infrastructure-dependent technique loses connections to nodes making them unreachable.

NLOS edges: *Locate3D*'s edge selection algorithm prioritizes LOS measurements over NLOS, optimizing for overall noise reduction in the graph. In contrast, ULoc, being an infrastructure-based system, is constrained to using



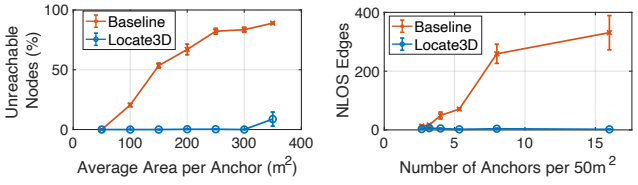
(a) Baseline (Infrastructure based) (b) *Locate3D*

Figure 15: Localization errors (in meters) for infrastructure baseline [94] and *Locate3D* for varying nodes and anchors.

anchor-to-node measurements regardless of whether they are LOS or NLOS. This limitation leads to the inclusion of noisy edges in the graph, affecting the overall localization performance. As shown in Figure 16b, *Locate3D* selectively includes LOS edges and minimizes the number of NLOS edges resulting in improved accuracy in realistic environments. By reducing the overall noise in the topology, *Locate3D* is able to provide more reliable and precise location estimates compared to ULoc, which is susceptible to errors introduced by NLOS measurements between nodes and anchors. This highlights the benefits of a peer-to-peer approach, which allows for greater flexibility in edge selection and noise reduction compared to infrastructure-based systems. However, infrastructure-based systems outperforms *Locate3D* in the following scenarios:

High anchor density and small-area experiments: As shown in Figure 15, infrastructure-based systems outperform in high anchor density cases where the error decreases due to close proximity with multiple anchors. Such small-scale scenarios with clear LOS and at least three anchors are where ULoc achieves better accuracy.

Locate3D's scope is to scale to a large number of nodes, while infrastructure-based systems are well-suited for small-scale deployments with sufficient anchor density.



(a) (b)

Figure 16: (a) Number of unreachable nodes and (b) Total number of NLOS measurements made with varying anchor area density.

5.3.2 LOS vs NLOS nodes

Figure 17(a) presents the CDF of the 2D and 3D localization errors in both line-of-sight and non-line-of-sight scenarios. In the NLOS scenario, there is a slight increase in localization error compared to the LOS scenario. This increase is primarily attributed to the higher noise levels in the received signals at the nodes due to obstructed signal paths and reflections. Specifically, in the 2D scenario, the median localization error

rises from 13*cm* in LOS to 28*cm* in NLOS. Similarly, in the 3D scenario, the median localization error increases from 31*cm* in LOS to 39*cm* in NLOS.

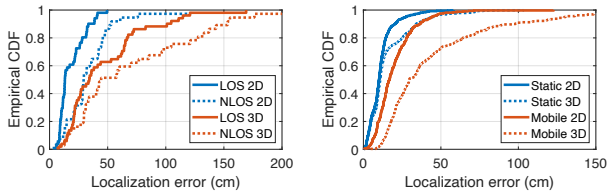
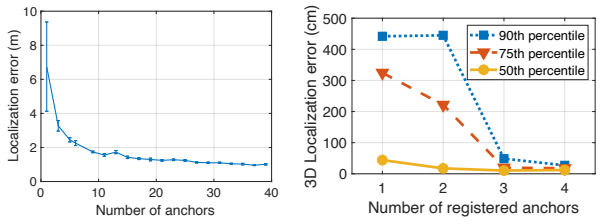


Figure 17: (a) LOS and NLOS localization errors. (b) Static vs Mobile nodes localization errors

5.3.3 Static vs Mobile nodes

To assess the impact of node mobility on the performance of *Locate3D*, we conduct experiments to compare static and mobile node scenarios. Figure 17(b) presents the CDF of 2D and 3D localization errors in both static and mobile node configurations. In the mobile scenario, the system experiences a slight degradation in performance due to varying environments and node vibrations. In the 2D scenario, the median localization error increases from a low of 6*cm* in the static setup to 13*cm* in the mobile scenario. Similarly, in the 3D scenario, the median localization error rises from 6.5*cm* in the static scenario to 26*cm* in the mobile scenario.



(a) (b)

Figure 18: (a) City-scale results for a 1000 node topology simulation in a $200 \times 200 \times 50$ meter 3D space for a varying number of static anchors. (b) Room-scale localization error for real-world 20-node experiments with varying numbers of anchors registered.

5.3.4 Impact of Anchor Density

We evaluate the impact of anchor density on localization performance in both large-scale simulations and real-world room-scale experiments. While *Locate3D* does not require anchors for operation, the presence of anchors can improve absolute localization accuracy in relative global coordinates. In a large-scale simulation with 1,000 nodes distributed within a 200m $\times$ 200m $\times$ 50m 3D space, *Locate3D* achieves a localization error below 2 meters with just 5 anchors (Figure 18a). In a real-world room-scale experiment, we evaluate the localization error while varying the number of registered anchors. *Locate3D* can peg the entire topology using only two anchors by leveraging angle information alongside location estimates. Figure 18b shows that the localization error decreases as more anchors are registered, with the main source of error being the rotation of the topology when only one anchor is present.

5.3.5 Ranging and AoA Performance

The localization errors inherently depend on the raging and AoA accuracies. We report the ranging and AoA estimation performance of the UWB sensor used in our implementation. While our system can be applied to any technology, here in the implementation we have used the UWB radio to get the distance and AoA error. So in this section, we report the accuracy of the ranging and angle estimation accuracy.
Figure 19 shows the ranging and AoA performance of our node for varying distances and incoming angles. We see that there are biases in certain angles as it is likely due to multipath. But still the 90^{th} percentile error remain within $\pm 5^{\circ}$.

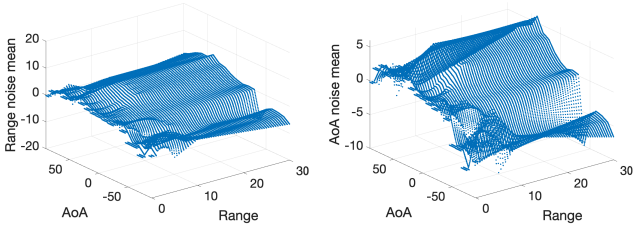


Figure 19: Ranging and AoA errors for various range-angles.

5.3.6 Orientation Estimation Accuracy

Figure 20(a) presents the CDF of the error in orientation estimation results. It is highly important for our system as these orientations play a critical role in transforming AoAs from the local frame of reference of the nodes to the global frame of reference, which is then used localization of nodes within a 3D space. Figure 20(a) shows that *Locate3D* consistently estimates accurate node's orientation with a median error of 4 degrees which is competitive with an inertial sensor.

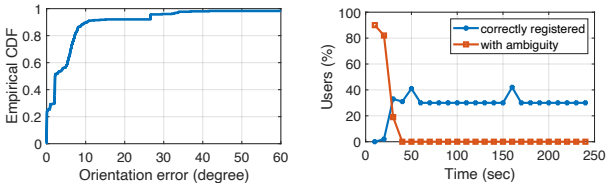


Figure 20: (a) CDF orientation errors. (b) Percent of users registered as Virtual anchors over time.

5.3.7 Virtual Anchor Registration

We evaluate the virtual anchor registration technique through a building-scale experiment with a network-connected camera. This setup included four node users and three non-participants walking within the camera's FoV, simulating typical crowd dynamics. Figure 20(b) shows the results that within the initial 40 seconds, the system registered 40% of the users as virtual anchors, while distinguishing between system users and non-participants.

6 Discussion

Scalability in real-time implementation: An extension of the proposed system with a semi-distributed approach can further improve its worst-case latency and robustness. In this case, the network is divided into sub-graphs managed by a local leader node. The leader nodes compute MSTs per sub-graphs in parallel and later aggregate to form the complete graph. This distributed architecture can offer better handling of high mobility scenarios subjected to future research.

Bootstrapping: *Locate3D*'s initial bootstrapping phase establishes the first set of measurements and constructs the initial graph without assumptions about node locations or network topology. Each node measures all neighboring nodes, building the graph incrementally based on responses. While this cold start phase naturally adds to the latency, it does not compromise localization accuracy. The duration of this phase depends on factors such as the number of nodes and application requirements. In subsequent iterations, *Locate3D* uses previous iteration information to efficiently select edges, reducing latency. While cold start is a common issue in state-of-the-art peer-to-peer localization systems, optimizing the bootstrapping process can be a direction for future work.

Applications: *Locate3D* caters to a wide range of applications requiring accurate and fast localization in infrastructure-free networks. Inspired by first responders scenarios, real-world experiments in multi-level buildings demonstrate the system's potential in emergency response scenarios. *Locate3D*'s ability to operate in GPS-denied environments and adapt to changes in network topology makes it suitable for disaster management and rescue operations. Its scalability, demonstrated through our city-scale simulations, shows its potential for smart city applications such as asset tracking, crowd monitoring, and urban planning. *Locate3D*'s framework can be easily adapted to incorporate application-specific parameters and constraints.

Advancements in angle field-of-view: Future sensor advancements, such as increased antenna counts, may enable 360° field-of-view coverage. In such cases, the spanning tree algorithm can be adapted to consider either range+angle edges or no edges, and account for varying noise distributions across different incoming angles. *Locate3D*'s framework is flexible to accommodate these advancements and generate optimal spanning trees without significant modifications.

Subgraphs without proximity for extended periods: Our critical edges technique addresses cases where subgraphs are out of vicinity for long durations. However, prolonged reliance can lead to drifting. Reinitializing the tree or cold-start can mitigate this issue. Presence of anchors in subgraphs can also provide reference points to align relative topology.

Other RF modalities: Our framework is adaptable to other modalities beyond UWB, including RF and acoustic sensors that can measure pairwise ranges and AoA [47, 48, 62, 88].

7 Related Work

Infrastructure-based systems: Motion tracking cameras (OptiTrack [1], Vicon [3]), passive markers (ARTags [41, 55], AprilTags [39, 87, 93]), and beacon-based solutions (UWB [57, 60, 75], Bluetooth [13, 43, 74], ultrasound [27, 43, 45]) are used for localization. These methods often combine ranging with odometry from IMU or VIO [16, 26, 49, 60, 69, 77, 85]. However, infrastructure-based systems are limited to prepared environments. *Locate3D* leverages peer-to-peer UWB ranges and angles, eliminating the need for infrastructure.

Infrastructure-free systems: Infrastructure-free localization determines relative positions between users without global references. This has been explored in sensor networks for stationary [56, 70] and mobile [18, 20, 53, 67] nodes, and in robotics using visual object detection [58, 84, 96] or odometry and distance measurements [6, 12, 31, 32, 46, 91, 92]. Recent research [51, 91, 92] employs visual-inertial features and UWB ranges for localization, but have limited scalability.

UWB localization: UWB-based systems have shown decimeter-level accuracy [25, 29, 30, 35, 40]. However, these systems often require a high number of infrastructure anchors, limiting their scalability. ULoc [94] uses AoA for triangulation but requires dense anchor deployments. XRLoc [5] employs a single large specialized anchor with multiple UWB nodes, focusing on small-scale, meter-level applications. SALMA [29] uses only one anchor but is limited to 2D and supports a small number of concurrent tags. In contrast, *Locate3D* is an infrastructure-free system that leverages both range and angle information for 6DoF localization. Unlike Cappella [51], which relies on VIO and UWB measurements, *Locate3D*'s performance is independent of lighting conditions. Moreover, *Locate3D* achieves localization without depending on odometry, preventing error accumulation.

8 Conclusion

This paper presents *Locate3D*, a peer-to-peer 3D localization and orientation estimation system for large networks. Our novel optimization and edge selection algorithm integrates both range and angle-of-arrival measurements for enhanced network topology estimation, demanding fewer edge measurements. As a result, *Locate3D* improves the latency by 75% without sacrificing accuracy, outperforming conventional range-only solutions.

Acknowledgement

This work was partially supported by NSF CAREER Award 2238433. We also thank the various companies that sponsor the iCoSMoS laboratory at UMD.

References

[1] Optitrack. https://optitrack.com/. Last accessed 12 March 2023.

[2] Technology and systems for drone shows. https://www.verge.aero/. Last accessed 21 September 2023.

[3] Vicon. https://www.vicon.com/. Last accessed 12 March 2023.

[4] Nima Alam and Andrew G Dempster. Cooperative positioning for vehicular networks: Facts and future. *IEEE transactions on intelligent transportation systems*, 14(4):1708–1717, 2013.

[5] Aditya Arun, Shunsuke Saruwatari, Sureel Shah, and Dinesh Bharadia. Xrloc: Accurate uwb localization to realize xr deployments. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*, pages 459–473, 2023.

[6] Nan Bai, Yuan Tian, Ye Liu, Zhengxi Yuan, Zhuoling Xiao, and Jun Zhou. A high-precision and low-cost imu-based indoor pedestrian positioning technique. *IEEE Sensors Journal*, 20(12):6716–6726, 2020.

[7] Yang Bai, Nakul Garg, and Nirupam Roy. Spidr: Ultra-low-power acoustic spatial sensing for micro-robot navigation. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 99–113, 2022.

[8] Yang Bai, Nakul Garg, and Nirupam Roy. Microstructure-assisted vision: Adding new senses to low-power devices. *GetMobile: Mobile Computing and Communications*, 27(4):15–20, 2024.

[9] Brian Beck, Robert Baxley, and Joseph Kim. Real-time, anchor-free node tracking using ultrawideband range and odometry data. In *2014 IEEE international conference on ultra-wideband (ICUWB)*, pages 286–291. IEEE, 2014.

[10] Neha Bhadwal, Vishu Madaan, Prateek Agrawal, Awadesh Shukla, and Anuj Kakran. Smart border surveillance system using wireless sensor network and computer vision. In *2019 international conference on Automation, Computational and Technology Management (ICACTM)*, pages 183–190. IEEE, 2019.

[11] Christophe Biernacki, Gilles Celeux, and Gérard Govaert. Choosing starting values for the em algorithm for getting the highest likelihood in multivariate gaussian mixture models. *Computational Statistics & Data Analysis*, 41(3-4):561–575, 2003.

[12] Zhiqiang Cao, Ran Liu, Chau Yuen, Achala Athukorala, Benny Kai Kiat Ng, Muraleetharan Mathanraj, and U-Xuan Tan. Relative localization of mobile robots with multiple ultra-wideband ranging measurements. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5857–5863. IEEE, 2021.

[13] Srdjan Čapkun, Maher Hamdi, and Jean-Pierre Hubaux. Gps-free positioning in mobile ad hoc networks. *Cluster Computing*, 5:157–167, 2002.

[14] Hongyang Chen, Gang Wang, Zizhuo Wang, Hing-Cheung So, and H Vincent Poor. Non-line-of-sight node localization based on semi-definite programming in wireless sensor networks. *IEEE Transactions on Wireless Communications*, 11(1):108–116, 2011.

[15] Jan De Leeuw and Patrick Mair. Multidimensional scaling using majorization: Smacof in r. *Journal of statistical software*, 31:1–30, 2009.

[16] Ashutosh Dhekne, Ayon Chakraborty, Karthikeyan Sundaresan, and Sampath Rangarajan. Trackio: Tracking first responders inside-out. In *NSDI*, volume 19, pages 751–764, 2019.

[17] Carmelo Di Franco, Amanda Prorok, Nikolay Atanasov, Benjamin Kempke, Prabal Dutta, Vijay Kumar, and George J Pappas. Calibration-free network localization using non-line-of-sight ultra-wideband measurements. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 235–246, 2017.

[18] Tolga Eren, OK Goldenberg, Walter Whiteley, Yang Richard Yang, A Stephen Morse, Brian DO Anderson, and Peter N Belhumeur. Rigidity, computation, and randomization in network localization. In *IEEE INFOCOM 2004*, volume 4, pages 2673–2684. IEEE, 2004.

[19] Fira. Unleashing the potential of uwb: Regulatory considerations. *FiRa Consortium*, August 2022.

[20] Nakul Garg, Yang Bai, and Nirupam Roy. Owlet: Enabling spatial information in ubiquitous acoustic devices. In *The 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '21), June 24–July 2, 2021, Virtual, WI, USA*. ACM.

[21] Nakul Garg, Aritrik Ghosh, and Nirupam Roy. Litefoot: Ultra-low-power localization using ambient cellular signals. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2024)*, New York, NY, USA, 2024. ACM. University of Maryland, College Park.

[22] Nakul Garg and Nirupam Roy. Enabling self-defense in small drones. In *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, pages 15–20, 2020.

[23] Nakul Garg and Nirupam Roy. poster: Ultra-low-power angle-of-arrival estimation using a single antenna. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, pages 569–570, 2023.

[24] Nakul Garg and Nirupam Roy. Sirius: A self-localization system for resource-constrained iot sensors. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, pages 289–302, 2023.

[25] Nakul Garg, Irtaza Shahid, Karthik Sankar, Mallesham Dasari, Ramanujan K Sheshadri, Karthikeyan Sundaresan, and Nirupam Roy. Bringing ar/vr to everyday life-a wireless localization perspective. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*, pages 142–142, 2023.

[26] Christian Gentner and Markus Ulmschneider. Simultaneous localization and mapping for pedestrians using low-cost ultra-wideband system and gyroscope. In *2017 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8. IEEE, 2017.

[27] David Gómez, Paula Tarrío, Juan Li, Ana M Bernardos, and José R Casar. Indoor augmented reality based on ultrasound localization systems. In *Highlights on Practical Applications of Agents and Multi-Agent Systems: International Workshops of PAAMS 2013, Salamanca, Spain, May 22-24, 2013. Proceedings 11*, pages 202–212. Springer, 2013.

[28] Ronald L Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[29] Bernhard Großwindhager, Michael Rath, Josef Kulmer, Mustafa S Bakr, Carlo Alberto Boano, Klaus Witrisal, and Kay Römer. Salma: Uwb-based single-anchor localization system using multipath assistance. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 132–144, 2018.

[30] Bernhard Großwindhager, Michael Stocker, Michael Rath, Carlo Alberto Boano, and Kay Römer. Snaploc: An ultra-fast uwb-based indoor localization system for an unlimited number of tags. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, pages 61–72, 2019.

[31] Kexin Guo, Xiuxian Li, and Lihua Xie. Ultra-wideband and odometry-based cooperative relative localization with application to multi-uav formation control. *IEEE transactions on cybernetics*, 50(6):2590–2603, 2019.

[32] Kexin Guo, Zhirong Qiu, Wei Meng, Lihua Xie, and Rodney Teo. Ultra-wideband based cooperative relative localization algorithm and experiments for multiple unmanned aerial vehicles in gps denied environments. *International Journal of Micro Air Vehicles*, 9(3):169–186, 2017.

[33] Moshe Hamaoui. Non-iterative mds method for collaborative network localization with sparse range and pointing measurements. *IEEE Transactions on Signal Processing*, 67(3):568–578, 2018.

[34] Yong-Gu Han, Youngho Cho, Ohseok Kwon, et al. The use of conservation drones in ecology and wildlife research. *Journal of Ecology and Environment*, 38(1):113–118, 2015.

[35] Frank Hartmann, Felix Pistorius, Andreas Lauber, Kai Hildenbrand, Jürgen Becker, and Wilhelm Stork. Design of an embedded uwb hardware platform for navigation in gps denied environments. In *2015 IEEE Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, pages 1–6. IEEE, 2015.

[36] Intel. https://dev.intelrealsense.com/docs/tracking-camera-t265-datasheet, 2023.

[37] Vikram Iyer, Hans Gaensbauer, Thomas L Daniel, and Shyamnath Gollakota. Wind dispersal of battery-free wireless devices. *Nature*, 603(7901):427–433, 2022.

[38] Vikram Iyer, Maruchi Kim, Shirley Xue, Anran Wang, and Shyamnath Gollakota. Airdropping sensor networks from drones and insects. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[39] Jan Kallwies, Bianca Forkel, and Hans-Joachim Wuensche. Determining and improving the localization accuracy of apriltag detection. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8288–8294. IEEE, 2020.

[40] Benjamin Kempke, Pat Pannuto, Bradford Campbell, and Prabal Dutta. Surepoint: Exploiting ultra wideband flooding and diversity to provide robust, scalable, high-fidelity indoor localization. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 137–149, 2016.

[41] Manfred Klopschitz and Dieter Schmalstieg. Automatic reconstruction of wide-area fiducial marker models. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 71–74. IEEE, 2007.

[42] Gerard Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering mathematics*, 4(4):331–340, 1970.

[43] Patrick Lazik, Niranjini Rajagopal, Oliver Shih, Bruno Sinopoli, and Anthony Rowe. Alps: A bluetooth and ultrasound platform for mapping and localization. In *Proceedings of the 13th ACM conference on embedded networked sensor systems*, pages 73–84, 2015.

[44] Inhee Lee, Roger Hsiao, Gordy Carichner, Chin-Wei Hsu, Mingyu Yang, Sara Shoouri, Katherine Ernst, Tess Carichner, Yuyang Li, Jaechan Lim, et al. msail: milligram-scale multi-modal sensor platform for monarch butterfly migration tracking. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 517–530, 2021.

[45] Jinyang Li, Zhiheng Xie, Xiaoshan Sun, Jian Tang, Hengchang Liu, and John A Stankovic. An automatic and accurate localization system for firefighters. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 13–24. IEEE, 2018.

[46] Ming Li, Zhuang Chang, Zhen Zhong, and Yan Gao. Relative localization in multi-robot systems based on dead reckoning and uwb ranging. In *2020 IEEE 23rd International Conference on Information Fusion (FUSION)*, pages 1–7. IEEE, 2020.

[47] Fen Liu, Jing Liu, Yuqing Yin, Wenhan Wang, Donghai Hu, Pengpeng Chen, and Qiang Niu. Survey on wifi-based indoor positioning techniques. *IET communications*, 14(9):1372–1383, 2020.

[48] Hongbo Liu, Yu Gan, Jie Yang, Simon Sidhom, Yan Wang, Yingying Chen, and Fan Ye. Push the limit of wifi based localization for smartphones. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 305–316, 2012.

[49] Ran Liu, Chau Yuen, Tri-Nhut Do, Dewei Jiao, Xiang Liu, and U-Xuan Tan. Cooperative relative positioning of mobile users by fusing imu inertial and uwb ranging information. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5623–5629. IEEE, 2017.

[50] Volodymyr Melnykov and Igor Melnykov. Initializing the em algorithm in gaussian mixture models with an unknown number of components. *Computational Statistics & Data Analysis*, 56(6):1381–1395, 2012.

[51] John Miller, Elahe Soltanaghai, Raewyn Duvall, Jeff Chen, Vikram Bhat, Nuno Pereira, and Anthony Rowe. Cappella: Establishing multi-user augmented reality sessions using inertial estimates and peer-to-peer ranging. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 428–440. IEEE, 2022.

[52] Daniel Minoli and Benedict Occhiogrosso. Ultrawideband (uwb) technology for smart cities iot applications. In *2018 IEEE international smart cities conference (ISC2)*, pages 1–8. IEEE, 2018.

[53] David Moore, John Leonard, Daniela Rus, and Seth Teller. Robust distributed network localization with noisy range measurements. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 50–61, 2004.

[54] Mouser. https://www.mouser.com/new/murata/murata-type2bp-uwb-module-eval-kit/, 2024.

[55] Alessandro Mulloni, Hartmut Seichter, and Dieter Schmalstieg. Handheld augmented reality indoor navigation with activity-based instructions. In *Proceedings of the 13th international conference on human computer interaction with mobile devices and services*, pages 211–220, 2011.

[56] Radhika Nagpal, Howard Shrobe, and Jonathan Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *Information Processing in Sensor Networks: Second International Workshop, IPSN 2003, Palo Alto, CA, USA, April 22–23, 2003 Proceedings*, pages 333–348. Springer, 2003.

[57] Thien Hoang Nguyen, Thien-Minh Nguyen, and Lihua Xie. Range-focused fusion of camera-imu-uwb for accurate and drift-reduced localization. *IEEE Robotics and Automation Letters*, 6(2):1678–1685, 2021.

[58] Ty Nguyen, Kartik Mohta, Camillo J Taylor, and Vijay Kumar. Vision-based multi-mav localization with anonymous relative measurements using coupled probabilistic data association filter. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3349–3355. IEEE, 2020.

[59] NYC Metropolitan Transportation Authority. Nyc mta turnstile data, 2024. [Online]. Available: https://data.ny.gov/browse/select_dataset?category=Transportation.

[60] Fredrik Olsson, Jouni Rantakokko, and Jonas Nygårds. Cooperative localization using a foot-mounted inertial navigation system and ultrawideband ranging. In *2014 International conference on indoor positioning and indoor navigation (IPIN)*, pages 122–131. IEEE, 2014.

[61] Zhongmin Pei, Zhidong Deng, Shuo Xu, and Xiao Xu. Anchor-free localization method for mobile targets in coal mine wireless sensor networks. *Sensors*, 9(4):2836–2850, 2009.

[62] Chunyi Peng, Guobin Shen, Yongguang Zhang, Yanlin Li, and Kun Tan. Beepbeep: a high accuracy acoustic ranging system using cots mobile devices. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 1–14, 2007.

[63] Raspberry Pi. https://www.raspberrypi.com/products/raspberry-pi-3-model-b/, 2023.

[64] Yihong Qi. *Wireless geolocation in a non-line-of-sight environment*. Princeton University, 2003.

[65] Qorvo. https://www.mouser.com/datasheet/2/412/DWM1000_Data_Sheet-1950396.pdf, 2016.

[66] Qorvo. https://www.qorvo.com/products/d/da008334, 2021.

[67] Hadi Jamali Rad, Alon Amar, and Geert Leus. Cooperative mobile network localization via subspace tracking. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2612–2615. IEEE, 2011.

[68] Fatma Raissi, Sami Yangui, and Frederic Camps. Autonomous cars, 5g mobile networks and smart cities: Beyond the hype. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 180–185. IEEE, 2019.

[69] Niranjini Rajagopal, John Miller, Krishna Kumar Reghu Kumar, Anh Luong, and Anthony Rowe. Improving augmented reality relocalization using beacons and magnetic field maps. In *2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8. IEEE, 2019.

[70] Andreas Savvides, Chih-Chieh Han, and Mani B Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 166–179, 2001.

[71] Judy Scott and Carlton Scott. Drone delivery models for healthcare. 2017.

[72] Yi Shang, Wheeler Rumi, Ying Zhang, and Markus Fromherz. Localization from connectivity in sensor networks. *IEEE Transactions on parallel and distributed systems*, 15(11):961–974, 2004.

[73] Yi Shang and Wheeler Ruml. Improved mds-based localization. In *IEEE INFOCOM 2004*, volume 4, pages 2640–2651. IEEE, 2004.

[74] Chong Shao, Bashima Islam, and Shahriar Nirjon. Marble: Mobile augmented reality using a distributed ble beacon infrastructure. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 60–71. IEEE, 2018.

[75] Wang Shule, Carmen Martínez Almansa, Jorge Peña Queralta, Zhuo Zou, and Tomi Westerlund. Uwb-based localization for multi-uav systems and collaborative heterogeneous multi-robot systems. *Procedia Computer Science*, 175:357–364, 2020.

[76] Gregory G Slabaugh. Computing euler angles from a rotation matrix. *Retrieved on August*, 6(2000):39–63, 1999.

[77] Yang Song, Mingyang Guan, Wee Peng Tay, Choi Look Law, and Changyun Wen. Uwb/lidar fusion for cooperative range-only slam. In *2019 international conference on robotics and automation (ICRA)*, pages 6568–6574. IEEE, 2019.

[78] Francesco Sottile, Maurizio A Spirito, Mauricio A Caceres, and Jaron Samson. Distributed-weighted multidimensional scaling for hybrid peer-to-peer localization. In *2010 Ubiquitous Positioning Indoor Navigation and Location Based Service*, pages 1–6. IEEE, 2010.

[79] Benjamin Thompson and Noah Baker. The floating sensors inspired by seeds. *Nature*, 2021.

[80] Sebastian Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[81] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. {FarmBeats}: an {IoT} platform for {Data-Driven} agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, 2017.

[82] Saipradeep Venkatraman, James Caffery, and Heung-Ryeol You. A novel toa location algorithm using los range estimation for nlos environments. *IEEE Transactions on Vehicular Technology*, 53(5):1515–1524, 2004.

[83] Vicon. https://docs.vicon.com/display/Vantage/V8+camera+specifications, 2023.

[84] Viktor Walter, Nicolas Staub, Antonio Franchi, and Martin Saska. Uvdar system for visual relative localization with application to leader–follower formations of multirotor uavs. *IEEE Robotics and Automation Letters*, 4(3):2637–2644, 2019.

[85] Chen Wang, Handuo Zhang, Thien-Minh Nguyen, and Lihua Xie. Ultra-wideband aided fast localization and mapping system. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 1602–1609. IEEE, 2017.

[86] Jiadai Wang, Jiajia Liu, and Nei Kato. Networking and communications in autonomous driving: A survey. *IEEE Communications Surveys & Tutorials*, 21(2):1243–1274, 2018.

[87] John Wang and Edwin Olson. Apriltag 2: Efficient and robust fiducial detection. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4193–4198. IEEE, 2016.

[88] Pengfei Wang and Yufeng Luo. Research on wifi indoor location algorithm based on rssi ranging. In *2017 4th International Conference on Information Science and Control Engineering (ICISCE)*, pages 1694–1698. IEEE, 2017.

[89] Eric W Weisstein. Euler angles. https://mathworld.wolfram.com/, 2009.

[90] Henk Wymeersch, Gabriel Rodrigues de Campos, Paolo Falcone, Lennart Svensson, and Erik G Ström. Challenges for cooperative its: Improving road safety through the integration of wireless communications, control, and positioning. In *2015 international conference on computing, networking and communications (ICNC)*, pages 573–578. IEEE, 2015.

[91] Hao Xu, Luqi Wang, Yichen Zhang, Kejie Qiu, and Shaojie Shen. Decentralized visual-inertial-uwb fusion for relative state estimation of aerial swarm. In *2020 IEEE international conference on robotics and automation (ICRA)*, pages 8776–8782. IEEE, 2020.

[92] Hao Xu, Yichen Zhang, Boyu Zhou, Luqi Wang, Xinjie Yao, Guotao Meng, and Shaojie Shen. Omni-swarm: A decentralized omnidirectional visual–inertial–uwb state estimation system for aerial swarms. *IEEE Transactions on Robotics*, 38(6):3374–3394, 2022.

[93] Boxin Zhao, Zongzhe Li, Jun Jiang, and Xiaolin Zhao. Relative localization for uavs based on april-tags. In *2020 Chinese Control And Decision Conference (CCDC)*, pages 444–449. IEEE, 2020.

[94] Minghui Zhao, Tyler Chang, Aditya Arun, Roshan Ayyalasomayajula, Chi Zhang, and Dinesh Bharadia. Uloc: Low-power, scalable and cm-accurate uwb-tag localization and tracking for indoor applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(3):1–31, 2021.

[95] Kan Zheng, Qiang Zheng, Haojun Yang, Long Zhao, Lu Hou, and Periklis Chatzimisios. Reliable and efficient autonomous driving: the need for heterogeneous vehicular networks. *IEEE Communications Magazine*, 53(12):72–79, 2015.

[96] Thomas Ziegler, Marco Karrer, Patrik Schmuck, and Margarita Chli. Distributed formation estimation via pairwise distance measurements. *IEEE Robotics and Automation Letters*, 6(2):3017–3024, 2021.

Towards Energy Efficient 5G vRAN Servers

Anuj Kalia *Nikita Lazarev*[†] *Leyang Xue*[‡] *Xenofon Foukas* *Bozidar Radunovic* *Francis Y. Yan*[⋆]
Microsoft [†]*MIT* [‡]*University of Edinburgh* [⋆]*Microsoft and UIUC*

Abstract

We study the problem of improving energy efficiency in virtualized radio access network (vRAN) servers, focusing on CPUs. Two distinct characteristics of vRAN software—strict real-time sub-millisecond deadlines and its proprietary black-box nature—preclude the use of existing general-purpose CPU energy management techniques. This paper presents RENC, a system that saves energy by adjusting CPU frequency in response to sub-second variations in cellular workloads, using the following techniques. First, despite large fluctuations in vRAN CPU load at sub-ms timescales, RENC establishes safe low-load intervals, e.g., by coupling Media Access Control (MAC) layer rate limiting with CPU frequency changes. This prevents high traffic during low-power operation, which would otherwise cause deadline misses. Second, we design techniques to compute CPU frequencies that are safe for these low-load intervals, achieved by measuring the slack in vRAN threads' deadlines using Linux eBPF hooks, or minor binary rewriting of the vRAN software. Third, we demonstrate the need to handle CPU load spikes triggered by control operations, such as new users attaching to the network. Our evaluation in a state-of-the-art vRAN testbed shows that our techniques reduces a vRAN server's CPU power consumption by up to 45% (29% server-wide).

1 Introduction

Virtualized RANs, which run the cellular radio stack on commodity servers instead of specialized hardware, are gaining adoption in modern cellular networks (e.g., 5G), owing to advantages such as a multi-vendor ecosystem, easier maintenance, and faster feature upgrades. This paper targets the energy efficiency of vRANs, focusing on the "Distributed Unit" (vDU) servers that perform real-time base station functions. Base stations are a major energy consumer: For instance, a recent report from China Mobile [49] estimates the deployment of 2.1 million 5G base stations (roughly three sectors/cells per base station) in China alone. Given today's 240-watt vRAN servers (§3.2), even a modest 1% reduction in server energy usage can translate to ∼44 million kWh/year.

Two unique attributes of vRAN software—stringent real-time deadlines and black-box nature—preclude the use of standard CPU energy saving techniques. The lower vRAN layers running in the DU have hard deadlines on the order of the sub-ms Transmission Time Interval (TTI) of the wireless protocol, which must be met for functional correctness. Conventional energy-saving techniques, e.g., OS-controlled CPU frequency scaling or deep CPU sleep states, operate at tens of milliseconds and fail to meet these deadlines, with consequences as extreme as crashes or malfunctioning of the vRAN software. As a result, today's vRAN deployments disable these optimizations and instead maintain the CPU at a consistently high frequency regardless of the traffic [45].

This paper presents RENC, an energy-saving system that dynamically adjusts the DU's CPU frequency in response to sub-second changes in cellular workloads. Our design is based on the observation that cellular networks experience low utilization during most of their operational time. This leads to significant intervals of low CPU load where there is slack in the deadlines, i.e., a large deadline fraction remains unused, and there is room to reduce CPU frequency.

The central challenge for CPU energy saving in vRANs is the high variability in CPU load at sub-ms timescales. A sequence of TTIs that need few CPU cycles may be followed by a TTI with the maximum load, e.g., due to a traffic burst or a control plane operation. If the system is in low-power mode at this time, the DU can miss its deadlines with severe consequences (Section 2.1). The central idea in RENC is to establish "low-load" intervals for energy saving. We define "low-load" in terms of cellular traffic, as periods with data traffic below a small threshold (e.g., 10% or 1%), and with no expensive control plane operations. RENC needs to (1) separate these intervals from the opposite "high-load" intervals, and (2) determine safe low CPU frequencies for the low-load intervals. In high-load intervals, RENC keeps the CPU at a high frequency to ensure that deadlines are met.

For separation, RENC uses both proactive and reactive techniques, as well as careful timing. For example, we use the MAC scheduler for proactive rate limiting to prevent

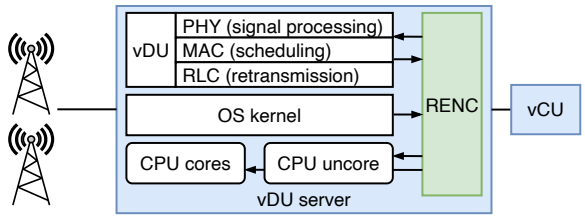


Figure 1: Overview of a vRAN deployment with RENC.

traffic bursts until RENC switches to high-load mode. RENC carefully orchestrates the timing of changes in MAC rate limits and CPU frequency, e.g., by ensuring that the increased CPU frequency is effective before lifting the rate limit. In addition to tackling user traffic-related CPU load, we also recognize control plane operations, such as user equipment (UE) attachments, as a cause of CPU load spikes. These can be handled in a reactive manner due to the longer control-plane latencies. RENC detects control plane messages using a small amount of telemetry from the vRAN and temporarily boosts CPU frequency to handle them.

The next challenge is to determine safe low CPU frequencies for the low-load intervals. This requires first measuring the "deadline slack" for each core, defined as the minimum unused fraction of the deadline. This is challenging because vRAN software is typically proprietary and closed-source, delivered to cellular operators by specialized vendors as a black-box binary. It is difficult for operators to directly add slack measurement code, especially since this must be done separately for each vendor. Instead, RENC aims to measure slack transparently to the extent possible: For interrupt-driven threads, we use eBPF [32] programmability in the Linux kernel scheduler, with no source code knowledge. For polling-based threads that do not yield to the OS, we use binary rewriting for a few DU functions. When the slack measured during low-load intervals remains high (e.g., above 10%) over a large number of samples, RENC chooses an iteratively lower CPU frequency for these intervals.

Inter-operability of RAN applications (such as energy-savings [6, Sec. 4.21]) across vRAN implementations is a key tenet of RAN virtualization and the "O-RAN" standards [4], which most vRAN vendors are adopting. The principle is to rely on interfaces exposed by vRAN components instead of internal details, to make wide deployment feasible. For this reason, we design RENC to run outside the vRAN software, and interact with it via well-defined interfaces.

Our evaluation in a commercial-grade 5G vRAN testbed, with two 5G 100 MHz 4x4 cells and nine commercial 5G UEs, shows that during periods of low traffic, RENC reduces the power consumption of vRAN CPUs and servers by up to 45% and 29%, respectively.

2 Background

Figure 1 shows a high-level overview of a vRAN deployment with RENC, focusing on the DU server that our work targets. It consists of one or more (typically three, corresponding to a three-sectored cell site) radio units (RUs) that connect to the DU via an Ethernet fronthaul link. For sub-ms latency, the DU server is close to the cell site, in a cabinet or a small nearby datacenter. Multiple (e.g., 100s) DU servers connect to a "centralized unit" (CU) server, which handles the non-realtime, less compute-intensive layers of the vRAN stack.

The DU runs the lower layers of the vRAN stack: the Physical (PHY) layer that does wireless signal processing converting signal samples to bits and vice-versa, the Media Access Control (MAC) layer that schedules wireless resources, and the Radio Link Control (RLC) layer that handles reliable in-order delivery etc. RENC runs externally to the DU as a userspace agent, with an in-kernel eBPF component.

Sources of energy consumption Cellular networks consume a huge amount of energy, of which the RAN consumes a majority (e.g., GSMA estimates 73% [40]). Within the RAN, the RU and the DU consume the most energy. The energy split between the two is variable, ranging from near-equal for small radios, to 9:1 for large massive MIMO radios [31,49]. For example, our DU server consumes around 236 W, whereas each of our 4x4 indoor radios (one server can theoretically handle six such radios) consumes around 45 W when active. This works focuses on the DU server, whose energy efficiency is a topic of wide industry interest [37, 39, 42, 43, 54, 59, 63]. RU energy is complementary to our work, which we plan to tackle in the future. RUs require different types of optimizations, e.g., achieving large savings requires turning off analog components (e.g., via MIMO- and cell-sleep [24]), which are controlled at longer timescales (hours) than the sub-second variations that our work targets.

2.1 Distinct characteristics of the vRAN DU

Real-time deadlines. Unlike typical applications—which are the focus of most prior work on energy efficiency (cf. survey [50])—where work can be queued-up if CPU power is insufficient, the vRAN DU has strict real-time deadlines. The deadline for most threads is the TTI, which is 500 µs in 5G deployments in sub-6 GHz bands, which are vRAN's focus today; it is even lower (125 µs) in mmWave deployments. The computation needed per TTI is highly variable, and can change from near-zero to maximum in consecutive TTIs. The consequences of violating deadlines range from alarming (e.g., dropped calls), to catastrophic (e.g., a cell going offline due to the vRAN software crashing). This requires that energy management be done without affecting the vRAN's real-time performance.

Black-box nature. Although energy efficiency for real-time systems is well-studied [21, 58], the vRAN software's proprietary and closed-source nature makes it difficult to simply apply existing techniques. The basic method from this literature is to measure, at a given CPU speed, the task's deadline *slack*, i.e., the minimum unused fraction of the realtime dead-

line. The CPU speed can be reduced while the slack stays above a threshold. For example, if tasks use only 100 µs of a 500 µs deadline in the worst case (i.e., 80% slack), CPU speed can be safely reduced by a small amount, and the slack re-evaluated. While the high-level method is simple, applying it to vRANs presents several challenges (§4.1).

2.2 Techniques in traditional base stations

In traditional base stations (e.g., using in-house DSPs [13]), energy efficiency has been extensively studied for over a decade (cf. surveys [64, 65]). The problem is less explored for vRANs, which are a recent advancement. Non-virtualized base stations save energy by opportunistically turning off different hardware components. Debavilli et al. [29] provide a power model for different base station sleep modes, characterized by the time to enter and exit the mode, ranging from sub-100 µs to one second, and the power consumption in the mode. Base stations based on specialized hardware may have proprietary and tightly hardware-integrated sleep mode implementations that are co-designed for RAN processing [22, 24]. In contrast CPUs have only general-purpose sleep modes, which are not easily applicable to vRANs (§3.2).

2.3 CPU and OS features

CPU energy management consists of mainly dynamic frequency scaling (P-states), and CPU sleep states (C-states). Today's vRANs typically disable these features (e.g., FlexRAN [45]), consuming high power at all times [27]. RENC uses P-states and shallow sleep states (§3.2).

eBPF in the kernel. RENC uses eBPF to transparently measure deadline slack for some vRAN threads. eBPF [32] is a modern framework for running user-provided programs called codelets at predefined code points (called *hooks*) in unmodified binaries. eBPF codelets are written in a restricted C-like language, and are compiled to a bytecode that is verified for safety and then JIT-compiled to native code for high performance. The typical use case for eBPF is to augment the Linux kernel with new functionality. The recent Janus system [36] uses vendor-provided eBPF hooks in vRAN binaries to add custom functionality to the vRAN.

3 Motivating observations

3.1 Low utilization in cellular networks

Cell sites have low utilization, as a natural consequence of the physical cell tower infrastructure being dimensioned for peak traffic separately in each area. Today's vRAN servers also run at a CPU capacity sufficient to handle the maximum traffic at all times, frequently wasting energy.

Although large-scale traces of cell site traffic at sub-second granularity are not publicly available today, several studies provide evidence for this. For example, an Ericsson report on RAN energy efficiency for LTE concludes that *"even a future network, where hotspot peak-hour traffic might be 1,000 times greater than today, will still feature low traffic loads most of the time and in most locations." [57].* Another report from Ericsson mentions that *"Half of the sites have long periods of just idling and waiting for users to make use of the capacity available."* [24]. In fact, a study of RU energy efficiency by SK Telecom and NTT Docomo evaluates the power draw of only idle RUs, since that is the common case [31].

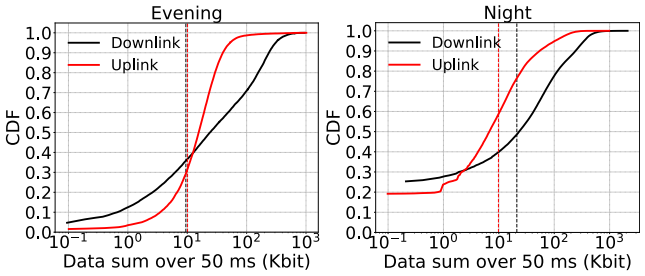


Figure 2: CDF of cumulative transport block size (TBS) measured every 50 ms, for two LTE cells. Vertical lines show 1% of peak traffic.

	C1	C1E	C6	P-states
Residency time	1 µs	4 µs	600 µs	N/A
Wakeup time	1 µs	4 µs	170 µs	N/A
Applicable to vRANs	✓	✓	?	✓

Table 1: Comparison of C-/P-states for our Ice Lake CPUs

To illustrate this, we collected TTI-level traces from two commercial LTE cells, using the Falcon LTE sniffer [34]. Since RENC waits for 50 ms to transition to low-load (§4), we plot the CDF of the total traffic measured every 50 ms in Figure 2. The left figure shows a 9:30–9:47pm trace for cell #1, located in a major university; the right for 10:11pm–0:11am in cell #2, located in a residential area close to a different university. These are busy locations, and we expect that these evening/night measurements approximate typical cells during day hours. In cell #1, over 50% of the 50 ms intervals have less than 1% of the peak traffic; such periods reach 60–80% in cell #2. These intervals represent opportunities for RENC to save energy. We currently target durations with <1% traffic, but relaxing this to 10% is possible in our system (§9.1).

3.2 CPU power management

Choosing between P- and C-states. RENC uses P-states and shallow C-states, but not the deep sleep states. Modern Intel server processors support C1, C1E, and C6 states. C1 and C1E are shallow and save far less energy than C6, but their sub-5 µs latencies make them easily applicable to vRANs.

Can RENC use the deep C6 state? Table 1 shows why this is difficult. A C-state's "residency time" is defined as the minimum time that the core must stay in that state to save energy, to break-even from the sleep and wakeup overheads. For our Ice Lake CPUs, C6's high 600 µs residency time—required to flush the large L2 cache—plus its 170 µs wakeup time, exceeds the sub-500 µs TTI used in 5G networks. Most vRAN threads must run in every TTI—this pattern is followed by all six

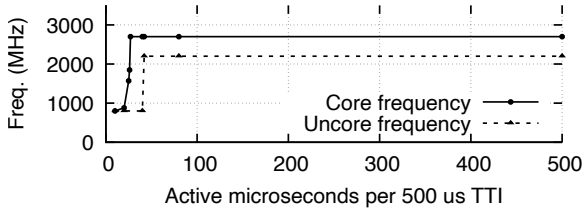
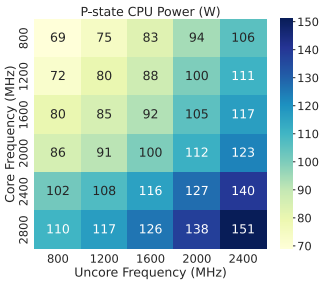


Figure 3: HWP-picked frequency for varying TTI activity levels

Component	Power
CPU (Xeon 6338N)	129 W
DIMMs (8x 32 GB)	14 W
Miscelaneous	93 W
Total	236 W

(a) Server power breakdown.



(b) Effect of core & uncore frequency

Figure 4: Components of vRAN server and CPU power

DUs that we have studied, including open-source [9, 15, 30], and proprietary [16, 33, 46]—so the CPU cores cannot remain inactive in C6 for long enough to save energy. We therefore focus on P-states instead, which allow the CPU to be active at all times and fine-grained frequency control. Recent advances reduce the impact of CPU frequency changes on core blocking time [28], and jitter on other cores [44]. C6 may be feasible for some DU threads with multi-TTI deadlines; we discuss this and its limitations in Appendix A.

CPU firmware controlled P-states. Can we simply let the CPU scale its own frequency? This does not work because the CPU's general-purpose P-state control loop is (1) too slow, and (2) not vRAN-aware. In modern CPUs, firmware-based frequency control (e.g., Intel's Hardware P-states [3], or HWP), is preferred over the older OS-based control. The firmware's simple 10 ms+ control loops are effective for general workloads, but vRANs require control at 1 ms timescales. For core P-states, we measured that HWP takes 60 ms to react after a sudden spike in CPU utilization; Schone et al. show that the uncore's reaction time is ~10 ms [60].

Figure 3 shows the CPU core and uncore frequency chosen by our server's HWP, (§ 8) in an experiment where one CPU core is active for varying fractions of a 500 µs TTI, and is asleep for the rest. We find that with only HWP picks the maximum core and uncore frequency with just 27 µs and 42 µs of activity, respectively. In contrast, RENC can recognize this thread as having high slack (over 90% of the TTI is unused), and choose a lower frequency.

CPU power breakdown. RENC focuses on CPU power since it is the largest component of DU server power. Figure 4a shows a wattage breakdown of our commercial-grade vRAN server (used in real vRAN deployments), when running two cells with no traffic and no energy management

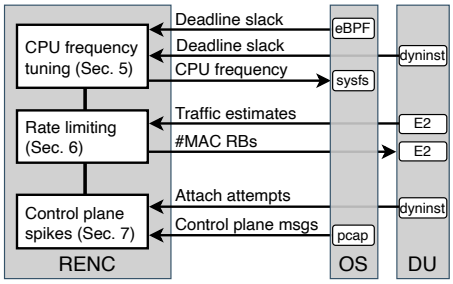


Figure 5: Overview of RENC's operation

features enabled (details in §8). The CPU consumes the largest share of the power, eclipsing the power consumption of typically power-hungry components like DRAM. Our server's Baseboard Management Controller (BMC) not provide a breakdown of the "Miscellaneous" component, which divides among the motherboard, devices (NIC, PHY accelerator, NVMe SSDs), cooling fans, and conduction loss [17]. This remains fairly constant in our experiments regardless of user traffic, so we ignore it.

Uncore frequency. In addition to CPU cores, the CPU's "uncore"—the L3 cache, memory controller, PCIe controller, etc—is a major power consumer, since it represents a large fraction of the CPU's die area (e.g., over 50% [26]). Figure 4b shows the CPU's power draw at different core and uncore frequencies. Both are significant, so RENC tunes both.

4 Design overview

We target three design goals for RENC.

G-1. Minimize energy during low-load periods. Since most cell sites have low utilization most of the time (Section 3.1) RENC targets saving energy during these periods. We leave energy saving during high load to future work.

G-2. Transparency to vRAN software and platform. Since vRAN software is often proprietary and multi-vendor, and provided by a different vendor than the hardware/OS vendor, we design RENC following O-RAN [4] principles of clear interfaces that require only minimal information from the vRAN and platform. Specifically, we need only (1) names and deadlines of realtime DU threads, (2) the signatures of key functions in poll-mode vRAN threads, and (2) low-latency access to standard MAC layer metrics and control knobs.

G-3. Low resource overhead. RENC must consume few CPU resources, else it will negate some energy savings. In particular, RENC must not require a dedicated CPU core, which would reduce cores available for the DU.

4.1 Overview and challenges

C-1. Finding slack. When measured simply across all time, there may be no slack in some DU threads. Here we explain this intuitively, and show it experimentally in §9. Figure 6 shows three TTIs (#1, #2, and #n) for a DU thread with a one-TTI deadline. TTIs #1 and #2 belong to RENC-controlled

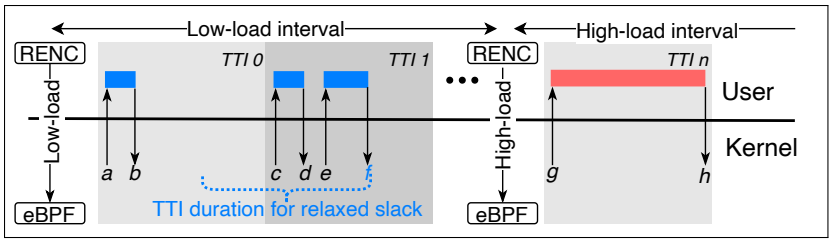


Figure 6: In this timeline for an interrupt-driven DU thread, RENC's eBPF program updates the relaxed slack for low-load intervals (blue) by summing over the dotted TTI-duration window at time f, i.e., $TTI - ((f-e)+(d-c))$. The userspace RENC agent tells the eBPF program the current low/high load type, so the latter ignores $TTI\ n$ with zero slack.

low-load intervals, by their low traffic rate and no expensive control plane operations. For these TTIs, the DU finishes work well before the deadline. In contrast, processing in the high-load TTI #n takes almost the entire deadline duration.

The key observation is that if we compute slack across all TTIs #1–#n, we will find no slack, i.e., the minimum unused deadline fraction is near-zero because of TTI #n. Without slack, RENC cannot reduce CPU frequency without risking deadlines, especially uncore frequency which affects all cores. However, if we carefully separate the low- and high-load intervals, we may find significant slack in the former. RENC introduces new techniques to (1) establish low-load intervals, (2) measure slack for precisely these intervals, and (3) limit energy saving to them. For low-load intervals, RENC iteratively computes the lowest safe frequency for each CPU core and the uncore. For high-load intervals, RENC runs the CPU at the highest configured frequency.

C-2. Separating low and high CPU load intervals. How can RENC ensure that the DU does not admit more load than the CPU can handle at low frequency? Prior work partially tackles this problem, e.g., PR3 [25], vrAIn [20] and CRT [55] use MAC-layer rate limiting to adjust traffic to match CPU frequency. However, these are best-effort, with no deadline guarantees that are crucial in commercial vRANs. For example, unlike less-performant open-source vRANs used in prior work [9, 15], our PHY (Intel's FlexRAN) is optimized to keep intermediate computation buffers sufficient for only the deadline period, minimizing its cache footprint for efficiency. Upon missing a deadline, there is no option but to crash the program. There are recent proposals that aim to alleviate this problem [38], but these require re-designing the DU.

Our solution is to couple MAC rate limiting with CPU frequency changes, by strictly ordering them. Before reducing frequency, RENC first applies a MAC layer rate limit to disallow sudden traffic bursts (§6). Similarly, RENC lifts the rate limit only after fully applying a CPU frequency increase. RENC also transitions to the high CPU load state on detecting CPU-intensive control-plane messages (§7).

C-3. Mostly-transparent slack measurement. Cellular operators typically receive the DU as a binary from a vendor, making it challenging to measure slack. We aim to infer slack externally as much as possible, for the two reasons. First, external measurement works generally for all DU implementations, increasing RENC's applicability to the diverse vRAN ecosystem. Second, some cores may run multiple realtime DU threads and have interference from unavoidable OS threads, in which case slack is best measured externally.

We have created two new techniques to measure slack, which rely on minimal information from the vendor that is easily available. First, we observe that the OS provides a vantage point to observe the activity of DU threads that yield to OS in every deadline interval. In the DU that we use (Section 8), five out of the seven types of realtime DU threads follow this pattern. For these threads, RENC measures slack with no code information from the vendor except thread names and deadline durations. Our insight is to use eBPF hooks in the Linux scheduler for this, since standard OS mechanisms are too coarse (§5).

For DU threads that do not yield to the OS, we require the function names and signatures of the realtime DU functions. Usually there is one top-level function that encapsulates the DU's computation in a deadline interval. RENC then uses binary rewriting to capture the start and end of these functions to measure the per-thread slack.

5 Measuring deadline slack

This section describes how RENC measures the slack for low-load intervals at runtime, while excluding high CPU load intervals whose slack can be zero; the next two sections describe how RENC keeps low-load intervals free from high load. RENC then uses slack measurements to iteratively reduce uncore and CPU cores' frequency until slack drops below a threshold (default 10% in our implementation) (§5.4).

In contrast to static measurement in prior work, RENC measures slack and tunes CPU frequency at runtime. For example, CRT [55] requires manual experiments to compute safe frequencies for different traffic loads by noting the CPU frequency at visible errors occur, e.g., UE disconnections and errors in the vRAN code. RENC provides an automated and principled solution to this problem. The heterogeneity of vRANs further complicates static approaches: vRAN sites may run different RAN configurations (e.g., different number of cells or wireless configuration parameters) with different server hardware, which change the CPU requirements.

5.1 Types of vRAN threads

RENC requires some basic information about the DU software from the RAN vendor, including whether the thread is interrupt-driven or busy-polling, and the thread's deadline. Table 5 shows our DU' threads, their types, and their

deadlines. For busy-polling threads, RENC also requires the names and signatures of the task functions (typically one per thread) executed by the thread. In our DU, we use three functions corresponding to three PHY layer thread types: a function called at the start and end of each TTI; a function that handles fronthaul Ethernet packets; and a function for processing accelerator and shared-memory queues.

We develop two different approaches to measure slack.

Interrupt-driven threads are woken up by the OS at the start of each TTI, e.g., via a timer interrupt, or a semaphore from another thread. The thread performs its tasks for the TTI (e.g., MAC scheduling) and then yields. All MAC- and RLC-layer in our DU fall in this category. RENC uses eBPF hooks in the OS scheduler to measure their slack (§5.2).

Busy-polling threads run completely in userspace, i.e., they do not yield to the OS scheduler and instead constantly poll for work. In our DU, the PHY layer threads fall in this category, likely because of the PHY's high compute and efficiency requirements. We use binary rewriting for a few functions to measure the slack for these threads (§5.3).

5.2 Handling interrupt-driven threads

Can we simply use standard OS mechanisms to measure slack for these threads? This is not possible today, since the OS-exposed time measurements are too coarse, using "jiffies" (10 ms) as the unit of time. Our insight here is that eBPF provides the required fine-grained kernel programmability. To our knowledge, RENC is the first system to use eBPF to track thread execution time for energy saving, and we believe that it is a promising direction for non-vRAN systems as well.

In-kernel eBPF programs consist of simple, provably-safe code that is JIT-compiled to native machine code. The program has access to in-memory maps to store data, data structures, and high-resolution timers. RENC uses the eBPF hook at the Linux scheduler's `sched_switch` function, which is called whenever the thread running on a CPU core changes. Using the example in Figure 6 (see the caption for a detailed explanation), `sched_switch` can capture all user-kernel transitions, indicated by the arrows labeled with timestamps a–h. The correct low-load slack for the case shown is $TTI - ((f - e) + (d - c))$, since (1) the activity during *TTI 2* exceeds *TTI 1*'s activity ($= b - a$), and (2) the higher activity during *TTI n* ($= h - g$) must be ignored because it is in a high-load interval. Note that in practice we can have tens of active intervals per TTI, e.g., caused by the vRAN software's structure, as well as unavoidable kernel tasks such as IRQ and read-copy-update (RCU) pre-empting the DU threads.

5.2.1 Dividing between eBPF and userspace

What part of the slack measurement should run in eBPF within the OS, and what part in userspace? In RENC, we chose to implement slack measurement fully in eBPF, without any userspace involvement. The userspace RENC agent periodically (once every five seconds by default) retrieves the slack measurements from the eBPF program, and uses them to tune CPU frequency. This section describes the rationale behind this design, and how our implementation works.

An intuitive approach that we implemented first is to use a minimal eBPF program that simply passes all scheduling events to the RENC agent in userspace. This was convenient because all logic stays in userspace, easing development and debugging. However, we abandoned this approach because we found that sending messages to userspace for every scheduling event is fraught with issues. The compute overhead is a clear one: For example, with 10 thread switches per TTI (500 µs) across 10 DU cores, there are 200K kernel-user messages per second. This required dedicating a CPU core for the RENC agent to constantly poll an eBPF ring buffer for new messages, else messages are dropped.

We noticed more insidious issues, too. The high amount of kernel processing for message handling can get deferred behind the high-priority DU threads, causing subsequent bursts of kernel activity that cause the DU threads to miss their deadlines. Sending messages to userspace from a scheduler hook can also cause an infinite loop of messages leading to a kernel hang, since the act of sending a message can cause further scheduling events to handle the message.

5.2.2 In-kernel slack measurement

To avoid the issues with heavy scheduler-to-userspace messaging, we instead measure slack in the kernel. Surprisingly, we found that today there is no easy way for eBPF programs to know when a TTI starts and ends: At the time of writing, Linux's eBPF does not support the system wall-clock (`CLOCK_REALTIME`), supporting only monotonic clocks (e.g., `CLOCK_MONOTONIC`). The wall clock is the DU's heartbeat: it is synchronized with the RUs via the Precision Time Protocol (PTP), and is used to time TTIs. A patch to add wall clocks to Linux eBPF was submitted in 2020, but abandoned in part due to concerns about the likelihood of programmers' incorrect use of a non-monotonic clock [53].

Relaxed slack. Due to eBPF's lack of a TTI-synchronized clock, we use a conservative estimate of slack that does not rely on TTI boundaries, defined as follows (assuming a one-TTI deadline). We measure the maximum fraction in *any* TTI-length interval that the core is active in; the relaxed slack is one minus this fraction. The chosen interval may overlap TTIs (see Figure 6), making the relaxed slack a conservative estimate. This works well in practice because DU tasks run at the beginning of each TTI. For low-load intervals, the latter portion of TTIs is idle, so the selected interval will align with a TTI unless utilization is very high, in which case RENC does not attempt to save energy for this core.

To keep our implementation within the eBPF verifier's limits, we use a fixed-size array of intervals for each CPU core. Each interval is a pair of timestamps for when any thread was active, i.e., the kernel's idle task was not running. We deliberately chose to include *all* threads and not

just DU threads, to include OS thread overheads which can be significant in real systems, e.g., following a NIC driver update that causes interrupts to land on DU cores. When a thread switch occurs at time t, the eBPF program sums all intervals overlapping with $(t - TTI, t)$. If the sum exceeds the current maximum, the program updates the maximum deadline utilization and the minimum slack.

The userspace RENC agent reads the per-core slack values from the kernel only when tuning frequency, i.e., once every five seconds by default. The user-kernel message overhead is now negligible, a significant reduction from the per-scheduling-event rate. We manually ensure that the in-kernel eBPF codelet has few instructions, and measure its overhead as <5% of the context switch cost (§9).

Handling load type changes. Our eBPF program maintains two arrays of intervals per core, one for each load type. When the userspace RENC agent decides to change the load type (§6), it indicates this to the eBPF program using a syscall to update a BPF map entry. The vertical "low-load" and "high-load" arrows between RENC and eBPF in Figure 6 show these updates. On every `sched_switch` invocation, our eBPF program sums up the intervals in the array corresponding to the current load type, keeping the two separate. While measurements for high-load intervals are not necessary for RENC, we still keep them to aid debugging.

5.3 Handling busy-polling threads

We use the popular `dyninst` framework to instrument the call and return instructions of the main datapath functions of busy-polling DU threads. Additionally, we use an internal tool that allows inserting userspace eBPF programs at the instrumented code points. Since these programs can access the wall clock, slack measurement is straightforward.

In our DU, we use this approach for the busy-polling PHY layer threads, specifically (1) the PHY signal processing threads' top-level function called at the start and end of TTI processing (`ebbu_pool_event_set_state()` in FlexRAN), and (2) the fronthaul packet I/O threads' top-level per-symbol callback function (`sym_ota_cb()` in FlexRAN). For the PHY signal processing thread, we also need the description for the enum values of its argument, e.g., specifying whether TTI processing is starting or ending. We chose to not instrument one PHY thread type ("PHY SHM rings" in Table 5) because it splits its work across four functions, making it cumbersome to instrument all of them though this can be done if needed. This thread's processing is largely independent of the traffic load, so we manually confirm that it can safely run at the lowest frequency.

5.4 Iterative CPU frequency tuning

Collecting slack samples. At each CPU frequency configuration, RENC collects slack measurement samples for low-load intervals, filtering out high-load intervals. We continue collecting samples until the cumulative duration spent in low-load intervals exceeds a large value V, to try capturing the full range of CPU activity possible in low-load intervals. We expect V to be several hours in real deployments, but use a shorter 5-second interval for testing to speed convergence.

Prioritizing uncore frequency. After RENC has sufficient slack samples, it tries to reduce CPU frequency. We set our slack threshold to 10%, leaving some headroom for cases not covered by the observation period. We prioritize the uncore, since it has a large impact on power draw (§9.4) and affects all cores. If all cores have >10% slack, and uncore frequency is not at the minimum, RENC reduces it by the supported quantum (100 MHz). Otherwise, sys starts reducing CPU core frequencies for cores with >10% slack. Then RENC starts a new observation period.

6 Coupled CPU frequency and MAC control

This section describes how RENC coordinates changes in CPU frequency and MAC rate limits to separate low- and high-load intervals. Neither take effect immediately, so RENC must control their order and timing.

6.1 Forecasting DU traffic

RENC uses DU-exposed telemetry for uplink and downlink traffic to measure the current traffic demand, and determine when to transition to low or high load. Among the several metrics that can be used to measure traffic, we use the one that provides the earliest indication of traffic demand. This allows quickly removing the rate limit when there is a surge in traffic, minimizing the impact on RAN performance.

RENC requires low-latency access to DU telemetry and control interfaces that DU vendors typically already implement, e.g., for the O-RAN E2 interface [18]. Our DU vendor provides access to these via a local UDP port.

Uplink traffic estimation. We use the per-UE Buffer Status Reports (BSRs) to estimate the uplink load. UEs send BSRs to the DU to indicate buffered data size. These are sent *before* the UEs send the actual data, and hence provide an early indication of the uplink traffic demand.

Downlink traffic estimation. For downlink traffic, we use the throughput at the DU's interface with the CU. While this is not as early an indication as the BSRs, it is the earliest indication of downlink traffic that is available to the DU.

6.2 Low/high-load classification

Our current implementation simply uses recent traffic samples as indicators of future load, though more sophisticated techniques (e.g., time-series analysis) may be applied. We define $R_{dir,t}$ as the traffic sample received at time t, summed over all UEs and cells, for uplink or downlink direction *dir*. We also define M_{dir} as the statically-measured maximum traffic sample for each direction; in our DU, this is 150 kB for the cumulative BSR, and 450 Mbps for the downlink throughput.

RENC transitions to low-load if all samples in the last n milliseconds are below 1% of M_{dir}; $n = 50$ in our implemen-

tation, i.e., we transition to low-load if there is little traffic for roughly three RTTs on our network (Figure 2). RENC transitions to high-load when the latest traffic sample exceeds 1% of M_{dir}. By not waiting for multiple >1% samples, RENC minimizes its impact on user-visible performance, at the cost of possibly reduced energy savings, e.g., when the high traffic sample is a transient spike. Note that some impact is acceptable, e.g., Ericsson uses a "Low Energy Scheduler" (LESS) that packs data in the frequency domain to make it sparse in the time domain [23].

6.3 Rate-limiting

RENC controls the number of wireless resource blocks (RBs, each RB corresponds to a small piece of spectrum) allocatable by the MAC for rate limiting, achieved via 3GPP's slice control interface [11]; this resembles the "ECO-Mode" optimization in some radios, which can power-down amplifiers for some RBs during times of low traffic [23]. This basic approach can be improved by using more information such as signal-to-noise (SNR) ratio, which affects the number of bits per RB. RENC allows a small percentage of RBs (10% by default in our implementation) to be used during low-load intervals. This is larger than the 1% threshold for traffic metrics, to account for the variable bits per RB.

6.4 Ordering and timing of changes

The process of changing CPU core/uncore frequency, or applying a MAC rate limit, is not instantaneous; RENC must ensure that these changes are applied in the correct order.

1. **CPU frequency and MAC rate limiting:** RENC must prevent high traffic while the CPU is at a low frequency.
2. **OS eBPF and MAC rate limiting.** RENC's kernel eBPF program must not pollute low-load slack measurements with those of high-load intervals (§5.2.2).

We next discuss how we apply these changes, and their maximum latencies measured when the system is running at RENC's low CPU frequency (§8.1). We use Linux's `sysfs` and MSR (Model-Specific Register) interfaces to control CPU core and uncore frequencies, respectively. We measure the maximum delays for these as 1100 µs and 2300 µs, respectively, which match prior work [60]. We define the time to change CPU frequency as the higher of the two, i.e., d_{cpu} = 2300 µs. Our DU applies RB changes in the TTI after it receives the request; The RB change latency equals RENC's maximum's latency to the DU's control UDP port (600 µs), plus one TTI (i.e., d_{rb} = 1100 µs). We use a syscall to update the load type in a BPF map in the eBPF program. This is a blocking call that takes at most 15 µs (= d_{ebpf}).

Figure 7 shows how RENC orders these changes. When transitioning from low to high load, it first increases CPU frequency and sets the eBPF load type to high-load. It waits for $max(d_{cpu}, d_{ebpf})$ for both changes to take effect, then lifts the MAC rate limit. Note that this sequence allows our kernel eBPF program to account slack for low-load TTIs to the high-load type, which is safe since it does not pollute the low-load slack measurements that RENC cares about. To transition to low load, RENC first applies the MAC rate limit, and waits for d_{rb} for this to take effect. It then reduces the CPU frequency and sets the eBPF load type to low-load.

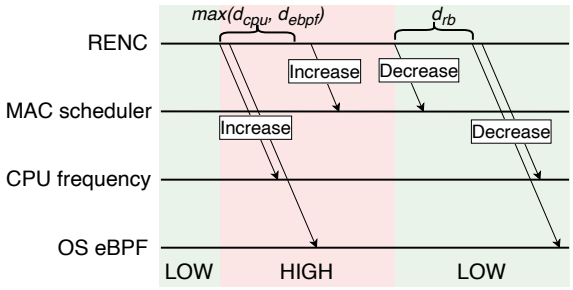


Figure 7: Ordering & timing of RENC's load-type transitions

Time	Event
34.448 s	Random access attempt from PHY (trigger)
34.469 s	CPU load spike; sudden slack decrease to 0%
48.515 s	UE Context Release Request from CU (trigger)
49.167 s	CPU load spike; sudden slack decrease to 68%

Table 2: Example of high load due to control messages

7 Control operation CPU spikes

The discussion until now has focused on handling variations in the DU threads' slack caused by changes in the traffic load. However, this alone may not ensure that the DU meets its deadlines: In addition to the user traffic, the DU's control plane processing (e.g., UE attachments) can also be a source of sudden bursts of compute load. The key challenge here is detecting these spikes and reacting to them in a timely manner, with few modifications to the vRAN software for transparency and ease of deployment.

7.1 Discovering spike-causing events

The following insight helps to systematically and transparently determine the causes of compute spikes related to control messages. DU control messages are carried on formally-specified interfaces, i.e., the Functional API (FAPI) between the MAC and PHY [1], and the F1AP interface between the CU and DU [12]. Control messages can therefore be observed by simply interposing on the interface, ignoring the internal complex details of the vRAN implementation. The messages can be captured along with sudden dips in the DU threads' slack (corresponding to a compute spike), and correlated to identify the causing message.

Table 2 shows example timeline for two sudden dips in a DU thread's slack when running a cell initially with no UEs attached, whose slack is initially 84%. These are caused by a (1) UE attaching and (2) detaching in our testbed (§8). The first dip is caused by a PHY-to-MAC Random Access

DU Server	HPE DL110 Gen10 "telco" server Xeon 6338N CPU (32c), 8 ×32 GB DDR4 DIMMs 1 TB NVMe SSD, Intel E810 NIC, ACC100 card [10]
Software	Realime Linux v6.1; Intel FlexRAN PHY [46] CapGemini DU and CU, commercial 5G core
5G cells	FoxConn 4×4 RUs at 3.5 GHz, 100 MHz bandwidth 30 KHz subcarrier spacing, TDD (DDDSU)
Lab #1	Two cells in building, nine (4 + 5) Raspberry Pi UEs 13 realtime DU CPU cores, FlexRAN version 22.11
Lab #2	One cell, one OnePlus 5G phone and Pi UE 11 realtime DU CPU cores, FlexRAN version 22.03

Table 3: Evaluation testbeds' hardware and software

Channel (RACH) Indication message, sent when a UE tries to attach. When this happens, the DU must initialize UE state, which causes the compute spike. The second spike is similarly caused by a F1AP UE Context Release Request message, which is sent by the CU to the DU when a UE detaches. Currently, we correlate events to spikes manually, but this can be automated using time series correlation methods. We expect that other events, such as UE handovers will also cause spikes, and leave this for future work.

7.2 Reacting to control messages at runtime

Control messages being delay-tolerant can be handled reactively, i.e., a proactive approach such as RENC's rate limiting of data traffic is not necessary. For example, in Table 2, the attachment-related spike happens 21 ms after the random access control message. The detachment related dip happens 435 ms after the corresponding control message.

On detecting a spike trigger, RENC transitions to the high-load state, and stays there for the maximum time between the trigger and the corresponding spike. To leave sufficient error margins, currently RENC stays in the high-load state for 200 ms after the attachment trigger, and for 1 second for the detachment trigger. This is a conservative choice that can be optimized. In our DU, UE detachments force high-load for a long time which is undesirable, but these are rare since UEs typically detach infrequently to minimize signaling, staying in an idle/inactive state [41] during periods of inactivity.

To detect FAPI RACH messages, we intercept the PHY's top-level call to its FAPI interface to the MAC, which uses a publicly-available shared library ("Wireless Service Library", or WLS [5]). This library has a public, well-defined interface for compatibility with different MAC layer vendors [2], allowing us to de-parse the messages list to detect RACH Indications. To detect spike-causing F1AP messages, we use Linux's packet capture (pcap) interface and decode them to find the target message types.

8 Evaluation

8.1 Experiment setup

We implemented RENC in 2500 source lines of code in C++ and C, and 100 lines of eBPF (`libbpf`) code. Table 3 summarizes our testbeds, which use state-of-the-art commercial-grade vRAN hardware and software that closely resembles real-world deployments [7]. To focus on the DU server, we run the CU and a commercial 5G core network on other servers. We use two labs, with different DU versions to test our design's generality. Lab #1 has two 5G cells on different building floors, with nine total Raspberry Pi UEs (four or five per floor) with Quectel RM500Q modems. We use the smaller lab #2 for experiments that need higher throughput, since it has better signal quality and modem. The small size of our testbeds is a limitation of our work, though lab #2's size is close to real deployments, which often have 10–20 active users per cell [56, Fig. 10]; Our cells can achieve 30 Mbps uplink and 500 Mbps downlink TCP throughput, matching commercial 5G sub-6 GHz networks [52, Fig. 6,7].

DU CPU core configuration. Our DU uses 13 realtime CPU cores for the two-cell configuration, and 11 for the one-cell configuration. Table 5 shows the latter; the two-cell configuration doubles the MAC layer cores. We did not design these configurations for RENC, but rather used the default stable configurations created over months with the vendors' help. The functional purpose of each thread is not important for RENC, but we include it for completeness. We minimize the energy effect of the unused DU server cores by setting them to the lowest 800 MHz frequency. Our default stable configuration disables Hyper Threading (HT), but RENC can be extended to support it by using the minimum of the two HT siblings' slack to tune the core's frequency.

Energy measurement. Our vRAN server's management interface (HPE iLO) provides full-server power measurements at coarse 10-second intervals. For fine-grained CPU power measurements, we use the CPU's RAPL (Running Average Power Limit) counters, which are accurate [48].

CPU frequencies. Our default server configuration before RENC used 3400 MHz for CPU cores, and 2400 MHz for the uncore, which we use as the baseline. RENC successfully separates-out low-load intervals that can run at the minimum CPU frequency that we allowed: 1000 MHz for all DU cores, and 800 MHz for the uncore. (Going further to 800 MHz for the CPU cores saves very little energy but costs 20% CPU performance.) We report per-core slack measurements, and the marginal impact of core and uncore frequencies in §9.

8.2 Idle-mode power consumption

We start by measuring RENC's idle-mode power (§3.1) savings, which is a key energy metric for RAN equipment [31]. We attach all nine UEs in lab #1, to the two cells, which then remain idle. We use three baselines: (1) the default configuration with no energy optimizations, (2) enabling C1 states, and (3) CPU's hardware-based automatic frequency scaling (HWP, §3.2). When using HWP, we set its "energy-performance bias" policy in the BIOS to favor energy. We ignore C1E states since they save negligible energy (< 1%) in

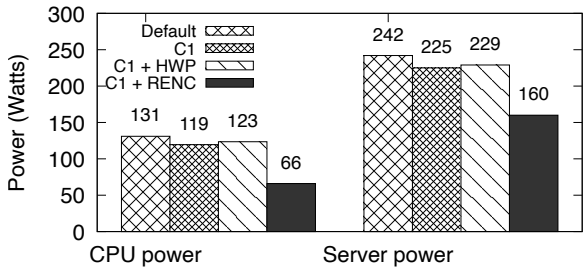


Figure 8: Idle-mode CPU and server-wide power consumption, with different energy saving configurations.

	DL Mbps	UL Mbps	Ping ms
Without RENC	486–520	29.6–29.7	27.1 (3.3)
With RENC	499–520	29.7–29.7	27.9 (3.4)

Table 4: Basic Internet performance without and with RENC

our experiments. Figure 8 shows that that enabling C1 states reduces CPU power from 131 W to 119 W, and server power from 242 W to 225 W. Enabling RENC on top of C1 further reduces CPU power by 45% to 66 W, and server power by 29% to 160 W, respectively.

Comparison with HWP. CPU power draw with HWP enabled (123 W) is 1.8x higher than with RENC. Counter-intuitively, HWP does not save power even when cells are idle, for two reasons: First, a vRAN being a realtime system creates a non-negligible amount of CPU activity even when idle. Our vRAN's realtime threads run every TTI, and do a small amount of computation which is nevertheless large enough to cause HWP to pick high core and uncore frequencies (Figure 3). Specifically, HWP runs all PHY cores and most MAC and RLC cores at too-high frequencies (2.5–2.7 GHz). Similarly, it picks a too-high uncore frequency (2200 MHz); artificially forcing the uncore (permitted via Model Status Registers even when HWP is active) to 800 MHz brings down CPU power to 83 W, which is still higher than RENC's 66 W by 26%. Second, enabling HWP *increases* power draw, because HWP (a) prevents us from fixing unused cores to 800 MHz, and (b) chooses slightly higher frequencies for these.

8.3 Energy savings with data traffic

Effect on network performance. RENC aims to save energy without significantly affecting network performance. Here we study high-level metrics, and run micro-benchmarks in §9. Since performance with multiple UEs is noisy due to spectrum sharing and interaction with TCP, we use lab #2 in this experiment. We use SpeedTest [8], and ensure that all tests use the same SpeedTest server. We run three tests both with and without RENC, and report the maximum and minimum values. Since SpeedTest's ping latency varies widely, we use a different ping application to measure the average and standard deviation of latency to cloudflare.com. Table 4 shows the results. The throughputs and latency of the two configurations are similar, which meets our goal.

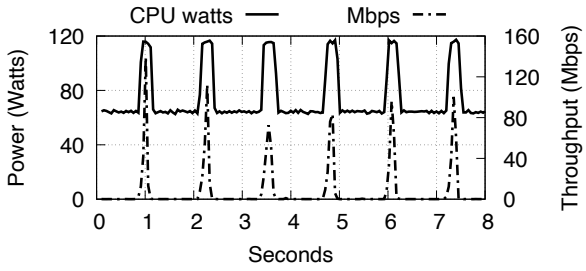


Figure 9: One UE repeatedly downloading a 1 MB file.

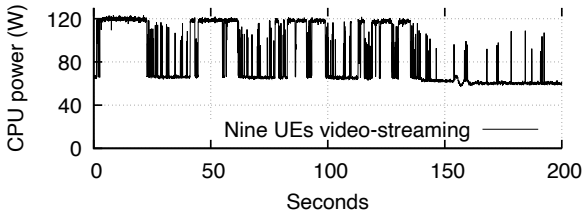


Figure 10: Energy saving with nine video-streaming UEs.

We next evaluate RENC in the larger lab #1 with real traffic.

File downloads. In this experiment, one Pi UE repeatedly downloads a 1 MB file from a lab server at 1 s intervals. Figure 9 shows the DU CPU's power draw measured at 10 ms intervals, along with downlink bandwidth. RENC successfully reacts to the load spikes by increasing CPU frequency, and reduces it when the load drops.

Video streaming. We use video streaming as an example application to show how RENC can save energy even during active user sessions. Here we use all nine Pi UEs, which stream a 720p video from a popular Internet site, using a headless mpv player [51]. The UEs start at roughly the same time. Figure 10 that, in the beginning, RENC keeps the DU server primarily in high-load mode as the UEs fetch the initial parts of the video. After the UEs buffer the video, there are gaps between video chunk downloads, during which RENC successfully saves energy. The average power draw with RENC is 83 W, compared to 121 W without (not shown).

Traffic mix. We use the nine UEs to create a traffic mix representative of a real-world workload: we run three video streams, two 1 MB file downloads and two 256 kB uploads at 1s intervals, two long-lived 1 Mbps iperf3 download streams, and one 1 Mbps iperf3 upload stream. Figure 11 shows how RENC saves energy during the traffic gaps, reducing average CPU power from 121 W to 109 W.

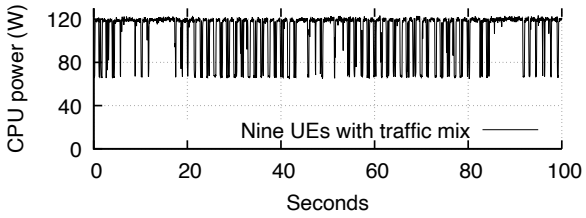


Figure 11: Energy saving with nine UEs running a traffic mix.

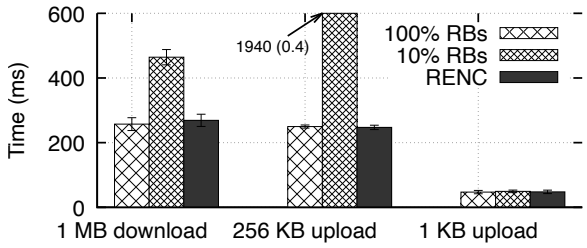


Figure 12: File transfer time with MAC rate limiting.

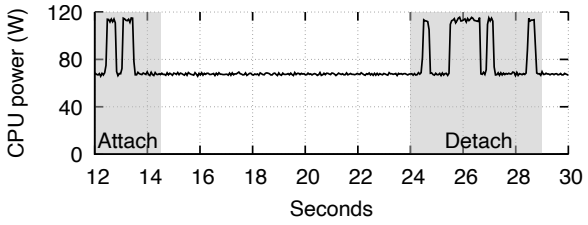


Figure 13: Handling of control message spikes.

9 Microbenchmarks

We next evaluate some aspects of RENC's design in isolation.

9.1 Effect of MAC rate limiting

To quantify how RENC's MAC rate limiting affects network performance, we measure the time for small file transfers that stress RENC's reactiveness; our SpeedTest experiment showed RENC's negligible impact on long transfers. Since RENC dynamically scales the allowed RBs between 10% and 100%, we show the download time with static 100% and 10% RB limits for reference. The experiment uses a simple Python HTTP client and server. Since downlink speed is higher than uplink, we use a larger 1 MB for downloads than uploads (256 kB). These are smaller than average mobile web page sizes today (e.g., 2 MB in one study [14]). Figure 12 shows that transfer time with RENC is close to the static 100% allocation (<2% higher), and better than the static 10% allocation. For small 1 kB uploads, common in IoT devices [62], all three perform similarly since the DU always stays under 10% RBs.

Sensitivity analysis. We have focused on a 1% threshold for traffic metrics, but this is tunable. Increasing it to 10% allows RENC to save more energy, but may affect user performance unless the algorithms in §6 are improved. For example, using a 10% BSR threshold in the 256 kB upload experiment increases the mean transfer time from 200 ms to 231 ms, due to the DU's slower reaction to the uplink traffic burst.

RU power. RENC's rate limiting may slightly increase the amount of time that the RU is active, increasing its power draw. We currently lack the precise RU power measurements needed to quantify this, but we expect it to be small since RENC lifts its rate limit in milliseconds in response to traffic.

9.2 Slack measurement

We conduct the following experiment to test RENC's establishment of low-load intervals that are safe for CPU frequency scaling. Since this experiment requires stressing the DU's compute resources, we use lab #2 to generate high traffic. We run two tests: without RENC, and with RENC, shown in the rightmost two columns of Table 5. In each test, we attach the phone, run two SpeedTest sessions, and then detach the phone. In the first case, the DU runs without RENC at a constant low frequency of 1000 MHz for all cores and 800 MHz for the uncore; this configuration frequently crashes the DU due to deadline violations, but we run it for this experiment. In the second case, RENC is enabled.

We make three observations (Table 5). First, without RENC, several DU threads have zero slack. The MAC scheduler thread has 8% slack, but that leaves insufficient safety margins. This confirms that this DU cannot run at this low frequency without RENC, since slack measured simply across all time is zero. Second, RENC successfully establishes low-load intervals with sufficient slack for frequency scaling. Third, some threads (e.g., the PHY threads and the RLC timer thread) have significant slack even without RENC, making it possible to statically configure them to run at a low frequency. ("PHY timer" has lower slack with RENC because its processing time depends more on CPU frequency than user traffic.)

Importantly, note that knowing whether a CPU frequency configuration is safe, or which threads have high slack, is difficult without our new slack measurement techniques. For example, simply running coarse-grained tool like `top` always shows <9% and <27% or the MAC-to-PHY and MAC scheduler threads, respectively, but their actual TTI deadline utilization without RENC reaches 100% and 92%, respectively.

9.3 Control message handling

Figure 13 shows the DU's CPU power draw when a Raspberry Pi UE attaches and detaches in lab #2. There are two high-load intervals during attachment because of a failed attempt followed by a successful one, which is common in cellular networks. The detachment event has one long high-load interval, and multiple three short ones caused by spurious attachment attempts that our UE makes when we turn on flight mode to detach it. This experiment shows how RENC successfully handles CPU load spikes from control processing, and handles unexpected behaviors of UEs. RENC also handles the more challenging case with nine UEs simultaneously attaching and then detaching.

9.4 Breakdown of energy savings

Table 6 shows that both uncore and CPU core frequency scaling contribute significantly to energy savings. Reducing the uncore frequency alone from 2400 MHz to 800 MHz reduces CPU power by 32% from 117 W to 80 W. Adding core frequency scaling reduces CPU power by a further 16% from 80 W to 67 W. A takeaway from our work is that uncore frequency scaling provides a convenient and effective knob for reducing CPU power. Since it is just one value, an operator

Thread name	Description	Running mode	Deadline	Slack w/o RENC	Low-load slack with RENC
F1 worker	Communication with CU	Interrupt-driven	1×TTI	0%	64%
RLC timer	RLC timekeeping	Interrupt-driven	1×TTI	62%	75%
RLC worker	RLC processing	Interrupt-driven	1×TTI	0%	76%
MAC scheduler	Wireless scheduler	Interrupt-driven	1×TTI	8%	79%
MAC-to-PHY	MAC-PHY messaging; UE context management	Interrupt-driven	1×TTI	0%	72%
PHY BBU pool (4x)	Wireless signal processing	Busy-polling	3×TTI	20%	28%
PHY timer	Symbol timing; callbacks	Busy-polling	Symbol ($\frac{TTI}{14}$)	72%	56%
PHY SHM rings	Network/accelerator I/O	Busy-polling	1×TTI	N/A (§5.3)	N/A (§5.3)

Table 5: Threads with realtime priority running at the DU: low CPU frequency vs RENC.

CPU cores freq	Uncore freq	CPU power
3400 MHz	2400 MHz	117 watts
3400 MHz	800 MHz	80 watts
1000 MHz	2400 MHz	105 watts
1000 MHz	800 MHz	67 watts

Table 6: Effect of CPU core (12x) and uncore frequency scaling

can adopt RENC by gradually reducing uncore frequency while monitoring low-load slack.

9.5 Overhead of RENC

Userspace agent. We measure the CPU cost of RENC's userspace agent as 20% of a CPU core, when running the CPU at low frequency. RENC does not require a dedicated CPU core nor does it busy-poll. It runs on shared cores used for other tasks such as the DU's management/logging threads.

Kernel eBPF. We measure the execution time of RENC's eBPF program as 70 ns per invocation at 1 GHz, using Linux's BPF stats functionality via the `bpftool` command. This is tiny when compared to the total cost of context-switching between threads, which is around 3 µs on our system even when running at 3.4 GHz. The eBPF program is called twice per context switch, costing <4.6% of a context-switch.

10 Related work

Section 2.2 covered energy saving techniques in traditional base stations. We present related work in other areas below.

Approaches that target vRANs. CRT [55] statically precomputes CPU core frequencies for different MAC modulation and coding schemes (MCS); this is done in an adhoc way, e.g., by checking if errors appear in code execution. Compared to CRT, RENC adds (1) principled deadline slack measurement which is done at runtime, (2) coupling of MAC scheduling and CPU frequency changes for safety, (3) handling of CPU spikes from control messages, and (4) evaluation in a real testbed. Ayala-Romero et al. [19] develop Bayesian algorithms to jointly optimize vRAN power consumption and performance. However, the system does not account for realtime deadlines. While these systems target open-source RAN implementations (OpenAirInterface and srsRAN), RENC tackles the additional challenges that arise in closed-source, binary-only setups, as well as in the comparatively higher complexity of commercial vRAN sofware.

Multi-server approaches. China Mobile has published details of a large RAN energy efficiency effort in China, which uses machine learning to pick cells to power-off [49]. A large body of theoretical work studies the bin-packing of RUs to the fewest DU servers, aiming to save energy during low load periods [61]. Cellular operators also optimize energy consumption at longer time horizons to match the diurnal traffic patterns [49], e.g., MIMO-level sleep, and cell-level sleep [22, 24]. RENC's single-server and sub-second optimization is complementary to these approaches.

Single-server approaches. Recent techniques for sharing the DU's realtime CPU cores with non-realtime workloads may be adapted for DU energy saving: Concordia [35]'s prediction of PHY task execution time can be used to place PHY cores in a deep C6 sleep state to save energy, although it will need to overcome C6's high residency time (Table 1, Appendix A). Nuberu [38] re-designs the PHY layer to better co-exist with interference from other workloads. If a DU implements a Nuberu-like technique, it can simplify RENC design by allowing rare deadline violations.

11 Conclusion

RENC takes the first step towards building a complete energy saving system for servers running commercial vRAN software. The difficulty lies in the closely-guarded nature of the software code that makes it a black box for operators, coupled with sub-ms real-time deadlines that must be met for functional correctness, as well as a highly bursty CPU load. We solve this challenge by carefully establishing intervals of low load that are safe for energy saving, and using new systems techniques such as our eBPF program to measure the available slack for CPU frequency reduction. An evaluation in a state-of-the-art vRAN testbed shows promising results, with up to 45% reduction in CPU energy compared to default settings.

A Appendix

Applicability of C6 sleep state to vDUs

C6 may work for some DU threads with multi-TTI deadlines. In our DU, these are only FlexRAN's PHY signal processing threads (which run on four of 11 cores in our DU, Section 8), whose deadline is three TTIs. Although FlexRAN provides a partial implementation for C6 [43], it is incomplete at the time of writing; FlexRAN's public C6 description suggests that it puts threads to sleep one TTI at a time with `usleep(500)` [43], which will not save energy on our CPUs, and may have been designed for older processors e.g., Cascade Lake, with 276 µs [47] C6 residency time.

On our Ice Lake CPUs, we found that a CPU core goes into C6 when an application requests a sleep of at least 650 µs (e.g., via `usleep(650)`). The call to `usleep(650)` returns after up to 820 µs, due to the 170 µs wakeup latency of the C6 state.

References

[1] 5G FAPI: PHY API specification. https://www.smallcellforum.org/reports/5g-fapi-phy-api-specification.

[2] FlexRAN 5G New Radio Reference Solution L1-L2 API Specification. https://docs.o-ran-sc.org/projects/o-ran-sc-o-du-phy/en/latest/fapi_5g_tm_overview.html#reference-documents.

[3] Intel CPU frequency scaling drivers. https://wiki.archlinux.org/title/CPU_frequency_scaling.

[4] O-RAN Alliance: Operator Defined Open and Intelligent Radio Access Networks. https://www.o-ran.org/.

[5] O-RAN Software Community: Wireless Service Library. https://docs.o-ran-sc.org/projects/o-ran-sc-o-du-phy/en/latest/wls-lib.html.

[6] O-RAN Use Cases Detailed Specification 11.0. https://www.o-ran.org/specifications.

[7] Rakuten Symphony Symware Phase Two Begins with Plans to Commercially Deploy 30,000 Units in Japan. https://symphony.rakuten.com/newsroom/rakuten-symphony-symware-phase-two-begins.

[8] Speedtest by ookla - the global broadband speed test. https://www.speedtest.net/. Accessed: yyyy-mm-dd.

[9] SRS: Software Radio Systems. https://www.srs.io/.

[10] v4l2-loopback device. https://github.com/umlaeute/v4l2loopback.

[11] 5G Network Resource Model (NRM) (3GPP TS 28.541 version 16.6.0 Release 16). https://www.etsi.org/deliver/etsi_ts/128500_128599/128541/16.06.00_60/ts_128541v160600p.pdf, November 2020.

[12] F1 Application Protocol (F1AP) (3GPP TS 38.473 version 15.8.0 Release 15). https://www.etsi.org/deliver/etsi_ts/138400_138499/138473/15.08.00_60/ts_138473v150800p.pdf, January 2020.

[13] The case for integrated high-performance RAN processing. *Ericsson Blog*, 2020.

[14] The Growth of Web Page Size. https://www.keycdn.com/support/the-growth-of-web-page-size, Nov 2022.

[15] OpenAirInterface. https://gitlab.eurecom.fr/oai/openairinterface5g, March 2023.

[16] Radisys 5G NR Software Suite. https://www.radisys.com/connect/connectran/5g, n.d.

[17] Kazi Main Uddin Ahmed, Math HJ Bollen, and Manuel Alvarez. A review of data centers energy consumption and reliability modeling. *IEEE Access*, 9:152536–152563, 2021.

[18] ORAN Alliance. E2 service model (e2sm). *O-RAN Fronthaul Working Group, O-RAN.WG3.E2SM-R003-v03.00*, 2023.

[19] Jose A Ayala-Romero, Andres Garcia-Saavedra, Xavier Costa-Perez, and George Iosifidis. Orchestrating energy-efficient vRANs: Bayesian learning and experimental results. *IEEE Transactions on Mobile Computing*, 2021.

[20] Jose A. Ayala-Romero, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Albert Banchs, and Juan J. Alcaraz. VrAIn: A Deep Learning Approach Tailoring Computing and Radio Resources in Virtualized RANs. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(1):1–34, 2016.

[22] Thomas Berglund and Helen Huynh. Energy Efficiency of Radio Units and its Impact on RAN Energy Consumption. 2017. Masters Thesis in Electrical Engineering, Ericsson AB, Lund University, Faculty of Engineering.

[23] Olof Bjering and Lakshmi Prasad. An energy efficient radio base station. 2018. Faculty of Engineering, Lund University.

[24] Blomgren, Anton and Ornstein Mecklenburg, Kasper. Energy Optimization of Radio NGR Micro G1, 2016. Student Paper.

[25] Nishant Budhdev, Mun Choon Chan, and Tulika Mitra. PR3: Power Efficient and Low Latency Baseband Processing for LTE Femtocells. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 2357–2365, 2018.

[26] Hsiang-Yun Cheng, Jia Zhan, Jishen Zhao, Yuan Xie, Jack Sampson, and Mary Jane Irwin. Core vs. uncore: The heart of darkness. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[27] Private communication with a major vRAN vendor, 2024.

[28] Intel Corporation. Enhanced Power Management for Low-Latency Workloads. https://networkbuilders.intel.com/docs/networkbuilders/power-management-enhanced-power-management-for-low-latency-workloads-technology-guide-1617438252.pdf, 2023.

[29] Bjorn Debaillie, Claude Desset, and Filip Louagie. A Flexible and Future-Proof Power Model for Cellular Base Stations. In *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pages 1–7, 2015.

[30] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. *Agora: Real-Time Massive MIMO Baseband Processing in Software*, page 232–244. Association for Computing Machinery, New York, NY, USA, 2020.

[31] NTT DOCOMO. Green Mobile Network: Energy Saving Efforts by SK Telecom and NTT DOCOMO. https://www.docomo.ne.jp/english/binary/pdf/corporate/technology/rd/docomo6g/GreenMobileNetworksWhitePaper_22February2023.pdf, Feb 2023.

[32] eBPF.io. eBPF. https://ebpf.io/, March 2023.

[33] CapGemini Engineering. CapGemini 5G gNodeB. https://capgemini-engineering.com/nl/en/services/next-core/wireless-frameworks/, March 2023.

[34] Robert Falkenberg and Christian Wietfeld. FALCON: An accurate real-time monitor for client-based mobile network data analytics. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2019.

[35] Xenofon Foukas and Bozidar Radunovic. Concordia: teaching the 5G vRAN to share compute. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 580–596. ACM, 2021.

[36] Xenofon Foukas, Bozidar Radunovic, Matthew Balkwill, and Zhihua Lai. Taking 5G RAN Analytics and Control to a New Level. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, page to appear. Association for Computing Machinery, 2023.

[37] Fujitsu. Fujitsu launches sustainable 5G vRAN to deliver potential reductions in CO2 emissions of over 50 percent. https://www.fujitsu.com/global/about/resources/news/press-releases/2022/0224-01.html, 2022.

[38] Gines Garcia-Aviles, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Pablo Serrano, and Albert Banchs. Nuberu: Reliable RAN virtualization in shared platforms. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 749–761, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Srihari Das Sunkada Gopinath, Sandeep Burugupally, and Ajeet Singh Nathawat. An adaptive power management method for radio access network data plane systems. In *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1129–1134. IEEE, 2022.

[40] GSMA Intelligence. Going Green: Benchmarking the Energy Efficiency of Mobile, 2021.

[41] Sofonias Hailu, Mikko Saily, and Olav Tirkkonen. RRC state handling for 5G. *IEEE Communications Magazine*, 57(1):106–113, 2018.

[42] Red Hat. Red Hat and Arm Collaborate to Deliver More Energy-Efficient 5G and vRAN Solutions. https://www.redhat.com/en/blog/red-hat-and-arm-collaborate-deliver-more-energy-efficient-5g-and-vran-solutions, 2023.

[43] Intel. Intel Network Builder Insights Series: The Full vRAN Experience. https://networkbuilders.intel.com/university/webcasts/the-full-vran-experience, Sep 14, 2022 2022. Slides: https://www.brighttalk.com/resource/core/408572/the-full-vran-experience---deck_873458.pdf.

[44] Intel. Dynamic Frequency Scaling for Mixed Criticality Real-Time Scenarios on 11th Generation Intel® Core™ Processors. https://www.intel.com/content/www/us/en/content-details/723446/dynamic-frequency-scaling-for-mixed-criticality-real-time-scenarios-on-11th-generation-intel-core-processors.html, 2023.

[45] Intel. FlexRAN README. https://github.com/intel/FlexRAN/blob/08ddafedce6a1b5690221f881738bddaa7093588/README.md, Aug 2023.

[46] Intel. FlexRAN Reference Architecture for Wireless Access. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html, March 2023.

[47] Intel. Native hardware idle loop for modern Intel processors. https://github.com/torvalds/linux/blob/master/drivers/idle/intel_idle.c, n.d.

[48] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[49] Tong Li, Li Yu, Yibo Ma, Tong Duan, Wenzhen Huang, Yan Zhou, Depeng Jin, Yong Li, and Tao Jiang. Carbon emissions and sustainability of launching 5g mobile networks in china, 2023.

[50] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V. Vasilakos. Cloud Computing: Survey on Energy Efficiency. 2014.

[51] mpv.io. mpv: a free, open source, and cross-platform media player. https://mpv.io/, May 2024.

[52] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. A variegated look at 5G in the wild: performance, power, and QoE implications. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 610–625, 2021.

[53] Ashkan Nikravesh and Bimmy Pujari. spinics.net patch: bpf: Add realtime clock to BPF. https://www.spinics.net/lists/bpf/msg23699.html, July 2020.

[54] NTT. NTT Succeeded in Low Power Consumption of 5G Virtualized Radio Base Station (vRAN). https://group.ntt/en/newsrelease/2023/05/24/230524b.html, 2023.

[55] Ujjwal Pawar, Bheemarjuna Reddy Tamma, and Franklin A Antony. Traffic-Aware Compute Resource Tuning for Energy Efficient Cloud RANs. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 01–06, 2021.

[56] Pablo Fernández Pérez, Claudio Fiandrino, and Joerg Widmer. Characterizing and Modeling Mobile Networks User Traffic at Millisecond Level. In *Proceedings of the 17th ACM Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization*, WiNTECH '23, page 64–71, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Ericsson Technology Review. Radio network energy performance: shifting focus from power to precision. https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/radio-network-energy-performance-shifting-focus-from-power-to-precision, 2014.

[58] Sonal Saha and Binoy Ravindran. An experimental evaluation of real-time DVFS scheduling algorithms. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[59] Samsung. Samsung Announces the Next Phase of Its 5G vRAN. https://news.samsung.com/global/samsung-announces-the-next-phase-of-its-5g-vran, 2023.

[60] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. In *2019 International Conference on High Performance Computing and Simulation (HPCS)*, pages 399–406, 2019.

[61] Tshiamo Sigwele, Atm S Alam, Prashant Pillai, and Yim F Hu. Energy-efficient cloud radio access networks by cloud based workload consolidation for 5G. *Journal of Network and Computer Applications*, 78:1–8, 2017.

[62] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[63] Vodafone. Vodafone, Wind River, Intel, Keysight Technologies and Radisys test green Open RAN. https://www.vodafone.com/news/technology/vodafone-wind-river-intel-keysight-technologies-radisys-test-green-open-ran, 2022.

[64] Jingjin Wu, Yujing Zhang, Moshe Zukerman, and Edward Kai-Ning Yung. Energy-efficient base-stations sleep-mode techniques in green cellular networks: A survey. *IEEE communications surveys & tutorials*, 17(2):803–826, 2015.

[65] Yi Zhang, Łukasz Budzisz, Michela Meo, Alberto Conte, Ivaylo Haratcherev, George Koutitas, Leandros Tassiulas, Marco Ajmone Marsan, and Sofie Lambert. An overview of energy-efficient base station management techniques. In *2013 24th Tyrrhenian International Workshop on Digital Communications-Green ICT (TIWDC)*, pages 1–6. IEEE, 2013.

Building Massive MIMO Baseband Processing on a Single-Node Supercomputer

Xincheng Xie, Wentao Hou, Zerui Guo, Ming Liu

University of Wisconsin-Madison

Abstract

The rising deployment of massive MIMO coupled with the wide adoption of virtualized radio access networks (vRAN) poses an unprecedented computational demand on the baseband processing, hardly met by existing vRAN hardware substrates. The single-node supercomputer, an emerging computing platform, offers scalable computation and communication capabilities, making it a promising target to hold and run the baseband pipeline. However, realizing this is non-trivial due to the mismatch between (a) the diverse execution granularities and incongruent parallel degrees of different stages along the software processing pipeline and (b) the underlying evolving irregular hardware parallelism at runtime.

This paper closes the gap by designing and implementing **MegaStation**[1]–an application-platform co-designed system that effectively harnesses the computing power of a single-node supercomputer for processing massive MIMO baseband. Our key insight is that one can adjust the execution granularity and reconstruct the baseband processing pipeline on the fly based on the monitored hardware parallelism status. Inspired by dynamic instruction scheduling, MegaStation models the single-node supercomputer as a tightly coupled microprocessor and employs a scoreboarding-like algorithm to orchestrate "baseband processing" instructions over GPU-instantiated executors. Our evaluations using the GigaIO FabreX demonstrate that MegaStation achieves up to 66.2% lower tail frame processing latency and 4× higher throughput than state-of-the-art solutions. MegaStation is a scalable and adaptive solution that can meet today's vRAN requirements.

1 Introduction

Massive MIMO (multiple-in multiple-out)–equipping a base station with an array of radio antennas to serve many users simultaneously–has become the essential wireless technology in modern cellular networks such as 5G. It boosts spectral efficiency via multi-antenna techniques and increases network coverage, capacity, and throughput, which many 5G mid-band deployments have adopted [3–5, 12, 14, 38].

RAN virtualization has gained significant traction. It has been employed by many mobile network operators [11, 15, 35, 43, 50–53] and strongly advocated by industry players [36, 118]. It replaces specialized RAN hardware with general-purpose commodity servers or programmable accelerators for baseband signal processing, entailing a slew of benefits, such as mitigating vendor lock-in, offering better inter-operability and operational visibility, reducing the total cost of ownership, and improving network management agility.

However, realizing massive MIMO under vRAN is challenging due to the high computational demands and single-digit millisecond processing deadline of its baseband processing. It consists of a pipeline of non-trivial matrix manipulation tasks–like FFT/IFFT, equalization, and en(de)coding–and converts between radio bits and data packets for each antenna-user pair at every subcarrier frequency. The compounding effect of continuously massive MIMO dimensional scaling [63, 64, 117], the rising number of subcarrier frequencies, increasing radio units being deployed at a cell site, and stringent performance requirements [18] necessitate enormous execution parallelism and computing power, which are hardly met by today's vRAN computing substrates.

People have tackled this problem by either building rack-scale distributed baseband processing systems [71, 121] or developing specialized accelerator-based platforms [28, 58, 79, 100]. For example, Hydra [71] splits the frame processing pipelines [65] over multiple X86 servers and designs efficient synchronization to minimize the data shuffling overheads. When scaling, one has to use multiple servers, adding capital/operational costs and taking more physical space, which is a key concern for future Telco site infrastructures [6, 39]. LuMaMi [100] develops a bespoke testbed using 50 Xilinx Kintex-7 FPGAs and supports one modest-sized MIMO (100×10) configuration. The system is not flexible and requires relaying out the software-defined radio array and partitioning symbol processing paths when updating the MIMO settings or accommodating user traffic vagaries. Therefore,

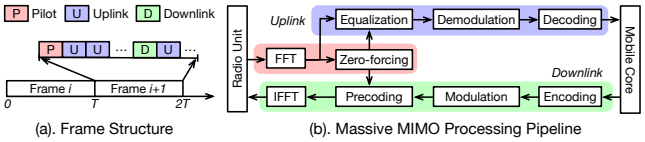
[1] MegaStation is available at https://github.com/netlab-wisconsin/MegaStation.

prior solutions struggle to provide adequate computing parallelism with flexible programmability at an acceptable cost.

Empowered by recent cluster interconnects (like Routable PCIe [2] and UALink [54]), infrastructure composability becomes possible, and an emerging computing platform–called **single-node supercomputer (SNC)** [44]–receives great attention and interest. It is a physically compacted testbed, comprising one server host and a composable GPU pool (enclosing tens of GPUs in standalone chassis), connected through backplane or external fabric switches, such as the GigaIO SuperNODE [44]. An SNC (a) holds substantial execution parallelism where remote GPUs behave as local ones and run host-native system stacks, (b) provisions adequate and consistent bandwidth for host-GPU and GPU-GPU communication, and (c) allows incrementally on-demand GPU scaling, making it a promising target for running highly parallel and compute-intensive applications, like AI/HPC.

In this paper, we design and implement **MegaStation** that harnesses the computing power of an SNC to process the baseband for vRAN-based massive MIMO. This is challenging because of the mismatch between the application software parallelism and the underlying hardware computing parallelism. The MIMO baseband processing pipeline comprises a sequence of stages that run at different execution granularities (such as antenna, user, and subcarrier) with incongruent parallel degrees. State-of-the-art solutions [65, 71, 100, 121] parallelize the pipeline at the frame, symbol, or task level. They are all suboptimal and would gradually make the underlying hardware parallelism irregular, adversely affecting the baseband processing performance (§3). Thus, without careful coordination and scheduling, one would experience radio bandwidth drops, frame processing deadline violations, and computing resource waste.

MegaStation addresses the challenge by revamping the baseband processing pipeline and applying techniques from the computer architecture field. *Our key idea is to model the single-node supercomputer as a tightly coupled microprocessor and employ a scoreboarding-like scheme [115] to schedule "baseband processing" instructions based on GPU hardware status.* MegaStation comprises four software modules: (a) *an instruction unit* that translates frame sequences to task instructions and analyzes their data dependency and structural hazards; (b) *processing function units*, which holds baseband processing instructions and runs over GPUs; (c) *the scoreboarding*, a centralized bookkeeper that continuously tracks the execution status of inflight instructions and performs resource accounting of the GPU pool, serving as the basis for other components; (d) *a pipeline scheduler*, orchestrating instruction execution via a new algorithm (called LROC) that sparingly synthesizes the least slack time scheduling with instruction reordering, over-commitment, and coalescing techniques. Therefore, MegaStation adjusts the execution granularity and reconstructs the baseband processing pipeline on the fly based on the dissected hardware parallelism status.



(a). Frame Structure (b). Massive MIMO Processing Pipeline

Figure 1: Frame structure and baseband processing pipeline.

We built MegaStation atop Agora [65] and Nvidia Aerial SDK [30] and evaluated it over the GigaIO FabreX platform. Compared with three state-of-the-art solutions [65, 71, 100, 121], MegaStation achieves up to 66.2% lower tail frame processing latency for the 256×128 MIMO setting. MegaStation effectively harnesses any available computing power of the GPU pool and schedules suitable baseband processing jobs without overloading the GPU. Further, we empirically demonstrate the scalability and adaptiveness of MegaStation under different vRAN settings. For example, our system can support 8 RUs under the 128×64 MIMO setting on 6 GPUs with up to $4\times$ higher throughput than other approaches.

2 Background and Motivation

We provide the necessary background about massive MIMO baseband processing, introduce the single-node supercomputer, and show its potential for vRAN acceleration.

2.1 Baseband Processing of Massive MIMO

In massive MIMO, the multi-antenna radio unit (RU) receives wireless signals combining multiple users' transmissions, where each antenna digitizes these signals into per-subcarrier (i.e., frequency) in-band and quadrature (IQ) samples and encapsulates these IQ samples as packets. The baseband processing unit (or BBU)–exchanging IQ sampled packets with an RU via a fronthaul link–performs a series of pipelined tasks to recover the bits from radios and then delivers them to the mobile core, and vice versa.

Take a typical $M \times K$ MIMO configuration as an example that serves K users concurrently via M antennas. Massive MIMO uses frequency canceling to support different users on the same frequency. We represent the signal streams transmitted by K user antennas for a given subcarrier as a $X_{K\times 1}$ vector. An RU then receives $Y_{M\times 1}$ signal, modeled as $Y_{M\times 1} = W_{M\times K} \times X_{K\times 1}$, where W is the computed precoder, i.e., $W_{i,j}$ capturing the wireless channel status between an antenna i and a user j. The baseband processing is responsible for converting between the signals Y from all RU antennas and the users' signals X. The processing granularity is a time frame (Figure 1-a), whose length depends on user and cell configuration, lasting up to a few milliseconds. Each frame is further divided into a few to tens of symbol durations (14 in a typical case). The beginning of a frame is a *Pilot* symbol that encodes the channel state information (CSI) and allows the BBU to estimate the channel matrix $W_{M\times K}$. The rest are *non-Pilot* data symbols, carrying one block of transmitted or received modulated data bits for all users on each subcarrier.

Processing Pipeline. Massive MIMO generally comprises

an uplink and a downlink processing pipeline inside its BBU (Figure 1-b), each consisting of multiple stages. Take the uplink one as an example. First, it transforms the time-domain IQ samples into frequency-domain samples for each antenna through FFT. Second, the pipeline obtains the channel matrix (W) from the pilot symbol, and then uses the zero-forcing (ZF) technique [56, 123] to compute the precoder via the pseudo-inverse of the matrix W, i.e., $H = (W^*W)^{-1}W^*$. Third, for non-pilot symbols, the BBU performs equalization to recover user signals (X) by computing $H \times Y$. Fourth, it applies demodulation to extract information bits from complex numbers. Finally, the BBU performs decoding using a forward error correction (FEC) scheme to generate and verify the user bits. Akin to prior studies [65, 71], we use the low-density parity-check (LDPC) algorithm [70]. The downlink pipeline works in the opposite way, where each task performs the reverse functionality as the one in the uplink.

2.2 High Computing Demand under vRAN

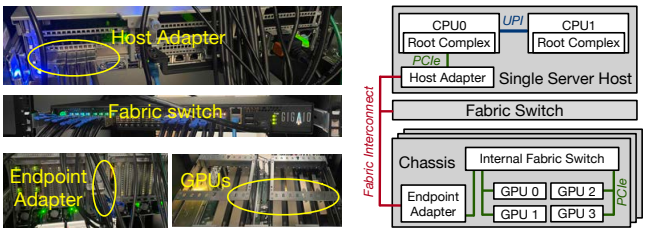
RAN virtualization realizes the BBU's functionalities using commodity servers or programmable engines instead of proprietary and specialized boxes [42, 46, 47, 49]. Even though this entails many benefits (e.g., cost efficiency, management agility, and operational visibility), vRAN drastically increases the computing demand, especially under massive MIMO.

There are three reasons. First, the baseband processing includes a pipeline of computing-intensive and highly parallel tasks (§2.1). For example, FFT and IFFT are parallelized at the antenna level, whose computing complexities depend on the number of subcarriers. Equalization, precoding, demodulation, and modulation tasks are subcarrier-parallel. Each requires non-trivial matrix calculations, and their dimensions depend on the number of subcarriers and users. The parallelism of en(de)coding hinges on the number of users. Such non-consistent parallel degrees indicate that data shuffling is required when crossing different execution stages [71].

Second, the MIMO is scaling up with an increasing number of antennas and users to satisfy future communication demands, such as XL-MIMO [63, 64, 117]. Third, cellular networks impose stringent requirements from data rate and latency perspectives. For example, only considering a bandwidth of 100MHz [18], the peak downlink/uplink spectral efficiency is 30/15 bps/Hz, translating to a maximum data rate of 3/1.5 Gbps. Further, 5G requires single-digit millisecond transmission latency between a user and a base station [17, 18] for two types of communication: ultra-reliable and low-latency communications (URLLC) and enhanced mobile broadband (eMBB). In the worst case (if an irrecoverable bit error happens), the BBU should send a downlink NACK to the user within four time slots (a few to 10s of milliseconds).

2.3 Prior Solutions

The de facto hardware substrate for deploying vRAN is commodity servers, starting from a pioneering effort–Sora [114].



(a). Hardward Testbed (b). System Architecture

Figure 2: The hardware testbed and system architecture of a single-node supercomputer based on the GigaIO's platform.

It processes the 802.11 a/b/g protocol using an Intel Core 2 processor (with SIMD extensions). BigStation [121], built atop Sora, realizes a distributed processing pipeline for multi-user MIMO (MU-MIMO) by parallelizing frames over multiple PCs. Regarding massive MIMO, Agora [65] is the first system that implements the baseband processing pipeline over an Intel Skylake processor by exploring data parallelism at the task level, supporting small-scaled massive MIMO (64×16). Hydra [71] scales up the massive MIMO capacity by distributing uplink/downlink pipelines to multiple X86 servers. It follows the BigStation architecture but designs efficient synchronization to minimize inter-stage data shuffling. Server-based solutions offer general-purpose computing and simplify software updates, but suffer from high TCO (total cost of ownership) and physical space constraints, especially when scaling the massive MIMO because more servers are required.

People also develop hardware-accelerated solutions to satisfy computing demands. Industry vendors build specialized BBUs (baseband processing units)–such as Nokia AirScale Baseband Module [47], Huawei BBU5900 [46], H3C BBU5200 [45]–and use ASIC DSP blocks to process the uplink and downlink pipelines. Atomix [58] takes programmable DSPs and builds a modular development framework for implementing wireless protocols and applications. FlexCore [79] parallelizes the detection of large numbers of mutually interfering information streams over GPUs. Intel's FlexRAN [28] provides a development SDK to build FPGA-accelerated vRAN stacks. LuMaMi [100] develops a specialized distributed FPGA testbed (connected via PCIe switches) and can handle 20 MHz bandwidth under a 100×10 sized MIMO. Albeit providing better energy/cost efficiency, such systems generally fall short in programmability, operability, and maintenance. For example, LuMaMi necessitates reconfiguring the FPGA array and partitioning the symbol handing path to accommodate MIMO and traffic changes.

2.4 Single-Node Supercomputer

Empowering by emerging load/store cluster interconnects [2, 40], infrastructure composability has become a new hardware architecture property [7–10, 13]. It transparently expands the server's computing capacity and improves the overall system cost efficiency. This yields a new and powerful computing platform called **Single-Node Supercomputer (SNC)**.

Take the GigaIO's FabreX and SuperNODE testbed [8] as

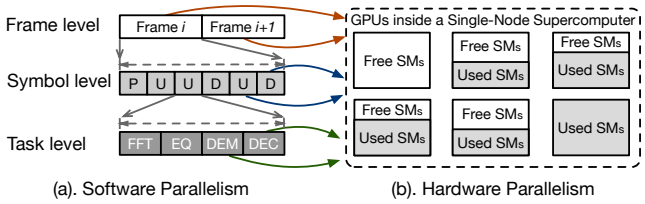


Figure 3: Three levels of parallel processing across the pipeline.

an example, which is our prototype and evaluation target. As shown in Figure 2-b, the platform consists of (1) host adapters (i.e., FA400X [23, 24]) that extend the host server's PCIe subtree; (2) standalone chassis boxes (i.e., RB4082 [33]), where each encloses endpoint adapters with up to 8 GPUs per appliance, connected through a PCIe backplane; (3) an external fabric switch (i.e., RS4024 [25, 34]), which connects host servers and chassis via SFF-8644 [37] copper/optical cables. In the current generation, the SuperNODE can attach up to 32 GPUs to a single server host. FabreX and SuperNODE are built atop routable PCIe 3.0/4.0 [2], and are under upgrading to other interconnects [40, 54]. Thus, a single-node supercomputer is one commodity server equipped with tens of GPUs, all of which stay under the server's PCIe root complexes.

The platform has several features. First, each remote GPU appears as a local PCIe device, which is directly accessed from host applications through existing driver and system stacks. This differs from today's distributed GPU clusters in the HPC and data centers, where a GPU access should traverse the message passing layer and networking stack [48, 68, 72, 76, 81, 99, 119], entailing communication overheads. Second, all GPUs are located under a multi-rooted tree topology inherited from PCIe, where host-GPU and GPU-GPU communications are provisioned with adequate bandwidth (e.g., $\times$16 lanes). Third, the platform can be scaled incrementally, where one can add/remove GPUs on demand without considering the available PCIe slots and power of the host server, simplifying infrastructure planning and management. Besides, the system is physically compacted (4U in our lab setup) with air cooling in the remote chassis. A single-node supercomputer resembles an NVIDIA DGX platform [31, 32]. However, it is loosely coupled physically with less GPU-GPU bandwidth (compared with NVLink 3.0/4.0) but has a higher GPU consolidation ratio and better infrastructure flexibility.

We believe that SNC is a promising vRAN substrate, especially when handling the dynamic high computing demand of massive MIMO. This is because (a) it encloses high computing density (GPU #) without bandwidth oversubscription in a single server, alleviating the need to handle the cross-node communication complexity in distributed systems; (b) it is a physically compact setup and has better space efficiency when scaling than today's rack-scale vRAN infrastructures; (c) GPUs can be added/removed on-demand flexibly via the PCIe hotplug without changing the underlying computing setting, whose management cost is less than dealing with the addition and removal of standalone GPU servers.

3 Understanding the Execution Parallelism

To effectively use SNC for baseband processing, this section characterizes the impact of different execution parallelism.

3.1 Types of Software and Hardware Parallelism

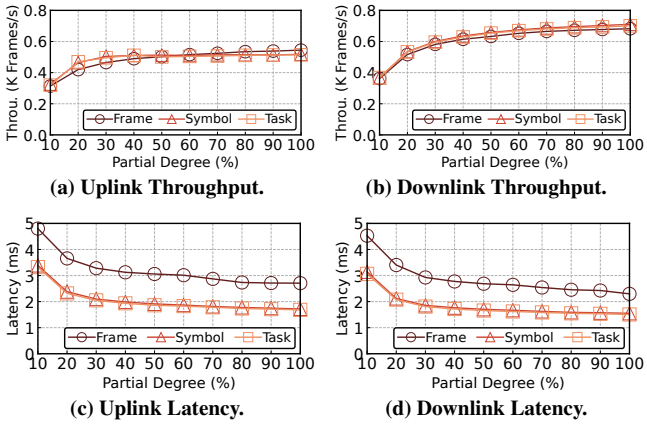
SW Parallelism. As described in §2.1, the baseband processing pipeline encloses a sequence of stages that exhibit different execution granularities and incongruent parallel degrees. There are three ways to parallelize it (Figure 3).

- *Frame Level.* The first is a coarse-grained approach that concurrently processes a window of consecutive frames, which has been explored in LuMaMi [100]. Since the computing intensity is inconsistent across different stages, it is challenging to do resource provisioning;
- *Symbol Level.* The second one is to parallelize symbol processing within a frame, such as Hydra [71]. Except for the pilot symbol (§2.1), other symbol handling can happen concomitantly. It is finer-grained than the first case but requires tracking execution dependency at runtime;
- *Task Level.* The third strategy is to schedule computations at the finest granularity (like BigStation [121]) that harnesses the available parallelism at best. We define a task as a MIMO processing stage as in Figure 1. However, this approach complicates the pipeline design because one should not only track dependent tasks, but also introduce the synchronization barrier (through shuffle) when the parallel degree across different phases changes, e.g., FFT (antenna#) $\rightarrow$ EQ/DEM (subcarrier#) $\rightarrow$ DEC (user#).

HW Parallelism. We quantify the available parallelism as the number of GPU's streaming processors (SM), denoted as P. These SMs could come from one or several GPUs. We then represent the adequate parallelism (SM #) to run a job in our baseband processing as Q. Thus, we categorize the hardware parallelism scenarios as follows:

- *Full Parallelism*, where $P \geq Q$ and $P\ SMs \in 1\ GPU$. The job can be placed in one GPU with no performance loss;
- *Fragmented Parallelism*, where $P \geq Q$ and $P\ SMs \in x\ GPUs$ and $x > 1$. Even though there are enough parallel engines, the job needs to be partitioned across several GPUs, and data shuffle would be required for synchronization when crossing different stages, hurting performance;
- *Partial Parallelism*, where $0 < P < Q$ and $P\ SMs \in x\ GPUs$ and $x \geq 1$. There is inadequate parallelism to run the job and one would see performance interference due to GPU sharing. The problem becomes even worse when the available SMs are from different GPUs;
- *Delayed Parallelism*, where $P = 0$. This indicates all SMs are busy. A job has to be queued first and be awakened when one of the above three types of parallelism emerges.

The Dilemma. Ideally, we should devise an execution strategy that always makes the baseband processing job run in full parallelism mode. To satisfy the stringent performance



(a) Uplink Throughput. (b) Downlink Throughput.

(c) Uplink Latency. (d) Downlink Latency.

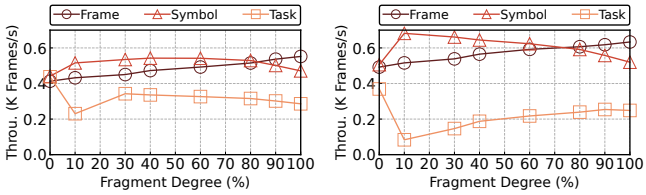
Figure 4: Baseband processing performance at three execution granularities varying the partial degree. We report the frame processing latency and throughput in all three cases.

requirements and high computing demand of massive MIMO, one should fully exploit the parallelism across three levels, i.e., parallelizing frame, symbol, and task processing as much as possible. As a result, the irregular software parallelism will gradually make fragmented, partial, and delayed hardware parallelism appear frequently, adversely impacting pipeline performance. We aim to characterize this dilemma and understand how different parallel executions perform under the above hardware parallel scenarios. Our experimental setup is described in §5.1 and the MIMO configuration is 128×32.

3.2 Partial Parallelism

Setup. We configure the partial parallelism by co-locating the target baseband processing task with a persistently running background job (which spawns a number of busy polling threads to consume the SM resources). We vary the number of occupied SMs and explore how it impacts three execution granularities. We define the partial degree as $\frac{P}{Q}$.

Results. The three software parallelisms can tolerate a modest to high partial degree. As shown in Figure 4, when half of the SMs are occupied, we observe up to 7.3% performance drop in all three cases. This is because the parallelism of the baseband processing pipeline is determined by its most computing-intensive phase. Other tasks requiring fewer SMs can still run in full parallelism when the partial degree is high. Thus, the performance-affected parts due to inadequate SMs would only contribute to a small portion of the total execution according to Amdahl's Law. Symbol/Task-level parallelism outperforms the frame-level one by 56.9%/60.4% in terms of uplink/downlink frame processing latency on average across all cases. This is because finer-grained execution parallelizes different symbols and tasks within a frame over multiple SMs. In contrast, frame-level execution cannot fully utilize the GPU for a single frame, resulting in a higher per-frame latency but allowing more frames to be scheduled concomitantly. As a result, all three software parallelism methods achieve similar throughput. We further explore an extreme scenario that only configures 1 SM (1.0% partial degree). Regardless of running



(a) Uplink. (b) Downlink.

Figure 5: Baseband processing perf. at three granularities varying the fragment degree. We measure the per-(frame | symbol | task) throughput (K frames per second).

at which granularity, the performance is reduced by more than $10\times$, indicating that a heavily-loaded GPU can easily violate the performance requirements of baseband processing.

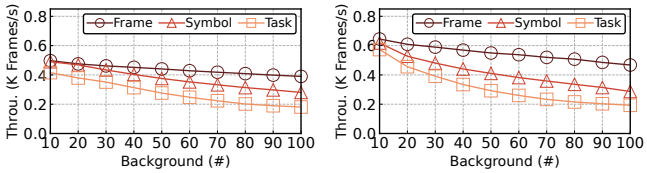
Takeaways. Partial parallelism with modest/high partial degree exhibits marginal performance degradation. However, as shown in Figure 4, the baseband processing performance drops significantly if a GPU has less than 20.0% free SMs. Therefore, one should carefully estimate the partial degree before allocating tasks to GPUs. Even though all three software parallelism options deliver similar throughput, frame-level parallelism presents the worst processing latency, making it ill-suited for an execution scenario with stringent deadlines.

3.3 Fragmented Parallelism

Setup. We provision required parallelism (Q SMs) from two composable GPUs and define the ratio of two GPUs' SMs as the *fragment degree*, i.e., $\frac{GPU1\ SM\#}{GPU2\ SM\#}$. The number of SMs is controlled via the same synthetic background job used in the partial parallelism section. We then examine how different execution granularities of the baseband processing behave when gradually varying the fragment degree.

Results. At the frame level, the pipeline proportionally schedules frames to two GPUs based on the number of SMs in a round-robin fashion. As shown in Figure 5, frame parallelism can tolerate high fragment degree. Compared to zero fragment degree, the throughput of the uplink/downlink increases by 34.1%/30.6% when fragment degree reaches 100.0%. The reason is that a single GPU exposes the FIFO queue structure [55, 105], limiting the number of concurrent scheduled frames. With one more GPU, the number of frames that can be scheduled simultaneously increases, causing overall throughput improvement. Although resources are split between two GPUs, each individual GPU exhibits partial parallelism. As shown in the partial parallelism, a slight reduction in the number of SMs can still be accommodated. This indicates that partitioning would not lead to a significant performance drop.

The execution strategy at the symbol level is similar; however, symbols (other than pilot) cannot run before the completion of the pilot symbol in each frame. If symbols do not stay within the same GPU as the pilot, extra data shuffling is required for another GPU to synchronize results. Also, symbols show up to 25.1%/65.1% increase under low fragment degrees (from 0.0% to 44.0%/16.0% fragment degree for uplink/downlink) thanks to the increasing hardware request



(a) Uplink. **(b) Downlink.**

Figure 6: Baseband processing perf. at three granularities varying background jobs #. We measure the per-(frame | symbol | task) throughput (K frames per second).

queue. It even outperforms the frame-level parallelism (up to 17.8%/33.3% higher) due to the fine-grained execution. This provides a GPU with a wider range of options for selecting smaller jobs to run when there are limited SMs, yielding higher utilization and performance improvement. Conversely, coarse-grained large kernels offer less flexibility, resulting in an inefficient use of fragmented resources. Nevertheless, symbol parallelism underperforms the frame case when the fragment degree exceeds 70.0%. At 100.0% fragment degree, we observe a 14.5%/17.4% uplink/downlink performance drop. The reason is that extra data shuffling required for symbol-level scheduling cannot be hidden by computation under a high fragment degree (discussed in Appendix A.1).

When parallelizing at the task granularity, we also partition tasks proportionally to two GPUs. A key challenge is to minimize data shuffling overheads. We follow the Hydra [71] approach that only synchronizes when the parallelism changes. This indicates that equalization and demodulation (precoding and modulation) tasks across the uplink (downlink) pipeline run in the same GPU. Task level experiment performs worst according to Figure 5, presenting a 48.2%/52.2% uplink/downlink throughput drop. In contrast to the symbol case that only synchronizes pilot computation once per frame, tasks synchronize data between every two stages for non-pilot symbol processing, resulting in 13 times of synchronization (excluding pilot) per frame in a typical 14-symbol frame.

Takeaways. Fragmented parallelism, albeit providing enough SMs, should carefully divide jobs based on the underlying GPU's computing capability to mitigate jeopardy. Finer-grained execution at the symbol level helps utilize the available SMs in each GPU. But one would see diminishing returns when synchronization overheads dominate. There is a trade-off between execution granularity and data shuffle frequency.

3.4 Delayed Parallelism

Setup. We design the delayed parallelism experiment by interleaving the target baseband processing job with special background kernels. They run large-scale GEMM (General Matrix Multiplication) tasks, where $M \times N \times K = 1280 \times 1024 \times 512$, consuming as many available SMs as possible. A bunch of such kernels is being pushed to the GPU consistently one after another. We investigate how the GPU internal scheduler affects different-sized baseband processing jobs.

Results. We find that the frame-based execution adapts to delayed parallelism the best, followed by symbol and then task granularity. For example, when there are 100 interleaved kernels, the uplink/downlink frame pipeline observes 23.5%/30.9% degradation while uplink/downlink symbol is dropped by 47.2%/60.8%, and uplink/downlink task is 60.0%/74.7% lower. The key reason is that GPUs employ a strict FCFS scheduling policy with a limited amount of hardware execution queues [55, 105, 106], providing little fairness guarantees. When execution contention happens, coarse-grained scheduling submits a sequence of jobs in batches and thus would receive more GPU computing cycles. Besides, frequent kernel launches and completions at a finer granularity cause more GPU interruptions and jeopardize performance.

Takeaways. The impact of delayed parallelism stems from how the computing resource of a GPU is shared among competing kernels. The inherent FCFS policy and kernel-agnostic partition make inefficient multi-tenancy support on the GPU. Thus, when determining the execution granularity under GPU contention, one should collaboratively consider the kernel size distribution of co-located baseband processing jobs.

3.5 Summary

There is no *one-size-fits-all* execution granularity that allows the baseband processing pipeline to accommodate different irregular hardware parallelism. Therefore, an ideal baseband processing solution over an SNC should be execution adaptive, aware of the underlying target status at runtime, and collectively consider co-located kernels. None of the prior solutions [28, 30, 58, 65, 71, 100, 114, 121] can achieve this.

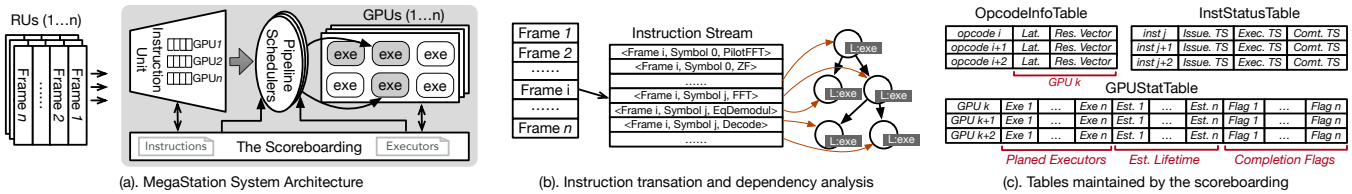
4 MegaStation: Design and Implementation

This section describes MegaStation and shows how we address the above dilemma. Our system design goals are:

- **Delay deadline.** MegaStation should satisfy the processing latency requirements of each time frame. It should ensure the average/tail delay stays within the constraint;
- **High utilization.** MegaStation should fully harness the available hardware parallelism (no matter which type) and minimize stalled or idle GPU cycles. It should strive to consolidate baseband processing tasks on just enough GPUs without over- or under-subscriptions;
- **Adaptability.** MegaStation makes no assumption about the frame structure and can handle different uplink/downlink frames. It should automatically adapt its processing capability to the target spectrum, and client rate.

4.1 Key Idea and System Overview

Key Idea. MegaStation effectively leverages the massive parallelism of a single-node supercomputer to process data frames. *Our key insight is that one can adjust the execution granularity and reconstruct the baseband processing pipeline on the fly based on the hardware parallelism status.* We are inspired by the dynamic instruction scheduling [115, 116] from computer architecture field. Dynamic instruction scheduling reorders instructions to prevent stalls by scheduling indepen-



(a). MegaStation System Architecture (b). Instruction transation and dependency analysis (c). Tables maintained by the scoreboarding

Figure 7: (a) provides an overview of MegaStation. (b)/(c) depict the details of the instruction unit and scoreboarding.

dent instructions when the execution engine is free. We view the baseband processing task on GPUs as instructions and apply dynamic scheduling. This flexible approach allows us to make real-time adjustments to the software parallelism based on the status of the GPUs. Hence, *we model the single-node supercomputer as a tightly coupled microprocessor and employ a scoreboarding-like algorithm* to schedule "baseband processing" instruction streams over the composable GPU pool. MegaStation continuously bookkeeps the occupancy of the GPU's computing resources (e.g., SMs, registers, and shared memory), and issues an instruction when it is ready. Within the frame deadline, MegaStation coalesces one or several instructions with data locality to maximize the probability of full parallelism, mitigates the fragmented/partial parallelism via instruction reordering, and tackles the delayed parallelism via managed over-commitment.

System Overview. MegaStation comprises four components (Figure 7-a): (a). *Processing Functional Units or Executors (§4.3)*, running baseband processing tasks over the GPU SMs; (b). *Instruction Unit (§4.4)*, which takes data frame sequences as inputs, translates them into instructions, and places them into an instruction queue; (c). *The Scoreboarding (§4.5)*, a centralized bookkeeper that continuously tracks the execution status of inflight instructions and performs resource accounting of the GPU pool, serving as the basis for other system components; (d). *Pipeline Scheduler (§4.6)*, which orchestrates instruction execution, determines what and how to run the next tasks based on committed status. MegaStation can support one or multiple radio units concomitantly [62, 71].

4.2 Baseband Processing Tasks as Instructions

MegaStation models a baseband processing task as an instruction, represented as $\langle$`Opcode`, `Operand1`$_{src}$, `Operand2`$_{src}$ (optional), `Operand`$_{dst}\rangle$. Our current design uses 4 bits to denote an opcode and supports 11 different opcodes (Table 1 in Appendix A.2), i.e., Data load & store, PilotFFT, UpFFT, iFFT, zero-forcing (ZF), equalization + demodulation (EqDemodul), modulation (Modul), precode, decode, and encode. Among them, equalization and precoding take three operands, the second source operand being a beamforming matrix, while the rest require two. An operand is mostly a pointer to the data input/output location. Thanks to the local view of GPUs on a single-node supercomputer, multiple GPUs share a unified memory address space when presenting a src/dst operand.

The lifecycle of an instruction comprises three basic execution stages: **ISSUE**, **EXEC**, and **COMMIT**. First, the instruction unit conducts dependency analyses of each enqueued instruction and marks it ready for issuing when data and executor are in place. Next, an instruction is scheduled and inserted into the request queue of a GPU executor. A functional unit processes requests following its prescribed order. Finally, upon completion, commit notification is broadcast to other architectural components (like the scoreboard), pushing the pipeline forward. At the beginning of each stage, We collect the timestamps TS_{issue}, TS_{exec}, TS_{commit}.

4.3 Baseband Processing Executors over GPUs

MegaStation instantiates GPU-based task executors on demand and uses them to model instruction running on GPUs. Since a GPU does not expose any internal resource usage primitives at run-time, an executor is not a physical computing entity but a logical representation that describes its required resources. We use a tuple $\langle$CTA #, Warp #, Regs #, SMem Bytes$\rangle$ to represent the executor. Table 1 presents the executor information in our design for a 128×32 MIMO configuration. Warp #, Regs #, SMem Bytes information can be obtained from the compiler or profiler; while CTA # is specified at the kernel launch based on the MIMO settings. MegaStation uses this resource to provision executors when planning to make the computation placement decision, while the actual allocation happens after instructions are loaded.

Executor Planning. MegaStation preallocates executors when the instruction unit has generated instructions for an incoming frame. We provision executors for all the enqueued instructions. This is a typical bin-packing problem [82, 122], whose goal is to find the minimal amount of GPUs to hold a list of executors without exceeding the capacity of any computing resources of a GPU. Since (a) the GPU pool keeps fluctuating with instructions issued and retired; (b) this process happens pre-execution; (c) resource allocation and partitioning inside GPU is a blackbox, finding an optimal allocation using an ILP (Integer Linear Programming) like solution is unnecessary. MegaStation resorts to an approximate approach following the first-fit strategy [67]. For each instruction, we first search the in-use GPUs and then the unoccupied ones to hold the executor such that the total amount of CTAs (or SMs), warps, registers, and shared memory of the candidate GPU stays within the device limit. This step updates the `GPUStatTable` of the scoreboarding (Figure 7-c).

Different GPU architectures exhibit diverse computing capacities [41]. Newer GPUs could hold more executors simultaneously with more CTAs and global memory. An executor's

Algorithm 1 Structural Analysis.

```
1: Inst_DAG: the DAG within the instruction unit
2: function PLAN_INST_EXECUTOR(cur_inst)
3:     preced_insts ← preced_nodes(Inst_DAG, cur_inst)
4:     est_time ← ESTIM_TIME(cur_inst, preced_insts)
5:     if est_time.start − TS_current > 2T_frame then return Resched.
6:     pred_GPU, pred_SM ← MAX_SM(preced_GPUs, est_time)
7:     if pred_SM > α pred_GPU.total_SM then
8:         return pred_GPU                ▷ Full/Partial Parallelism
9:     max_GPU, max_SM ← MAX_SM(all_GPUs, est_time)
10:    if max_SM < α max_GPU.total_SM then
11:        return delayedQ.push(cur_inst) ▷ Delayed Parallelism
12:    if pred_GPU.wait < data.copy then     ▷ Fragmented Para.
13:        return pred_GPU
14:    else /* including pred_GPU = NULL */
15:        return max_GPU
16: function ESTIM_TIME(cur_inst, preced_insts)
17:    if cur_insts = frame_start then
18:        return ESTIM_FRAME(T_frame)
19:    else if cur_insts = symbol_start then
20:        return ESTIM_SYMBOL(T_symbol)
21:    else
22:        return ESTIM_INST(T_inst)
```

resource tuple may vary depending on the GPU architecture it is compiled for. These all require us to keep track of the instructions' behavior in different GPUs (`OpcodeInfoTable`) and monitor the device limit and planned executors of a GPU (`GPUStatTable`), used by our scoreboarding scheme (§4.5). The scheduler (§4.6) then consults the scoreboarding to decide which instructions run next on which GPUs at runtime.

4.4 Instruction Unit

MegaStation receives IQ samples from the fronthaul link and delivers them to the instruction unit. It performs three tasks: instruction translation, dependency graph construction, and structural analysis. As shown in Figure 7-b, we first decode the frame structure, parse each pilot/uplink/downlink symbol, and then generate instructions for the pipelined tasks. MegaStation attaches a prefix to each instruction to facilitate later analysis: ⟨Frame i, Symbol j⟩+*Inst*.

Next, the instruction unit builds the directed acyclic graph (DAG) among pending instructions to capture the execution dependency. MegaStation explores this from both data and structural perspectives. Regarding data dependency, we construct the DAG graph by analyzing the RAW (read-after-write) relation, where the source operand of instruction $I2$ depends on the destination operand of instruction $I1$. For example, an *Inst*[equalization] can be executed when it receives the beamforming matrix from the preceding *Inst*[zero-forcing] and the output of *Inst*[FFT]. In our case, WAR (write-after-read) and WAW (write-after-write) rarely happen because the destination operands of subsequent tasks along the baseband processing pipeline differ from the source or destination operands of precedent tasks. Since time frames are independent, the DAG is per-frame, whose size is bounded.

For structural analysis, MegaStation analyzes each GPU status to determine its hardware parallelism. According to this analysis, MegaStation decides the GPU on which the executor of instruction should run. Unlike microprocessors, our functional units are the SMs of each GPU and are not fixed. This indicates that an instruction could have multiple executor candidates (§4.3) based on the runtime hardware occupancy. As runtime information fluctuates frequently with instructions being submitted and committed, merely obtaining the latest GPU status at runtime is insufficient. There exists a time gap between when an instruction is issued and when it actually runs on GPUs. In order to predict the GPU status at the time when the instruction is running, we maintain a `GPUStatTable` with the status of all the planned executors (§4.5), which is used to estimate the lifetime of an instruction.

ALG1 dispatches instruction executor to different GPUs based on hardware parallelism within instruction's running time range. To enhance estimation accuracy, MegaStation only schedules instructions that are set to begin within a two-frame time window. MegaStation employs a top-down approach, initially finding a fit for the entire frame before moving on to symbol and then instruction (ALG1: ESTIM_TIME). We prioritize GPUs with input locality, coalescing instructions with dependency. If the minimum number of available SMs during the instruction's running time range is less than threshold partial degree α times the total number of SMs of the GPU (indicating full or partial parallelism), we could safely schedule the instruction to that GPU (ALG1:L7-8). α is GPU architecture-dependent and profiled offline. When adding a new GPU architecture (discussed in §3.2), we vary the partial degree under three execution granularities, measure the uplink/downlink performance (Figure 4), and empirically determine α which yields marginal performance drops (10%).

When there are insufficient SMs in the GPU of precedent nodes or there are no precedent nodes (i.e., a new frame is received), we search for a GPU with the maximum number of free SMs within the instruction's time range. If the newly searched GPU exhibits an adequate number of SMs, it indicates fragmented parallelism. Based on the findings in §3.3, the algorithm must consider the wait time due to the SM scarcity and data transfer time between the GPUs (ALG1: L12-L13). For new incoming frames (no precedent GPUs), we always choose one GPU with the lowest occupancy.

When the newly searched GPU still lacks SM (ALG1: L10-L11), suggesting delayed parallelism, we enqueue the instruction to a delay priority queue where frames are sorted based on their deadlines. All incoming and unscheduled frames are redirected to the delayed queue until there are sufficient available SMs and the delayed queue is empty. Instructions in the delayed queue are assigned to the GPU with minimal occupancy at the frame level granularity and labeled as "delayed". When the frame's processing deadline is violated, it indicates that the incoming baseband processing load exceeds the MegaStation's capacity. All the obsolete frames are discarded

and MegaStation sends back a customized NACK signal that triggers load balancing (4.7). If a consistently heavy load is observed, we would add more GPUs from the current SNC (if there are free ones) or redirect traffic to another SNC.

4.5 The Scoreboarding

MegaStation bookkeeps execution statistics for instructions and GPU hardware using a scoreboarding mechanism. In terms of instruction tracking, as depicted in Figure 7-c, we introduce two tables. One is the `OpcodeInfoTable` that stores executor information regarding each instruction opcode. Each entry in this table contains the resource vector of its corresponding executor for each type of GPU and the average execution latency within a calibrating window. The resource vector of each opcode's executor is obtained from the compiler or profiler prior to MegaStation initialization and remains static unless a new type of GPU is added. In such cases, the resource vector is profiled for that type of GPU before executors can be scheduled to the GPU. Upon completion of an instruction, the execution time is calculated as $T_{inst} = TS_{commit} - \max(TS_{preced_commit}, TS_{exec})$, where TS_{preced_commit} denotes the commit timestamp of instruction's precedent instruction. The execution time (T_{inst}) is averaged within a calibrating window to mitigate fluctuations. Along with the execution time of each instruction, MegaStation also maintains the average execution time of each category of frame (T_{frame}) and symbol (T_{symbol}), and the average scheduling time in pipeline scheduler $T_{sched} = TS_{exec} - TS_{issue}$ in `OpcodeInfoTable`. The other is the `InstStatusTable` that keeps track of timestamps in the **ISSUE**, **EXEC**, and **COMMIT** stages for each instruction. When the instruction unit plans the executor, it records TS_{issue}. Upon submission of the executor by the pipeline scheduler, TS_{issue} is measured. Finally, when the GPU signals completion, TS_{commit} is stored.

Regarding monitoring the utilization of GPU hardware, as shown in Figure 7-c, we maintain a data structure called `GPUStatTable` for each GPU to track the status of each instruction executor deployed to that GPU. This table comprises (a) the number of SMs used by each executor; (b) the estimated start and end timestamp for each executor; and (c) a completion flag indicating the retirement of the executor. The SM usage of an executor is determined by CUDA's occupancy calculator [19], which considers warp, register, and shared memory information to calculate the number of CTAs that an SM can hold. This information helps obtain the number of SMs needed to execute a given instruction. The planned executor is inserted by the instruction unit with the estimated timestamp. The scoreboarding actively queries GPUs about the status of each executor and updates the completion flag.

4.6 Pipeline Scheduler

The pipeline scheduler instructs what to run next in each GPU with a finer granularity than ALGO1. It consults the scoreboarding (§4.5), fetches instructions (baseband processing tasks) from the instruction unit, and deploys them over its GPU, which adjusts the software execution granularity to the dynamic hardware parallelism adaptively at runtime. Each scheduler traverses its instruction queue (§4.4) (Figure 7-a) and fires out instructions using a new algorithm, called LROC (ALG2, Appendix A.2)–a careful synthesis of Least Slack Time First (LSTF) scheduling along with instruction Reordering, Over-commitment and Coalescing techniques.

LSTF Scheduling. As described before, the baseband processing of massive MIMO has stringent performance requirements, especially per-frame processing delay. Thus, we apply the LSTF scheduling discipline to prioritize task execution. MegaStation implements the instruction queue using a red-black tree with a list of keys. The first key is slack timestamp, which is computed with two factors: frame-induced hard deadline (TS_{ddl}), reserved executing time of the instruction and its succedents in the frame along the depth of the DAG (dubbed as $\sum^{Depth} T_{inst}$), where T_{inst} is acquired from the scoreboarding (§4.5) and calibrated online. $TS_{slack} = TS_{ddl} - \sum^{Depth} T_{inst}$.

Reordering. As discussed before (§3), the baseband processing performance is affected by the amount of execution parallelism, parallelism distribution, and data movement overheads. When planning (§4.4), MegaStation predetermines the executor of an instruction based on overall available resources across GPUs. When executing, it adjusts the instruction order with runtime status of the local GPU for performance maximization. The second key of the instruction queue is SM usage of each instruction. Our algorithm prioritizes scheduling instructions with the least SM usage to greedily occupy the remaining GPU resources after handling urgent instructions. In this way, it opportunistically reorders instructions with no dependency to improve GPU utilization.

Over-commitment. MegaStation strives to keep the GPU busy (i.e., work-conserving) and reduce the idle cycles when waiting for the host to launch CUDA kernels if the GPU's SMs become available. Our scheduler thus allows computation overloading, inevitably introduces delayed parallelism, and resolves it through the stream priority mechanism provided by CUDA. After the GPU is full, MegaStation schedules an instruction labeled as "delayed" and submits it to a low-priority CUDA stream. This allows the instruction to run whenever the GPU has available resources without interfering with higher-priority normal instructions.

Coalescing. To avoid the synchronization gap between GPU completion and CPU being signaled, MegaStation provision instruction scheduling with the help of the sequential property within a single CUDA stream. If an instruction has precedent instructions on the local GPU that require a larger resource than the current instruction's executor, we preemptively submit the current instruction to the same CUDA stream with the precedent instructions without waiting for the completion signal. This preemptive submission has minimal impact on future scheduling, as the current instruction can inherit resources reserved by the precedent executor on the GPU.

4.7 Deployment and Failure Handling

Fault tolerance is crucial in practice. We apply Slingshot's approach [89] to handle failures, i.e., treating short-term loss of frames as bad signal quality effects in the wireless environment and recovering them through inherent cellular network impairments. All discarded frames in MegaStation are handled similarly to Slingshot [89]. Since frames are independent, some frame loss would not affect other frames' processing.

There are three failure domains in MegaStation: server host, interconnect fabric, and GPU pool. When the host fails, MegaStation enters fail-stop mode. All pending frames are discarded and incoming frames should be reissued to another backup vRAN server. For the interconnect failure, MegaStation loses connection with all the GPUs. One option is to use the same solution as host failure, or alternatively fall back to a CPU-only mode with a slower frame processing rate.

MegaStation depends on GPU drivers to detect GPU failure. If a GPU malfunctions, device exceptions are raised when trying to submit instructions or query instruction status, causing it to be excluded from the scoreboard. Any running and pending (uncommitted) instructions on the GPU are moved to the instruction queues for rescheduling. In the meantime, a daemon continuously monitors PCIe device lists for new GPUs. Once a new functional GPU is detected, it is installed by MegaStation to the scoreboard for instruction scheduling.

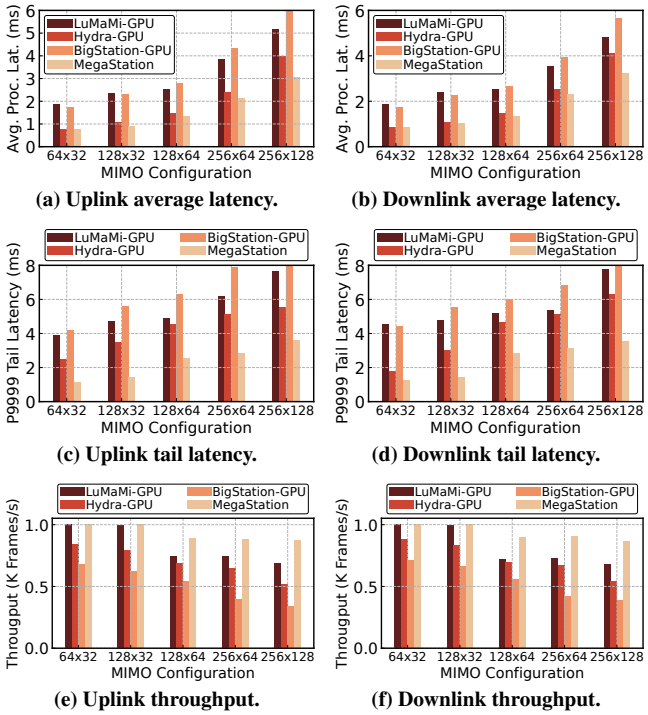
4.8 Implementation Details

We implement MegaStation atop Agora [65] and NVIDIA Aerial SDK [30] in C++ with 8372 LOC and CUDA with 2147 LOC. Regarding the baseband processing instruction, MegaStation uses CUTLASS [22] and cuSOLVER [21] when computing the CSI matrices, and cuFFT [26] for FFTs. For LDPC encoding and decoding, we use the NVIDIA Aerial SDK [30] and enhance it with inline data copy. We reimplement precoding, equalization, modulation, and demodulation functions in CUDA following Agora's algorithms. Our executors leverage the GPU hardware features. For example, we use tile-based iterators [120] to fuse data transformation into the data loading and storing stage. We develop a lightweight mechanism for instruction synchronization, where each instruction allocates a flag on GPUs. These flags are manipulated via CUDA stream operation [20] on GPU side and are synchronized via GDRCopy [27] to CPU side. MegaStation employs a master-slave architecture. A master thread realizes the instruction unit, where per-GPU slave threads run the pipeline scheduler. Each slave thread also manages a pool of memory blocks, which translates the operand label to the actual memory pointer. Appendix A.3 presents more details.

5 Evaluation

5.1 Experimental Methodology

Hardware testbed. We use the GigaIO FabreX-based SuperNODE [8, 44] as the single-node supercomputer target.



(a) Uplink average latency. (b) Downlink average latency.

(c) Uplink tail latency. (d) Downlink tail latency.

(e) Uplink throughput. (f) Downlink throughput.

Figure 8: Average/tail uplink/downlink per-frame processing latency and throughput comparing five MIMO settings.

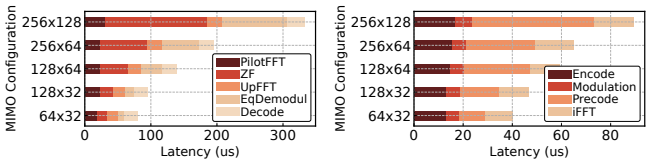
The host server is a 2U Dell R740 box, consisting of two 32-core Intel Xeon Gold 6248 processors (running at 2.5GHz), 192GB DRAM, and 1.92TB HDD. The load/store fabric uses routable PCIe [2], whose details are described in §2.4. We use NVIDIA V100 and A100 GPUs in the composable pool.

Comparison baselines. We compare MegaStation with three baselines: LuMaMi [100], Hydra [71], BigStation [121]. Since none of these prior systems target the GPU pooling, we develop the baseline over our target following their proposed design. The first one (*LuMaMi-GPU*) performs frame-level parallel execution and distributes frames to available GPUs in a work-conserving manner. The second one (*Hydra-GPU*) parallelizes the pipeline at the symbol level. Akin to Hydra [71], we place dependent stages in the same GPU to minimize the cross-GPU communication overheads. The third one (*BigStation-GPU*) implements task-level parallelism, where tasks are distributed to different executors.

Workload. We use the Agora [65] traffic generator to emulate an RU (radio unit). It applies the Rayleigh fading channel and transmits time-domain IQ samples. Our experiments use 2048 subcarriers and consider different MIMO configurations.

5.2 End-to-end Performance

We set up 1 RU under 5 massive MIMO configurations (i.e., 64×32, 128×32, 128×64, 256×64, 256×128) and launched the experiment over just one chassis box with A100 GPUs. We measure the average and P9999 (tail) processing latency, and overall throughput for both 1ms uplink and downlink frames. As shown in Figure 8, MegaStation meets



(a) Breakdown on uplink path. **(b) Breakdown on downlink path.**

Figure 9: Latency breakdown for five MIMO settings.

the deadline requirements, with P9999 latency ranging from 1.2 to 3.6 ms for uplink frames, and 1.2 to 3.5 ms for downlink frames across different MIMO settings. We satisfy the stipulated 4m deadline for 1ms frames.

Across 5 MIMO cases, MegaStation achieves 50.6/58.9%, 12.2/46.9%, 53.7/66.2% lower average/tail latency on average for the uplink frames compared with *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU* approaches, and 45.2/57.1%, 9.5/41.6%, and 47.9/61.9% for the downlink ones, respectively. For throughput, MegaStation performs 10.7/12.2%, 25.3/22.7%, and 44.9/41.8% better than three baselines on average for the uplink/downlink frames. MegaStation outperforms *BigStation-GPU* the most because the latter incurs too much cross-task synchronization. *LuMaMi-GPU* and *Hydra-GPU* perform suboptimally mainly because (a) their coarse-grained execution cannot fully leverage all available SMs promptly, especially when the GPU parallelism becomes irregular; (b) the scheduling mechanism is workload-agnostic, ignoring data locality and deadlines. Both are captured in our structural analysis, which places executors on suitable GPUs, and pipeline scheduler, which reorders instructions based on slack time and coalesces them with data locality.

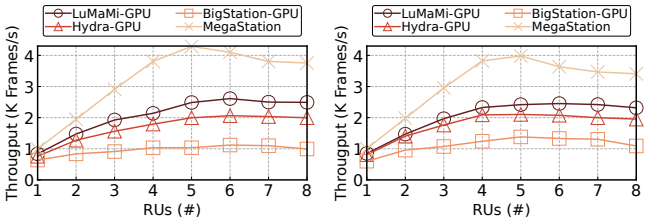
5.3 Uplink/Downlink Pipeline Latency Breakdown

We break down the uplink and downlink processing latency of MegaStation (Figure 9). Overall, zero-forcing is the most expensive stage, consuming 36.3% of the whole pipeline on average across five MIMO cases. However, it only happens for the pilot symbols which could be amortized by uplink/downlink symbols. The next time-consuming stage is EqDemodul of the uplink path and Precoding of the downlink path, which spends 44.2% and 42.7%. We also compare the GPU executor with the CPU one (Agora [65]). Our experiment chooses the 64×16 MIMO, the largest one supported by Agora [65]. The CPU one uses 60 cores, while we use a single V100 GPU in MegaStation. The results (Figure 13 in Appendix A.4) demonstrate that the GPU can achieve an average speedup of $10\times$ compared to the CPU due to its massive parallelism.

5.4 Scalability

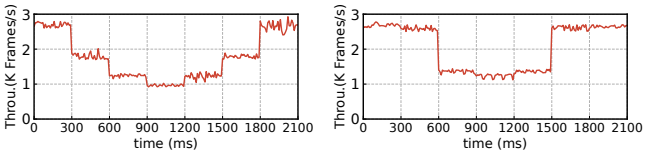
We increase the number of A00 GPUs on the SNC and analyze scalability in two dimensions: (a) how MegaStation performs as the number of RUs increases, and (b) how MegaStation scales with an increasing number of GPUs.

RU Scalability. We gradually increase the number of RUs from 1 to 8 under 128×64 MIMO for both uplink and downlink frames. Figure 10 presents the throughput per-



(a) Uplink throughput. **(b) Downlink throughput.**

Figure 10: We report the frame throughput as increasing the number of consolidated RUs for the 128×64 MIMO.



(a) 3 RUs 128×64. **(b) 3 RUs 128×32.**

Figure 11: We adjust the number of GPUs from 5 to 2 and back to 5 every 300 ms while measuring the throughput every 10 ms.

formance of four approaches. MegaStation scales the best and can harvest all the available SMs for frame processing. Over uplink/downlink frames, MegaStation achieves $1.5/1.4\times$, $1.8/1.6\times$, $3.2/2.6\times$ higher performance compared with *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU*. They scale poorly due to the emergence of irregular hardware parallelism when more RUs send frames concurrently, yielding inefficient parallel execution and lower performance. MegaStation scales nearly linearly with the number of RUs regarding throughput. This is because MegaStation uses scoreboarding to track the real-time status of each GPU, allowing our scheduler to allocate tasks based on GPU availability and effectively manage congestion. Also, the over-commitment technique further helps exploit GPU computing power in the delayed parallelism and minimizes GPU idle cycles.

The number of RUs cannot be scaled infinitely. It is bottlenecked by the fabric interconnect between the host and chassis. The maximum number of RUs is theoretically bounded by $\frac{F \times S \times RU\#}{B} < T$, where F is frame rate (frames per second), S is frame size (bytes per frame), B is the bandwidth of interconnect (bytes per second), and T is frame duration (seconds). When the interconnect saturates, adding more GPUs would not help. One could load balance traffic among multiple SNCs via a fronthaul switch to solve this problem [89].

GPU Scalability. We add the number of GPUs from 1 to 5 under 3 RRUs under a 128×64 MIMO (Figure 14 in Appendix A.4). MegaStation achieves $1.9/1.8\times$ speedup for uplink/downlink frames. However, *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU* only gain $1.3/1.2\times$, $0.9/1.0\times$, and $0.6/0.7\times$ improvement, respectively. The performance drop of *LuMaMi-GPU* is attributed to its coarse-grained execution granularity, which fails to fully utilize the increasing SM resources. *Hydra-GPU* and *BigStation-GPU* exhibit limited speedups due to higher communication overhead introduced by distributing symbols or tasks across more GPUs. MegaStation addresses these problems by (a) scheduling at

task-level granularity, which is the finest granularity in baseband processing; (b) in structural analysis, taking data transfer time into consideration when distributing instructions with dependency to different GPUs; (c) implementing coalescing technique to avoid GPU waiting for instructions.

5.5 MegaStation Drill-Down

We evaluate each system component in MegaStation via instruments and report their CDF. As shown in Figure 15 (Appendix A.4), we find that 90% of the instruction unit (IU) execution takes less than 12 μs, indicating that it is not the system bottleneck even if it is single-threaded. The per-GPU pipeline scheduler (PS) is 1.3$\times$ more costly than IU because it interacts with the CUDA driver on submitting instructions.

The pilot symbol processing is the most consuming part (Figure 15-b/c), where 90% of the execution takes 0.3ms to compute zero forcing and 0.12ms to transfer the zero-forcing (ZF) results to other GPUs. To accelerate ZF under large MIMO, we split the ZF instruction into multiple sub-instructions based on frequency bins and distribute them to multiple GPUs. Subsequent instructions would then have a dependency on the sub-instructions, captured in the DAG. To amortize large GPU-GPU communication, we balance irregular parallelism-induced waiting time and data shuffling overheads when copying tasks to idle GPUs.

5.6 Discussion

Reliability. There are three failure domains in MegaStation (§4.7). We manually toggle GPU availability in the GigaIO FabreX control panel to observe system response to the GPU fluctuation. We alternate between enabling and disabling GPUs, transitioning from 5 to 2 and back to 5 every 300 milliseconds. We configure 3 RRUs under the 128×64 and 128×32 MIMO and measure the throughput every 10 ms. The system throughput decreases with the number of GPUs, but returns to its original level when GPUs are restored to the initial setting (Figure 11). The scoreboarding scheme allows monitoring the status of each GPU in real time, preventing further scheduling instructions on unavailable GPUs.

Heterogeneous GPUs. We examine how performance is affected when utilizing heterogeneous GPUs. We compare A100+V100 GPUs vs. A100-Only. We show that *LuMaMi-GPU*, *Hydra-GPU*, and *BigStation-GPU* experience a 23.1%, 30.7%, and 32.8% average latency increase (Appendix A.4), respectively, while MegaStation only observes a 16.1% increase. Our method could handle heterogeneity better for two reasons. First, MegaStation takes different GPU architectures into account and dynamically adjusts its scheduling policy based on the available SMs. Second, MegaStation could capture the increased data movement overhead during profiling and rebalance the waiting and communication overhead.

In Appendix A.4, we show how MegaStation executes under small MIMOs, operates under other GPUs, and tackles the fragmented parallelism issue.

6 Related Work

Quantum Acceleration. Researchers have explored using quantum computing to tackle the high computing demand of massive MIMO. QuAMax [86] leverages quantum annealing to build the Sphere Decoder-based maximum likelihood (ML) MIMO decoders/detectors. HyPD [83] develops a hybrid classical-quantum decoder for polar error correction codes, where it employs CMOS processing for the decoder's binary tree traversal and quantum annealing processing for the polar decoder. Minsung Kim *et al.* [85] investigate a spin-level preprocessing approach for fine-grained decomposition that enables flexible parallel quantum signal processing.

Infrastructure Composability. Emerging high-bandwidth networks [1,16] and cluster interconnects [2,40,54,78] enable physical resource pooling. For example, [91,94,101,113,125] develop object/paging-based remote memory over CXL memory expanders. [57,74,80,87,88,92,96,102–104,126] build disaggregated storage. Researchers also develop composable accelerators [29,59,60] and explore efficient host-accelerator collaborated computation. The single-node supercomputer is another composable platform and we use it for accelerating the baseband processing of massive MIMO.

Network Function. Our work benefits from prior pioneering efforts of accelerating networking functions. RouteBricks [66] parallelizes router functionalities across multiple servers and multiple cores within a single server. PacketShader [75] examines using a commodity GPU to build a software router. NBA [84] introduces system techniques to optimize NF workloads on a CPU-GPU heterogeneous system. [61,69,73,77,90,93,95,97,98,107–112,124] offload networking functions onto reconfigurable switches and SmartNICs. MegaStation follows the past strategy and dynamically exploits the GPU's hardware parallelism at a fine granularity.

7 Conclusion

This paper makes a case for using an emerging single-node supercomputer for the massive MIMO baseband processing. We build MegaStation, an application-platform co-designed system that dynamically adjusts the execution granularity and reconstructs the baseband processing pipeline on the fly to accommodate the irregular underlying hardware parallelism. We model the single-node supercomputer as a tightly coupled microprocessor and employ a scoreboarding-like scheme to orchestrate baseband processing computations. Real system-based evaluations show that MegaStation outperforms comparable state-of-the-art approaches considerably and can fully leverage the capabilities of this new computing platform.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Wenjun Hu, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, and CAREER-2339755.

References

[1] 800G Specification. https://ethernettechnologyconsortium.org/wp-content/uploads/2021/10/Ethernet-Technology-Consortium_800G-Specification_r1.1.pdf, 2021.

[2] FabreX PCIe network fabric: a primer. https://gigaio.com/wp-content/uploads/2021/02/FabreX-PCIe-network-fabric-a-primer-1.pdf, 2021.

[3] Huawei's 5G Indoor Distributed Massive MIMO. https://www.huawei.com/en/news/2021/9/ubiquitous-gigabit-5g-indoor-dmm-wins-award, 2021.

[4] Mavenir and NEC deploy massive MIMO in France. https://www.lightreading.com/open-ran/mavenir-and-nec-deploy-massive-mimo-in-france, 2022.

[5] Nokia and AT&T collaborating to improve 5G uplink with distributed massive MIMO. https://www.nokia.com/about-us/news/releases/2022/02/28/nokia-and-att-collaborating-to-improve-5g-uplink-with-distributed-massive-mimo-mwc22/, 2022.

[6] The 5G Network Backbone: A Guide to Small Cell Technology. https://www.telit.com/blog/5g-networks-guide-to-small-cell-technology/, 2022.

[7] Enfabrica's Accelerated Compute Fabric. https://blog.enfabrica.net/press-release-enfabrica-raises-125-million-series-b-to-fuel-ramp-of-ai-infrastructure-networking-a8a0b21653d2, 2023.

[8] GigaIO's FabreX System. https://gigaio.com/products/fabrex-system-overview/, 2023.

[9] GroqRack Compute Cluster. https://groq.com/wp-content/uploads/2022/10/GroqRackâĎć-Compute-Cluster-Product-Brief-v1.0.pdf, 2023.

[10] H3's Falcon System. https://www.h3platform.com/solution/composable-ai, 2023.

[11] Introducing Azure Operator Nexus. https://techcommunity.microsoft.com/t5/azure-for-operators-blog/introducing-azure-operator-nexus/ba-p/3753393, 2023.

[12] KDDI kicks off deployment of 5G Open vRAN sites in Japan. https://www.datacenterdynamics.com/en/news/kddi-kicks-off-deployment-of-5g-open-vran-sites-in-japan/, 2023.

[13] Liqid's SmartStack System. https://www.liqid.com/products/gpu-on-demand, 2023.

[14] T-Mobile's 5G network gets capacity boost from MU-MIMO: report. https://www.fiercewireless.com/tech/t-mobiles-5g-network-gets-capacity-boost-mu-mimo-report, 2023.

[15] The 5G vRAN solution from Fujitsu. https://www.fujitsu.com/global/about/resources/news/press-releases/2023/0220-01.html, 2023.

[16] Ultra Ethernet. https://ultraethernet.org/wp-content/uploads/sites/20/2023/10/23.07.12-UEC-1.0-Overview-FINAL-WITH-LOGO.pdf, 2023.

[17] 3GPP Release 17. https://www.3gpp.org/specifications-technologies/releases/release-17, 2024.

[18] 5G UE Data Rate. https://devopedia.org/5g-ue-data-rate, 2024.

[19] CUDA Occupancy Calculator. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator, 2024.

[20] CUDA Stream Memory Operations. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__MEMOP.html, 2024.

[21] cuSOLVER. https://docs.nvidia.com/cuda/cusolver, 2024.

[22] CUTLASS 3.5. https://github.com/NVIDIA/cutlass, 2024.

[23] FA4003–FabreX PCIe Gen3 Adapter. https://gigaio.com/wp-content/uploads/2022/02/GIO_NetworkAdapterCard_G3-V4-2.pdf, 2024.

[24] FA4004–FabreX PCIe Gen4 Adapter. https://gigaio.com/products/fabrex-network-adapter-card/, 2024.

[25] FabreX Gen3 Top of Rack PCIe Switch Hyper-Performance Network. https://gigaio.com/wp-content/uploads/2022/02/GIO_FabreX_TOR-Gen3-V4-1.pdf, 2024.

[26] Fast Fourier Transform for NVIDIA GPUs. https://developer.nvidia.com/cufft, 2024.

[27] GDRCopy. https://github.com/NVIDIA/gdrcopy, 2024.

[28] Intel FlexRAN Reference Architecture for Wireless Access. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html, 2024.

[29] Leo CXL Smart Memory Controllers. https://www.asteralabs.com/products/leo/leo-cxl-memory-connectivity-controllers/, 2024.

[30] NVIDIA Aerial SDK. https://developer.nvidia.com/aerial-sdk, 2024.

[31] NVIDIA DGX Platform. https://www.nvidia.com/en-us/data-center/dgx-platform/, 2024.

[32] NVIDIA HGX AI Supercomputer. https://www.nvidia.com/en-us/data-center/hgx/, 2024.

[33] RB4082–Accelerator Pooling Appliance. https://gigaio.com/products/accelerator-pooling-appliance/, 2024.

[34] RS4024–Top of Rack PCIe Switch (Gen 4). https://gigaio.com/products/top-of-rack-pcie-switch/, 2024.

[35] SoftBank Corp., NEC and Broadcom Jointly Validate RAN Modernization with Virtualization by Unifying O-RAN Architecture and Telco Cloud. https://www.broadcom.com/company/news/product-releases/61881, 2024.

[36] The O-RAN Alliance. https://www.o-ran.org, 2024.

[37] The SFF-8644 connector. https://cs-electronics.com/sff-8644/, 2024.

[38] Verizon advances O-RAN technology by deploying 130,000 new O-RAN capable radios in its network. https://www.verizon.com/about/news/verizon-advances-o-ran-technology, 2024.

[39] 5G Sites infrastructure. https://www.ericsson.com/en/ran/5g-sites-infrastructure, 2025.

[40] Computer Express Link Specification. https://computeexpresslink.org/cxl-specification/, 2025.

[41] CUDA Compute Capabilities. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities, 2025.

[42] Ericsson Antenna System. https://www.ericsson.com/en/portfolio/networks/ericsson-radio-system/antenna-system, 2025.

[43] Ericsson Cloud RAN. https://www.ericsson.com/en/ran/cloud, 2025.

[44] GigaIO SuperNODE. https://gigaio.com/supernode/, 2025.

[45] H3C BBU5200 Baseband Unit. https://www.h3c.com/en/Products_and_Solutions/InterConnect/Moblie_Communication/Products/Access/BBU/BBU5200/, 2025.

[46] Huawei DBS5900 Distributed Base Stations. https://e.huawei.com/en/products/wireless/base-station/dbs5900, 2025.

[47] Nokia AirScale Baseband Solutions. https://www.nokia.com/networks/mobile-networks/airscale-radio-access/baseband/, 2025.

[48] NVIDIA GPUDirect: Enhancing Data Movement and Access for GPUs. https://developer.nvidia.com/gpudirect, 2025.

[49] Samsung Baseband Unit. https://www.samsung.com/global/business/networks/products/radio-access/baseband/, 2025.

[50] The Intel vRAN Solution. https://www.intel.com/content/www/us/en/wireless-network/5g-network/radio-access-network.html, 2025.

[51] The Open Virtualized RAN from Mavenir. https://www.mavenir.com/portfolio/mavair/radio-access/vran/, 2025.

[52] The Samsung vRAN Solution. https://www.samsung.com/global/business/networks/products/radio-access/virtualized-ran/, 2025.

[53] The Software-Defined Virtualized RAN (vRAN) from Parallel Wireless. https://www.parallelwireless.com/products/software-defined-virtualized-ran/, 2025.

[54] Ultra Accelerator Link. https://www.ualinkconsortium.org, 2025.

[55] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

[56] Aslan, Yanki and Roederer, Antoine and Fonseca, Nelson JG and Angeletti, Piero and Yarovoy, Alexander.

Orthogonal versus zero-forced beamforming in multi-beam antenna systems: Review and challenges for future wireless networks. *IEEE Journal of Microwaves*, 1(4):879–901, 2021.

[57] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 49–67, 2023.

[58] Manu Bansal, Aaron Schulman, and Sachin Katti. Atomix: A Framework for Deploying Signal Processing Applications on Wireless Infrastructure. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[59] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*, page 37–44, 2022.

[60] Boles, David and Waddington, Daniel and Roberts, David A. Cxl-enabled enhanced memory functions. *IEEE Micro*, 43(2):58–65, 2023.

[61] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[62] Yongce Chen, Y. Thomas Hou, Wenjing Lou, Jeffrey H. Reed, and Sastry Kompella. O-M3: Real-Time Multi-Cell MIMO Scheduling in 5G O-RAN. *IEEE Journal on Selected Areas in Communications*, 42(2):339–355, 2024.

[63] Özlem Tuğfe Demir, Emil Björnson, and Luca Sanguinetti. Channel modeling and channel estimation for holographic massive MIMO with planar arrays. *IEEE Wireless Communications Letters*, 11(5):997–1001, 2022.

[64] Ruoqi Deng, Boya Di, Hongliang Zhang, H Vincent Poor, and Lingyang Song. Holographic MIMO for LEO satellite communications aided by reconfigurable holographic surfaces. *IEEE Journal on Selected Areas in Communications*, 40(10):3071–3085, 2022.

[65] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. Agora: Real-time massive MIMO baseband processing in software. In *Proceedings of the 16th international conference on emerging networking experiments and technologies (CoNext'20)*, pages 232–244, 2020.

[66] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, page 15–28, 2009.

[67] György Dósa and Jirí Sgall. First Fit bin packing: A tight analysis. In *30th International symposium on theoretical aspects of computer science (STACS'2013)*, 2013.

[68] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231, 2010.

[69] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 51–66, 2018.

[70] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[71] Junzhi Gong, Anuj Kalia, and Minlan Yu. Scalable Distributed Massive {MIMO} Baseband Processing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 405–417, 2023.

[72] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 485–500, 2019.

[73] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, page 916–929, 2023.

[74] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 1012–1027, 2023.

[75] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM'10)*, page 195–206, 2010.

[76] Bowen He, Xiao Zheng, Yuan Chen, Weinan Li, Yajin Zhou, Xin Long, Pengcheng Zhang, Xiaowei Lu, Linquan Jiang, Qiang Liu, Dennis Cai, and Xiantao Zhang. DxPU: Large-scale Disaggregated GPU Pools in the Datacenter. *ACM Trans. Archit. Code Optim.*, 20(4), dec 2023.

[77] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, page 33–47, 2023.

[78] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 297–312, 2024.

[79] Christopher Husmann, Georgios Georgis, Konstantinos Nikitopoulos, and Kyle Jamieson. FlexCore: Massively Parallel and Flexible Processing for Large MIMO Access Points. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017.

[80] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

[81] Yuki Iida, Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, and Shinpei Kato. GPUrpc: Exploring Transparent Access to Remote GPUs. *ACM Trans. Embed. Comput. Syst.*, 16(1), oct 2016.

[82] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[83] Srikar Kasi, John Kaewell, and Kyle Jamieson. A Quantum Annealer-Enabled Decoder and Hardware Topology for NextG Wireless Polar Codes. *IEEE Transactions on Wireless Communications*, pages 1–1, 2023.

[84] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems (Eurosys'15)*, 2015.

[85] Minsung Kim and Kyle Jamieson. Finer-Grained Decomposition for Parallel Quantum Mimo Processing. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, 2023.

[86] Minsung Kim, Davide Venturelli, and Kyle Jamieson. Leveraging quantum annealing for large MIMO processing in centralized radio access networks. In *Proceedings of the ACM Special Interest Group on Data Communication (Sigcomm'19)*, page 241–255, 2019.

[87] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, 2016.

[88] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 345–359, 2017.

[89] Nikita Lazarev, Tao Ji, Anuj Kalia, Daehyeok Kim, Ilias Marinos, Francis Y. Yan, Christina Delimitrou, Zhiru Zhang, and Aditya Akella. Resilient Baseband

Processing in Virtualized RANs with Slingshot. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, 2023.

[90] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, page 1–14, 2016.

[91] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, pages 574–587, 2023.

[92] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, page 591–605, 2020.

[93] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[94] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[95] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, page 318–333, 2019.

[96] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 305–323, 2020.

[97] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 795–809, 2017.

[98] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 363–378, 2019.

[99] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 289–304, 2020.

[100] Steffen Malkowsky, Joao Vieira, Liang Liu, Paul Harris, Karl Nieman, Nikhil Kundargi, Ian C Wong, Fredrik Tufvesson, Viktor Öwall, and Ove Edfors. The world's first real-time testbed for massive MIMO: Design, implementation, and validation. *IEEE Access*, 5:9073–9088, 2017.

[101] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 3*, pages 742–755, 2023.

[102] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, page 106–122, 2021.

[103] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[104] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage*, 20(3), June 2024.

[105] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, page 595–610, 2023.

[106] Sreepathi Pai. How the Fermi Thread Block Scheduler Works (Illustrated). https://www.cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html, 2014.

[107] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 663–679, 2018.

[108] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, page 16–22, 2020.

[109] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 772–787, 2021.

[110] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 740–755, 2021.

[111] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 1–16, 2018.

[112] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 685–699, 2020.

[113] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, pages 105–121, 2023.

[114] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: High Performance Software Radio Using General Purpose Multi-core Processors. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, 2009.

[115] James E Thornton. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, pages 33–40, 1964.

[116] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[117] Li Wei, Chongwen Huang, George C Alexandropoulos, EI Wei, Zhaoyang Zhang, Mérouane Debbah, and Chau Yuen. Multi-user holographic MIMO surfaces: Channel modeling and spectral efficiency analysis. *IEEE Journal of Selected Topics in Signal Processing*, 16(5):1112–1124, 2022.

[118] Jun Wu, Zhifeng Zhang, Yu Hong, and Yonggang Wen. Cloud radio access network (c-ran): a primer. *IEEE network*, 29(1):35–41, 2015.

[119] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 533–548, 2020.

[120] Xincheng Xie, Junyoung Kim, and Kenneth Ross. Gamut: Matrix multiplication-like tasks on gpus. In *14th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS'23)*, 2023.

[121] Qing Yang, Xiaoxiao Li, Hongyi Yao, Ji Fang, Kun Tan, Wenjun Hu, Jiansong Zhang, and Yongguang Zhang. BigStation: enabling scalable real-time signal processingin large mu-mimo systems. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (Sigcomm'13)*, page 399–410, 2013.

[122] Andrew Chi-Chih Yao. New algorithms for bin packing. *Journal of the ACM (JACM)*, 27(2):207–227, 1980.

[123] Taesang Yoo and Andrea Goldsmith. Optimality of zero-forcing beamforming with multiuser diversity. In *IEEE International Conference on Communications (ICC)*, volume 1, pages 542–546, 2005.

[124] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM'20)*, pages 283–295, 2020.

[125] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 658–674, 2023.

[126] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

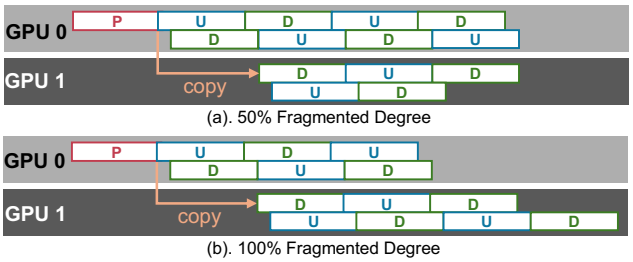


Figure 12: Symbol-level parallelism under fragmented parallelism with different fragment degrees. P/U/D=Pilot/Uplink/Downlink.

Executors	CTAs (#)	Warps (#)	Regs (#)	SMem (Byte)
PilotFFT	128	4	56	8448
UpFFT	128	4	56	8448
iFFT	128	4	56	8448
ZF	4864	32	30	8448
EqDemodul	1216	1	30	0
Modul	1216	1	18	0
Precode	4864	1	30	0
Decode	32	4	168	14144
Encode	32	8	56	843

Table 1: Resource tuples of MegaStation executors in a 128×32 MIMO configuration. We use NVIDIA terminology. CTAs=The number of cooperative thread array (i.e., thread block) per kernel. Warps=The number of warps per CTA. Regs=The number of registers per thread. SMem=Shared memory size per CTA.

A Appendix

A.1 More Fragmented Parallelism Discussion

Under a low fragment degree (Figure 12-a), GPU0 (with the pilot symbol) has more symbols than GPU1 due to the round-robin strategy. GPU0's computation time exceeds the aggregated execution cost of data copying and GPU1's computation. As the fragment degree increases (Figure 12-b), symbol distribution becomes more balanced. GPU0 becomes idle during non-overlapping time intervals when its computing phase interleaves with the data copy and execution phase on GPU1.

A.2 MegaStation Details

Table 1 displays the logical representation of executors. The values for Wraps #, Regs #, and SMem are compiler-specific and remain constant regardless of kernel runtime. However, CTAs # is determined at runtime based on MIMO configuration. MegaStation calculates CTAs # and pass it to the kernel when kernel is launching.

The LROC algorithm described in ALG 1 illustrates the scheduling process in our pipeline scheduler. It starts by prioritizing the most urgent instructions. When there is available space left in the GPU, it continues by reordering instructions whose precedents have been completed. Once the GPU is fully utilized, the algorithm assigns instructions marked as "delayed" to a lower priority stream. Additionally, the algorithm takes advantage of opportunities to combine instructions with dependencies if precedent instructions have already allocated larger GPU resources.

Algorithm 2 LROC Algorithm.

```
1:  procedure LROC_SCHEDULER
2:      min_inst = instQ.min();
3:      while min_inst.TS_slack − TS_cur < T_thr do          ▷ LSTF
4:          SUBMIT(min_inst, preced_stream(min_inst))
5:          min_inst = instQ.next_min()
6:      for inst ∈ instQ do                                  ▷ Reorder
7:          if GPU.avail_SM ≤ 0 then break
8:          if pred_insts.finish & inst.SM ≤ GPU.avail_SM then
9:              SUBMIT(inst, pred_insts.stream)
10:     for inst ∈ instQ do                                  ▷ Over-commit
11:         if inst.label = delayed then
12:             SUBMIT(inst, low_prior_stream)
13:             break
14:     for inst ∈ instQ do                                  ▷ Coalesce
15:         if pred_insts.running & inst.SM ≤ pred_insts.SM then
16:             SUBMIT(inst, pred_insts.stream)
```

A.3 More MegaStation Implementation

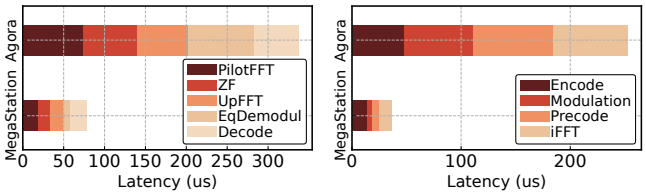
To take advantage of GPU's shared memory and reduce the number of CUDA kernels within instructions, we use tile-based iterators [120] to fuse data transformation into the data loading and storing stage of a kernel computation. In this way, intermediate results are not written back to the global memory and thus data read/write round trips are reduced.

We develop a lightweight mechanism for instruction synchronization. We register and manage a 64KB buffer inside each GPU and map it to host memory. Upon an incoming frame, MegaStation allocates a free array ($sizeof(array) = instruction\#_in_frame \times 4B$) from the buffer, where the CPU actively fetches the array via GDRCopy [27] during the frame execution. Each element in the array is a completion flag of the instruction. When an instruction is finished, we use the CUDA stream operation [20] (`cuStreamWriteValue`) to signal the completion by writing the completion status word to the corresponding place. To enforce the execution order of two dependent instructions in two different CUDA streams, we insert `cuStreamWaitValue` before the second instruction to wait for the completion of the first one without blocking the GPU execution engine.

MegaStation employs a master-slave architecture. A master thread realizes the instruction unit, where per-GPU slave threads run pipeline scheduler and track the status of instructions. To minimize allocation overhead, MegaStation reserves memory for every type of operand in each GPU and manages them in a free list on the host side. Similar to the idea of register renaming, our memory manager translates the operand labels to the actual memory pointer before submitting the instruction to GPU. An exception will be triggered when the free list is empty, where we either allocate a new chunk or signal the scheduler to block.

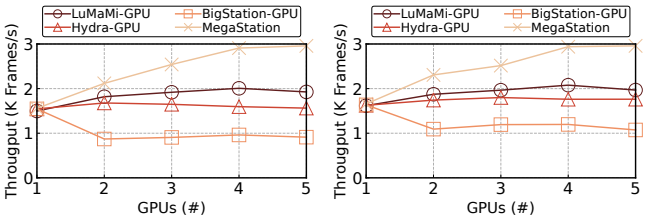
A.4 More MegaStation Evaluation

GPU v.s. CPU Executor. Figure 13 compares the performance between GPU executors (MegaStation) and CPU ex-



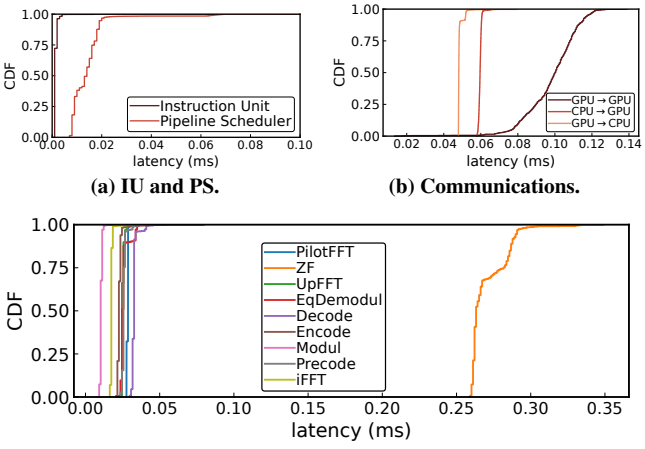
(a) Breakdown on uplink path. (b) Breakdown on downlink path.

Figure 13: Latency breakdown comparing MegaStation (GPU) and Agora [65] (CPU).



(a) Uplink throughput. (b) Downlink throughput.

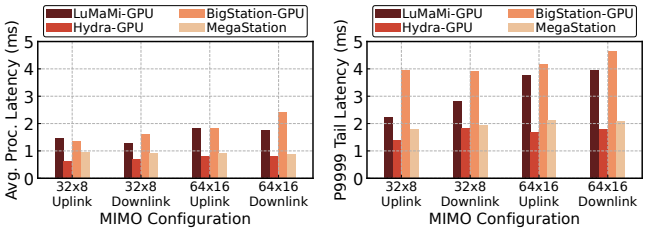
Figure 14: We report the frame throughput as increasing the number of GPUs for 3 RUs (128×64 MIMO).



(a) IU and PS. (b) Communications.

(c) Executors.

Figure 15: Performance of different components in MegaStation under the 128×32 MIMO setting. IU=Instruction Unit. PS=Pipeline Scheduler.



(a) Average Latency. (b) P9999 Tail Latency.

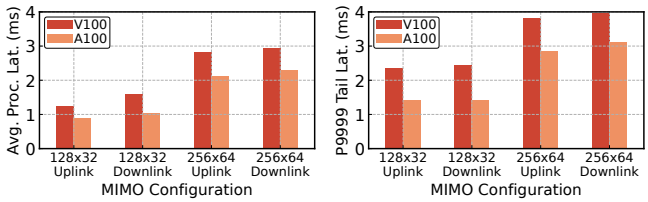
Figure 16: Average/Tail Latency for small MIMOs.

ecutors (Agora [65]).

GPU Scaling. Figure 14 explores how MegaStation scales with an increasing number of GPUs.

MegaStation Drill-down. Figure 15 reports the CDF of different system components in MegaStation.

Small MIMOs. We investigate the performance of MegaStation under small MIMO configurations (32×8, 64×16 MIMO). Using the same hardware setup described in §5.2, we depict the average and tail latency of five different methods in Figure 16. Our results show that in contrast to large MIMO scenarios, *Hydra-GPU* performs the best. Compared with MegaStation, it achieves a 20.1%/15.6% improvement on average in average/tail latency due to the lower scheduling overhead of static methods. *Hydra-GPU* outperforms *LuMaMi-GPU*/*BigStation-GPU* because of the finer-grained execution and fewer data synchronizations, respectively.



(a) Average Latency. (b) P9999 Tail Latency.

Figure 17: Average/Tail Latency on V100/A100.

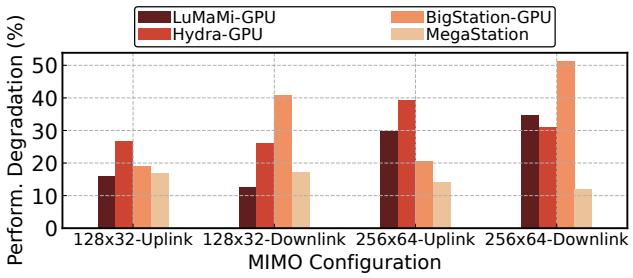
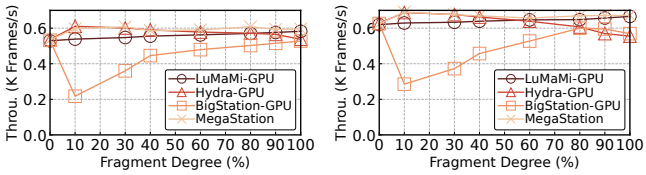


Figure 18: Performance drop on V100+A100 with respect to results on two A100s



(a) Uplink throughput. (b) Downlink throughput.

Figure 19: Throughput varying the fragment degree.

V100 GPUs. We examine the performance when running MegaStation on different types of GPUs. Specifically, we compare 2 V100 GPUs in a single chassis with two A100 GPUs and report overall average and tail latency. As shown in Figure 17, we observe that the A100 GPUs outperform the V100 GPUs by 27.2%. This is due to higher FLOPS and a greater number of SMs in the A100 GPUs (108 SMs compared to 80 SMs in the V100, which is 25.9% higher).

Tackling Fragmented Parallelism. While developing MegaStation, we find that fragmented parallelism happens frequently. Accurate computing resource accounting and careful scheduling are essential to ensure minimal frame processing deadline violation. We then revisit how MegaStation addresses the fragmented parallelism, where resources are partitioned among multiple GPUs and we should trade off communication overhead with executor distribution.

Similar to the characterization setup (§3.3), we evaluate the performance of these four approaches as the fragment degree increases under 128×64 MIMO with half uplink and half downlink symbols in one frame. As previously discussed, symbol-level execution granularity (*Hydra-GPU*) outperforms frame-level (*LuMaMi-GPU*) at low fragment degrees because data copying is hidden by computation. However, at high fragment degrees, frame-level execution performs better than symbol-level, as data copying cannot be overlapped with computation. Taking the data shuffling overhead into consideration in structural analysis, MegaStation leverages the benefits of both execution granularities, as illustrated in Figure 19. MegaStation employs a top-down approach that first selects a GPU for one frame and then assigns its symbols to that GPU. If the GPU reaches its capacity, MegaStation trade-offs between finding a new GPU (i.e., introducing communication overhead) or waiting for the current one to become capable. By dynamically adapting the execution granularity based on the fragment degree, MegaStation is able to achieve high performance across all fragment degrees.

Efficient Multi-WAN Transport for 5G with OTTER

Mary Hogan[‡], Gerry Wan[•], Yiming Qiu[†], Sharad Agarwal[*], Ryan Beckett[*], Rachee Singh[°], Paramvir Bahl[*]
[‡]Oberlin College, [•]Google, [†]University of Michigan, [*]Microsoft, [°]Cornell University

Abstract

In the ongoing *cloudification of 5G*, software network functions (NFs) are replacing fixed-function network hardware, allowing 5G network operators to leverage the benefits of cloud computing. The migration of NFs and their management to the cloud causes 5G traffic to traverse an operator's wide-area network (WAN) to the cloud WAN that hosts the datacenters (DCs) running 5G NFs and applications. However, achieving end-to-end performance for 5G traffic across two WANs is hard. Placing 5G flows across two WANs with different performance and reliability characteristics, edge and DC resource constraints, and interference from other flows is different and more challenging than single-WAN traffic engineering. We address this challenge and show that orchestrating paths across a multi-WAN overlay[1] allows us to achieve average 13% more throughput, 15% less RTT, 45% less jitter, or reduce average loss from 0.06% to under 0.001%. We implement our multi-WAN 5G flow placement in a scalable optimization prototype that allocates 26%–45% more bytes on the network than greedy baselines while also satisfying the service demands of more flows.

1 Introduction

The majority of 5G access networks today use legacy telecom equipment and architecture. The next generation of 5G is designed to deliver superior Quality-of-Service (QoS) through a combination of new frequency bands, new radio technology, and the *cloudification* of telecom network functions (NFs). These innovations are expected to revolutionize 5G and subsequent generations of access networks. In this work we focus on the implications of improved QoS and cloudification of access on wide-area networking (WAN).

As part of the cloudification of 5G, monolithic and proprietary hardware implementations of mobile wireless NFs are evolving into disaggregated software-based NFs that run on commodity off-the-shelf compute [79]. This disaggregation allows 5G operators to deploy their virtualized radio access networks (vRAN) on edge compute close to end users, and packet cores in datacenters (DCs) [56, 61].

Cloudification is driving performance-sensitive inter-domain traffic onto cloud WANs. For instance, traffic from user equipment (UEs) traverses a 5G operator's WAN to a cloud WAN to reach 5G Core NFs and applications hosted in the cloud [56, 61]. The burden of providing QoS to this traffic is now shared between two WANs: the 5G operator's WAN and the cloud WAN. However, existing traffic management mechanisms lack the ability to orchestrate paths shared between two WANs. In fact, 5G networks are already bottlenecked on WAN performance [91], and this will get worse as the upcoming 5G New Radio (5G NR) unlocks ultra-low latency and high-bandwidth radio modes.

Improving the performance of inter-domain 5G traffic is challenging for three main reasons. First, cloudified access traffic is destined to cloud DCs that host 5G NFs and applications. The location of these 5G services in the cloud is itself dynamic [33, 80] and conditional on the availability of compute resources, making it hard to achieve performance-optimal routing. Second, the primary mechanism to achieve network performance goals in WANs today is *intra-domain traffic engineering* (TE) [39, 41, 44] which does not capture inter-domain route performance goals or fine-grained service objectives of 5G traffic. Third, the dynamic nature of 5G traffic requires on-demand flow placement, which is not achievable by traditional TE mechanisms that operate on periodic (*e.g.,* 5 minute) allocation intervals.

We develop OTTER (Overlay Traffic Transport and Efficient Resource allocation), a system that provides efficient, on-demand transport across operator and cloud WANs for 5G traffic with differing QoS needs. It uses a multi-WAN overlay to co-optimize the placement of 5G NFs and applications, and the network paths of 5G traffic. OTTER dynamically computes optimal routes for multiple performance metrics and implements them using scalable forwarding mechanisms involving VMs and cloud routing gateways. OTTER is cloud-agnostic, allowing operators to deploy it on cloud providers of their choice.

Our contributions. OTTER makes two key contributions. First, it develops an efficient algorithm for allocating network and compute resources to 5G demands across operator and cloud WANs. This algorithm forms the core of OTTER's multi-WAN SDN (Software Defined Networking) controller. Second, OTTER implements the optimal routing and compute placement strategy of the controller using a novel multi-WAN forwarding mechanism. We develop the multi-WAN forwarding mechanism as part of the OTTER Orchestrator, which programmatically creates multi-WAN overlay networks with fine-grained routing policies for operators to deploy their 5G NFs.

[1] OTTER topology release: https://OTTER-5GWAN.github.io/

We design OTTER with the observation that the operator and cloud have newly aligned economic incentives for 5G, as their business relationship goes beyond peering to also include SaaS and compute. The OTTER Orchestrator scales to country-wide sizes of large operators by leveraging cloud-native support. The OTTER Optimizer dynamically places both compute and network workloads on multi-WAN overlays, for multiple traffic classes simultaneously.

Results. OTTER's path orchestration across two commercial WANs outperforms both public paths as well as topologies designed for enterprise-grade privacy and security. Our deployment topology scaled out to the continental US shows 13% higher throughput on average, with a best case of 136% higher, which can be 6-10 Gbps more. For flows that care about round trip time (RTT), we achieve 15% average reduction, up to 42 ms less. Jitter-sensitive flows can expect an average reduction of 45% and loss is 0.06% less on average. OTTER's on-demand allocation outperforms greedy baselines by up to 26-45% in allocated bytes (the amount of demand that is satisfied), and comes close to what an infeasible, infinitely-fast optimizer can achieve.

2 Cloudification of Access

Traditionally, vendors have implemented cellular technology on monolithic and proprietary hardware. However, 5G cellular networks are being re-architected using SDN and NF virtualization (NFV) [9]. This new architecture enables operators to scale up and out NFs on statistically multiplexed compute at operator edge sites (at or near cellular base stations and mobile switching centers), cloud edge sites (at hosting centers or peering points), and cloud DCs [55]. The use of the same compute management plane across all three allows NFs and interactive applications (*e.g.,* AR/VR) to be deployed flexibly, as user demand dictates. This allows operators to unlock the elasticity and efficiency of cloud computing. This convergence of access networks and hyperscalers is known as the *cloudification* of 5G [27, 64, 65].

2.1 Implications for WANs

Performance-sensitive demands over multiple WANs. The deployment of 5G applications and NFs on the cloud puts the cloud WAN on the critical path of 5G traffic that originates in the operator WAN, as shown in Figure 1. Extremely performance-sensitive NFs, such as RAN (Radio Access Network) NFs are hosted in operator far edges [37, 64], close to cell sites. Other NFs, such as the 5G Core, are hosted in the cloud, in its edges and/or DCs. For example, Nokia's 5G Core runs on the AWS cloud [61], Microsoft's 5G Core runs on both on-prem servers and Azure DCs [56], and its voicemail NF is hosted on Azure DCs [52]. Multiple operators have already moved parts of their 5G infrastructure

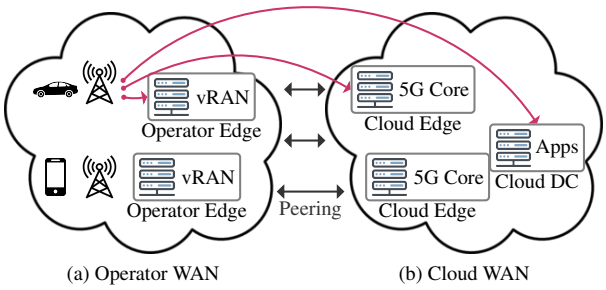


Figure 1: A simplified example of a 5G network

to the public cloud. AT&T is using Azure [8, 60]. Verizon is using AWS [12], leveraging multiple cloud technologies including Amazon EKS, AWS Wavelength, and HashiCorp Consul. The operator and cloud WANs peer [69] directly in multiple locations. An edge or DC can serve traffic flows from operator cell sites depending on where the destination NF or application is hosted [55]. Applications may include operator applications such as phone storage backup, private enterprise applications such as industrial AR/VR, and consumer applications such as game streaming.

Heterogenous network demands. 5G NFs have different performance requirements. For example, some NFs in the vRAN have stringent RTT, loss, and jitter requirements, while others such as UPF (User Plane Function) in the 5G Core need high throughput, and voicemail is performance-insensitive. Furthermore, upcoming releases such as 5G NR enable new applications, each with diverse performance requirements. For example, interactive AR/VR requires sub-20 ms network RTT for acceptable user experience [79, 91], beyond 4K video streaming needs 100+ Mbps throughput, and remote surgery demands high reliability. The 5G specification supports these diverse requirements through multiple radio-layer modes [65, 67, 79, 84], including: (1) *Ultra-reliable low latency communication (URLLC)* with ∼1 ms latency and 99.999% reliability, (2) *Enhanced mobile broadband (eMBB)* with multi-Gbps data rates, and (3) *Massive machine-type communication (mMTC)* for ultra-low energy communication at scales of up to 1 million nodes per KM^2.

Compute selection and routing decisions jointly matter. Placement of applications and NFs on compute that is *near* vs. *far* from the user [73], and the choice of network routes between users and this compute both impact performance. While operator edges are closer to users (few ms away) and hence are an ideal target in which to place highly responsive workloads (*e.g.,* vRAN), they have very limited compute due to physical space and power constraints at cellular base stations and mobile switching centers [74]. On the other extreme, the DC is furthest away (tens to hundred ms away) but has far more available compute.

Traditional selection techniques, such as those in CDNs (Content Delivery Networks), fall significantly short here. Directing all traffic from an entire subnet or LDNS (local

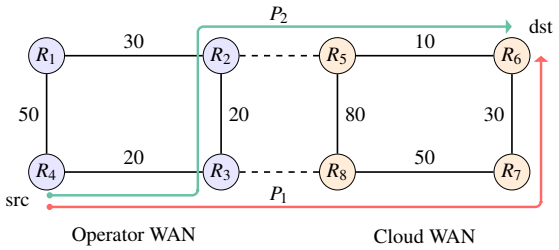


Figure 2: Sub-optimal routing for independent WANs.

DNS) coverage area to a single destination leads to undifferentiated performance for multiple flows with different requirements. Similar to DNS, the 5G NRF [33,80] (NF Repository Function) maintains profiles of available instances of NFs, and provides discovery and load balancing. However, it is not network-aware and only considers compute load. The problem is further exacerbated by the use of SFC (Service Function Chaining) [20] in 5G, where instances of different NFs are dynamically chained together to serve a request.

Picking a destination for a network demand based on compute capacity alone can lead to poor E2E performance. Picking a destination for a network demand based on network performance alone can lead to overloaded compute. Performance studies of existing 5G networks [91] have already found that "the untamed latency in the wireline paths, which is beyond mobile carriers' control, may neutralize 5G's latency advantage" and argue that "the wireline paths, upper-layer protocols, computing and radio hardware architecture need to co-evolve with 5G."

Need for multi-WAN coordination. Cloudification forces operators and clouds to share responsibility for tight performance guarantees of 5G workloads. Individual routing and compute placement decisions leads to sub-optimal performance. Figure 2 shows a latency sensitive flow traversing two WANs from source R_4 to destination R_6. Each link has an associated latency. The operator chooses the best egress point-of-presence (PoP) R_3 and path R_4R_3 to R_3 to minimize RTT on its network. The cloud routes traffic from R_8 to R_6 following the optimal RTT path, resulting in the E2E path P_1. However, a more efficient path P_2 exists. We show in §7 that decision making across both the cloud and operator unlocks such performance improvements.

Given the multitude of 5G networks across the world, multiple cloud providers, and dynamic nature of traffic and operational WANs, manual coordination to address performance problems as they appear does not scale.

New incentives for multi-WAN co-operation. Historically, it has been challenging to achieve collaborative routing beyond traditional peering agreements on the Internet. However, the cloudification of access has aligned the economic incentives of cloud and operator networks. The operator is not only selling access to end users to the cloud, it is now also a customer of the cloud for hosting 5G applications and NFs. There is a shared responsibility to manage compute and network between the two WANs for 5G demands.

3 OTTER

Prior empirical work [91] has shown that as 5G performance over-the-air has improved, WANs have become the bottleneck. We tackle this problem by placing heterogenous performance demands on the most appropriate compute and network paths across operator and cloud WANs. We refer to this as the multi-WAN flow placement problem, and it requires two core capabilities — the ability to direct a flow to a specific compute endpoint over a specific network path, and the ability to calculate that optimal path and endpoint.

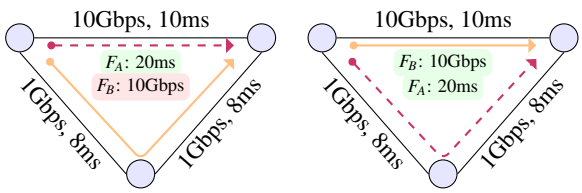
3.1 Design Goals

We design OTTER with the following goals:

Co-exist with non-5G WAN traffic. 5G traffic will share operational WANs that serve existing customers. For example, operator WANs already provide wireline connectivity to consumers or *eyeballs*. Cloud WANs carry enterprise and consumer workloads. Moreover, both types of WANs operate existing TE systems to engineer the capacity of their network efficiently [11,25,26,39,41,89]. However, both operator and cloud TE systems aim to achieve network welfare goals such as high utilization or bandwidth fairness [39,41], rather than optimizing for individual service objectives. Instead of overhauling existing TE systems in cloud and operator WANs, we design OTTER to be an overlay that operates in a layer of abstraction above existing WAN TE.

Dynamic flow placement. This overlay design choice further allows OTTER to place 5G flows on network paths and compute endpoints on-demand. In contrast, existing TE and other optimizations operate on batched traffic demands by periodically (*e.g.,* every 5 minutes) computing flow allocations from a predicted traffic matrix [39,41], or running a single optimizer instance for a single input demand [40]. A broker [39] enforces those periodic allocations by squelching flows that exceed them. While existing TE will continue to run on each WAN separately, OTTER can dynamically place flows without being subject to periodic allocations nor requiring bandwidth brokers in 5G deployments.

Support multiple, fine-grained service objectives. Today's WAN TE systems provide service differentiation using only a small number of priority classes (*e.g.,* high priority vs. low priority) [10, 39]. Such few classes are insufficient to express the more complex and fine-grained objectives of 5G traffic. These coarse-grained priority classes can fail to capture the demands of lower priority flows, even when there exists an allocation that maps all flows to acceptable paths. Consider a simple example in Figure 3, where each link can carry at most one flow. Flow F_A has high priority with a latency requirement of 20 ms, and flow F_B has low priority



(a) Preferential (b) Non-preferential

Figure 3: Two flow allocation strategies

with a throughput requirement of 10 Gbps. On the left, we show the allocation where flows are assigned in strict priority order, preferring shorter paths—this is how modern TE systems like OneWAN [45] and SWAN [39] operate. F_A will be assigned first to the top path while F_B will be assigned to the remaining bottom path and fall short of its throughput requirement. On the right, we show an alternate allocation that can map both flows to paths that meet their service demands. However, it is not trivial to find such an allocation at the scale of hundreds of endpoints and thousands of flows, each with their own performance demands across a variety of metrics.

Take inter-domain routing protocols as a given. Inter-WAN routing leads to sub-optimal E2E routes due to limited information disclosure and different goals between WANs [47, 48]. While prior work has shown that ASes can exchange information to produce E2E optimal routes, such negotiation entails heavyweight communication overheads that may not be practical [47, 48]. In the context of 5G, this overhead is further exacerbated by numerous traffic flows, each with widely varying objectives. So, we design OTTER without requiring changes to inter-domain routing protocols, policies, and implementations.

Not require private WAN data. While sharing intra-WAN topologies could lead to better multi-WAN paths, ASes have economic incentives to limit such visibility [47, 48, 75, 81]. 5G cloudification presents a unique opportunity for efficient cross-domain routing without sharing such private data. The operator, as a customer of the cloud, owns the workloads deployed on VMs in cloud edges, DCs, and operator edges. The cloud provides the control plane for managing bare metal at these locations, such as provisioning VMs and establishing connectivity. Thus both the operator and the cloud have visibility into network performance at an overlay layer between all compute locations. OTTER uses this visibility to make informed flow and compute placement. We show that the operator can deploy and run OTTER across both WANs, without access to private data or capabilities, and still provide significant performance improvements.

Multi-cloud support with open interfaces. 5G operators want to avoid cloud-lock in and deploy their NFs to one or more cloud providers of their choice. To achieve this, we design OTTER as a multi-cloud network overlay across operator and cloud WANs. OTTER uses publicly available VMs as nodes in the overlay, and a variety of public cloud NFs for

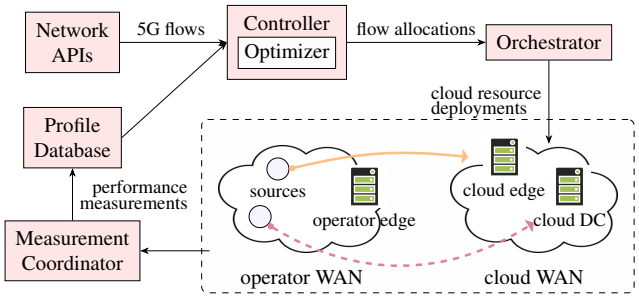


Figure 4: Overview of OTTER

routing between the nodes. This allows operators to use OTTER in multi-cloud deployments, without requiring private or special privileges from an underlying cloud.

3.2 OTTER Overview

OTTER places flows in a multi-WAN environment to satisfy their service demands. It does so by co-optimizing which 5G application / NF compute will serve the flow, and the route that flow will take to it. Figure 4 shows OTTER's architecture consisting of the Controller and Orchestrator. The Controller accepts flow demands to 5G NFs and applications using APIs [28, 34, 53, 57]. These "Quality on Demand" APIs [3] allow an application to create a session by issuing an HTTP POST to the */sessions* API, providing flow identifiers (IP addresses, port ranges, device phone number, etc.) and a QoS profile that best matches the application needs (see the "quality-on-demand v0.11.1" API on Swaggger [3]). Such standardized APIs have been implemented by 5G operators (see the implementations by Deutsche Telekom, Orange, and Spry Fox Networks posted on GitHub [3]) and by clouds that offer to host 5G services [15]. In addition to allowing application developers to specify their 5G flow needs, these APIs allow for termination of a session, extending a session, and getting notified when the requested QoS cannot be fulfilled. A UE that disconnects and reconnects, or incurs an inter-RAT (Radio Access Technology) handover, can create a new session with this API.

The Controller allocates compute to meet these performance requirements within the limits of compute availability on edges and DCs. The Controller calculates the most appropriate network paths to the allocated compute. Unlike TE systems, OTTER relies on available compute capacity at destination sites, as well as periodic measurements of network paths that it makes. This not only allows OTTER to route flows along alternate paths, but also to alternate destinations. The Orchestrator manages cloud resources and implements the forwarding mechanism to steer traffic along routes computed by the Controller. It provisions compute as per the Controller's optimal placement.

While similar challenges can be faced by other applications across WANs, OTTER is designed to serve 5G deployments. The business alignment between "eyeball" networks and cloud networks, where 5G deployments are hosted

across both, drives the necessary incentive to orchestrate both WANs. The diversity of 5G applications requires a design that supports multiple classes of traffic, not just bulk traffic [40]. End-user flows have to be placed in real time, not at traditional, periodic TE intervals. OTTER leverages APIs [28,34,53,57] already deployed by 5G providers to accept these application demands, where such APIs have not been deployed at scale for other applications nor networks. Finally, we design OTTER as an overlay without forcing one WAN to have administrative control nor require use of private capabilities – this design hence does not preclude a cloud from supporting multiple 5G operators, and a 5G operator from relying on multiple clouds.

4 OTTER Controller

The OTTER Controller maps traffic demands to network paths and compute resources. It does so while maximizing the allocated traffic and minimizing the degree to which 5G application-specific service requirements are violated. The resulting mapping of flows to destination compute sites and the network paths to reach them is an input to the OTTER Orchestrator. It informs the Orchestrator at which of the available edge sites or DCs to terminate 5G flows and which network paths to use. In this section, we formalize this problem, referred to as the *5G multi-WAN flow placement* problem, explain the constraints on the Optimizer, and describe techniques used to ensure its scalability.

The Optimizer takes as input the set of paths P that are available in the multi-WAN network. Each path $p \in P$ is represented by the tuple $(s,d,L,\sigma)_p$, where s_p is the source, d_p is the destination, L_p is the set of links used by the path, and σ_p is a vector of performance metrics for path p. σ_p^m refers to the value of metric m for path p. For example, the RTT, jitter, and packet loss of path p are represented by $\sigma_p^{rtt}, \sigma_p^{jit}, \sigma_p^{loss}$. L is the set of links in the network, and each link $l \in L$ has capacity c_l. Let P_l be the set of paths that share a link l, and P_d be the set of paths that share a destination d.

The Optimizer takes as input a set of 5G flows F that need to be allocated. Each flow $f \in F$ is represented by the tuple $(s,D,\mathcal{D},bw,r)_f$. s_f is the source of the demand. D_f is the set of destinations (specific edges & DCs) that can serve the flow. $\mathcal{D}_f$ represents the service demands of the flow with $\mathcal{D}_f^m$ giving the demand for metric m. bw_f is the requested bandwidth. r_f is a vector of destination resource requirements of the flow – compute, memory, and storage. We write r_f^π to refer to the requirements of flow f with respect to resource type π. Unlike TE, an OTTER flow is not limited to a single destination. Rather, we leverage the agility of the cloud to place compute resources for serving a 5G application or NF across multiple destinations. This allows OTTER to further optimize path performance by adjusting the destinations of flows. Compared to resource-constrained edge sites, a destination d that is a DC has $\approx \infty$ capacity c_d^π. For scalability, individual flows with the same source, destination set, service demands, and resource requirements are treated as part of a single "flow aggregate" for the Optimizer to allocate. The Optimizer solves the flow placement problem such that all flows within the same flow aggregate are routed in the same way and to the same destination.

Modeling service demands. We encode the service demand of a flow that is obtained from operator APIs (§ 3.2) as a vector of *demand functions* $\mathcal{D}_f$. A demand function $\mathcal{D}_f^m$ maps a flow's performance expectation, *i.e.,* what the flow expects to achieve for a given allocation, to a *tolerance coefficient* $\mathcal{D}_f^m(\sigma_p^m) \in (0,1]$. The tolerance coefficient is a measure of how good the path is for a flow. For example, an application may consider a path RTT under 100 ms acceptable, over 200 ms unacceptable, and linearly decrease its preference for a path between 100 and 200 ms. In this case, the Optimizer computes the tolerance coefficient to be 1 for all paths with an RTT under 100 ms, ε for all paths greater than 200 ms, and a value inversely proportional to the RTT for paths between 100 and 200 ms:

$$\mathcal{D}_f^{rtt}(\sigma_p^{rtt}) = \left\{ \begin{array}{ll} 1, & \text{if } \sigma_p^{rtt} \leq 100 \\ -\frac{1}{100}\sigma_p^{rtt} + 2, & \text{if } 100 < \sigma_p^{rtt} \leq 200 \\ \varepsilon, & \text{if } \sigma_p^{rtt} > 200 \end{array} \right\}$$

The Optimizer's goal is to allocate demands such that the degree of satisfaction (tolerance coefficients) is maximized. All tolerance coefficients are constant, allowing the objective function to be linear and the optimization problem to be a linear program (LP). The use of demand functions allows the Optimizer to scale over numerous path metrics and performance demands, as well as future-proof it for new applications that have arbitrary service demands.

Modeling resource requirements. Each flow specifies a resource requirement vector r_f. Each element represents compute in vCPUs, RAM, and disk storage needed to serve the 5G workload. While we focus on those three resources, the vector can be expanded and compared component-wise to the resource capacities of destination sites.

Decision variables. The Optimizer assigns network flows to paths in P and compute resources to destination sites in D. We have two decision variables: $x_{f,p}$, the amount of bandwidth of flow f assigned to path p, and $y_{f,d}^\pi$, the amount of resources of type π for flow f allocated at destination d. OTTER co-optimizes the flow allocation and the destination placement. See Table 1 for a summary of variables.

Objective function. The Optimizer aims to assign network flows to multi-WAN paths such that as much traffic is allocated as possible while still satisfying performance demands. The cost of a single flow assignment is the product of its allocated bytes and the overall tolerance coefficient summed over all performance metrics. To prevent bias towards large flows, we normalize this cost by the requested bandwidth of the flow bw_f. The objective function is as follows:

Inputs	
P	paths in network
s_p, d_p	source and destination of path p
L_p	links used by path p
σ_p^m	performance metric m for path p
L	links in network
c_l	capacity of link l
F	flows to allocate
s_f, D_f	source and destination set of flow f
$\mathcal{D}_f^m$	path performance demands for metric m of flow f
bw_f	requested bandwidth of flow f
r_f^π	resource requirements of flow f for resource π
c_d^π	resource capacities of destination d for resource π
Outputs	
$x_{f,p}$	amount of flow f assigned to path p
$y_{f,d}^\pi$	amount of resource π for flow f assigned to destination d

Table 1: Symbol table for the OTTER optimizer.

maximize	$\sum_{f\in F}\sum_{p\in P}\left(\frac{1}{bw_f}x_{f,p}\cdot\sum_m \mathcal{D}_f^m(\sigma_p^m)\right)$		
subject to			
	$x_{f,p} \geq 0$	$\forall f,p$	(c1)
$\sum_{p\in P}$	$x_{f,p} \leq bw_f$	$\forall f$	(c2)
$\sum_{f\in F}\sum_{p\in P_l}$	$x_{f,p} \leq c_l$	$\forall l$	(c3)
$\sum_{f\in F}$	$y_{f,d}^\pi \leq c_d^\pi$	$\forall d,\pi$	(c4)
$\sum_{d\in D_f}$	$y_{f,d}^\pi = r_f^\pi * \left(\sum_{p\in P_d} x_{f,p}\right)/bw_f$	$\forall f,\pi$	(c5)

Table 2: OTTER optimizer formulation.

$$\text{argmax}\sum_{f,p}\left(x_{f,p}\cdot\sum_{m\in\{rtt,jit,loss,bw\}}\mathcal{D}_f^m(\sigma_p^m)\right)/bw_f$$

We note that our objective function does not inherently prioritize one metric over another. Rather, applications can indicate priority through the demand functions.

Constraints. Table 2 summarizes the constraints in OTTER's optimization. Constraint $c1$ ensures that the total allocated flow is non-negative. $c2$ and $c3$ ensure that the allocated flows are bounded by their requested bandwidth as well as the bottleneck capacity of the paths. Destination sites have resource constraints that limit how many flows can map to them, and $c4$ prevents overload. Constraint $c5$ requires that the proportion of resources assigned to a flow at a destination is equal to the proportion of the amount of bandwidth allocated to the flow at the same destination.

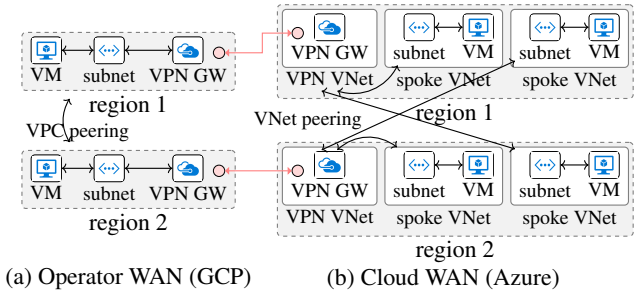
On-demand 5G flow allocation. Unlike TE systems which allocate flows periodically (*e.g.,* every 5 minutes) for predicted traffic demands, the OTTER Controller must perform on-demand flow placement. This difference arises due to the dynamic and unpredictable nature of 5G traffic [24] compared to typical WAN workloads. Unfortunately, re-running the Optimizer too frequently incurs a large delay. While the Optimizer computes newly optimal flow allocations, stale routes remain in-use, potentially leading to service demand violations [93]. To handle this, the Controller implements a greedy heuristic that approximates the optimal (*i.e.,* invoking the Optimizer on every new flow) configuration. It greedily allocates flows to available paths and destinations to provide the best demand satisfaction, while adhering to resource constraints. The Controller then periodically runs the full Optimizer in the background to optimally place new flows and those that were greedily assigned since the previous call to the Optimizer. The Controller can then shift those flows to achieve a globally optimal allocation, or it can pin flows to specific paths or destinations to avoid packet re-ordering. While shifting flows *could* result in an individual flow experiencing lower service demand satisfaction compared to its initial greedy allocation, reallocation still improves overall satisfaction across more flows, as shown in §7.

Computational complexity of OTTER. OTTER's optimizer is a linear program. Off-the-shelf optimization solvers implement polynomial time algorithms to solve linear programs, making it possible to solve OTTER's optimization quickly. While linear programs are quick to solve in practice, increasing the size of the problem can increase the run time of the solver. OTTER has two ways of dealing with high run times. First, OTTER reduces the set of possible destinations for each flow to a set of size n prior to solving the optimization. We estimate the n "best" destination sites using a flow's service demands ($\mathcal{D}_f^m$) to rank candidate destinations by preference. This effectively reduces the number of variables in the linear program. Second, depending on the size of the linear program, OTTER can run the full optimization at longer periodic intervals (*e.g.,* ten minutes) since the greedy flow allocation can assign flows to paths without waiting for the next interval. We evaluate the performance of greedy flow allocation between consecutive runs of the Optimizer, as well as the Optimizer, in detail in §7.

5 OTTER Orchestrator

The Orchestrator includes the forwarding mechanism necessary to steer traffic along routes computed by the Controller (§4). It also manages compute as per the Controller's optimal placement. We design the Orchestrator as a multi-WAN overlay on a cloud WAN and an operator WAN. To evaluate OTTER without impacting an operational 5G network, we substitute the operator WAN with GCP (Google Cloud Platform). Figure 5 shows one small deployment of OTTER, and Figure 19 in Appendix A shows a larger one. The Orchestrator uses only existing, native cloud functionality, which satisfies our design goal of not requiring private WAN data (§3).

Connectivity in the OTTER Orchestrator overlay. We host multiple VMs in each Azure region to represent 5G NF and application servers. VMs in GCP serve as 5G cell sites with far edges, while virtual private network (VPN) gateways behave like 5G near edge sites. We create private subnets in each GCP region and virtual private cloud (VPC) peer-



(a) Operator WAN (GCP) (b) Cloud WAN (Azure)

Figure 5: OTTER Orchestrator's multi-WAN topology.

ing connections in full-mesh between GCP regions. We use virtual network (VNet) peering between clusters of VMs in different regions to connect them to each other and to VPN gateways. Given that much of the 5G infrastructure, including cell sites and NFs, use private IP address ranges to protect communication, VPN tunnels are necessary to establish private connectivity among them. We have configured the VPN gateways to allow transit with the 5G operator WAN where both clouds have regional DCs and underlying peering (see Appendix A).

Forwarding in the OTTER Orchestrator overlay. The OTTER Orchestrator leverages native functionality in each cloud to achieve high forwarding performance, ease of scale, and lower deployment complexity. To control the path that a 5G flow takes in the overlay, we use a combination of route tables with user-defined routes (UDRs) and choice of multiple VNets in the destination region, something that both GCP and Azure support without requiring the customer to deploy their own BGP speakers and packet forwarders. Other design choices, such as using SNAT and DNAT with VPN gateways and firewalls [51], or MP-TCP on source and destination VMs, incur different tradeoffs in flexibility and overheads. We configure advanced parameters on VNet peering and network firewalls to allow VMs in one region to use a VPN gateway in another and allow transit traffic between regions. Each cloud component supports scale-up and scale-out, allowing the entire topology to easily cover the continental US, for example, without any code changes to individual components. We use symmetric paths for a flow but can easily use different paths in each direction. The use of one system to identify, manage, and use routes across WANs allows us to satisfy the need for multi-WAN coordination (§ 2).

Secure multi-WAN routing. We use private VNets to host 5G cell sites, edge sites, and NFs. We use VNet peering to allow only authorized traffic to transit between VNets. We use VPN gateways to enforce encryption when transiting WANs. Our design choices were inspired by product demonstrations from industry 5G NF providers [61].

Extensible multi-WAN topology. This design allows both scale and flexibility. Given an arbitrary source in a GCP region and a destination in an Azure region, it can use any VPN connection and any intermediate Azure and/or GCP region to send and receive traffic. We achieve this through a combination of VPN connections, VNet peering, and selective route announcements. We trivially increase inter-WAN forwarding throughput by scaling out VPN gateways and tunnels.

6 System Implementation

We have implemented the entire OTTER system shown in Figure 4 to operate on top of Azure and GCP, with no special support from either underlying cloud. The Orchestrator is built with HashiCorp's Terraform [86]. Our evaluation topology (available in an anonymous repository [1]), listed in Table 3 and described in §7.1, is built in ∼4,100 LoC of HCL (HashiCorp Configuration Language), with ∼2,600 LoC for module deployments, ∼600 LoC for GCP resource declarations, and ∼900 LoC for Azure resource declarations.

We implemented the Measurement Coordinator in ∼1,200 LoC in C#. It uses the Azure and GCP SDKs to automate periodic execution of measurement tools for path bandwidth, RTT, jitter, and loss. We use iPerf3 [23] to measure throughput, configured to use TCP CUBIC with 60 parallel connections, 2 seconds of warm up, and 5 seconds of measurement. For packet loss, we use iPerf3 in UDP mode. For RTT and jitter, we use sockperf [49]. We use round-trip values, and jitter is calculated as the standard deviation of RTT.

The Measurement Coordinator analyzes the output of the Orchestrator's Terraform execution to determine the deployment topology and the names and IP addresses of its components, including source and destination VMs. It then creates a non-overlapping schedule of measurements. The output of the measurement tools is parsed and converted to JSON. That JSON is then inserted as a new document record into an Azure Cosmos DB NoSQL database [54].

The Optimizer issues simple SQL queries on that database to obtain sliding window values of median throughput, RTT, jitter, and loss for all paths. 5G flow requirements are modeled from a subset of application profiles in the 3GPP standard [79], which we describe in §7.2. However, we expect to integrate with network APIs [28, 34] such as Azure Programmable Connectivity [53, 57] that allow 5G applications to specify flow QoS requirements. We have implemented the Optimizer using the Gurobi [35] solver. The Optimizer is an LP of ∼500 LoC. We run our implementation on a VM with a 16 core 3GHz CPU and 48 GB of memory.

7 Evaluation

We evaluate OTTER in two ways. First, in §7.1, we demonstrate the value of orchestrating paths across two WANs, compared to the default path from traditional Internet routing. Second, in §7.2, we evaluate how OTTER's optimization algorithm performs under a wide variety of situations at scale, and in comparison to alternative algorithms.

Cloud	Element	Deployment strategy	Total
GCP	regions	cover US	8
	VPC	1 per region	8
	subnet	1 per region	8
	cloud router	1 per region	8
	VPC peering	full mesh	56
	32 core VM [31]	1 per VPC	8
	classic VPN GW	4 per VPC	32
	cloud VPN tunnel	2 per VPN GW	64
Azure	regions	cover US	8
	VpnGw5 GW [14]	1 per region	8
	VPN GW interface	8 per VPN GW	64
	32 core VM [13]	8 per region	64
	VNet	1 per VM and VPN GW	72
	VNet peering	each VPN GW VNet to VM VNet	64

Table 3: Scale of OTTER topology across US regions.

7.1 OTTER Orchestrator Evaluation

§5 briefly describes how OTTER's Orchestrator works, with a secure, private, scalable design that allows flexible path selection to cater to multiple QoS requirements. We evaluate this orchestration at scale on two commercial cloud WANs.

We have scaled out the topology shown in Figure 5 to cover the US and support a large number of flows. Table 3 lists how many of the main components we use, though there are many other sub-components that we do not list, such as VM drives, NICs, custom routes, firewalls, and security rules. We use the biggest VPN GWs and multiple connections. We create VPN connections between the two clouds in nearby regions, listed in Appendix A. This topology is sized similarly to that of large 5G operators [21]. We use the largest VMs that have sufficient CPU and network bandwidth such that the VMs themselves are not bottlenecks in any of our measurements. We validate this through intra-cloud testing, where for example we achieve >25Gbps throughput between any two VMs in GCP in our topology, more than what is possible between two VMs across the two clouds.

We have run experiments on different weekdays and observed qualitatively similar results. For brevity, we show results from a 24 hour period starting on 09 Feb 2023 13:00PM (PT), where we repeatedly measure $8^3 = 512$ paths over 20 times. When comparing any two paths between the same source on GCP and the same destination on Azure, we only consider measurements immediately adjacent in time to each other. This ensures an "apples to apples" comparison, since comparing a path at 2am with one at 2pm is not useful for real-time demands.

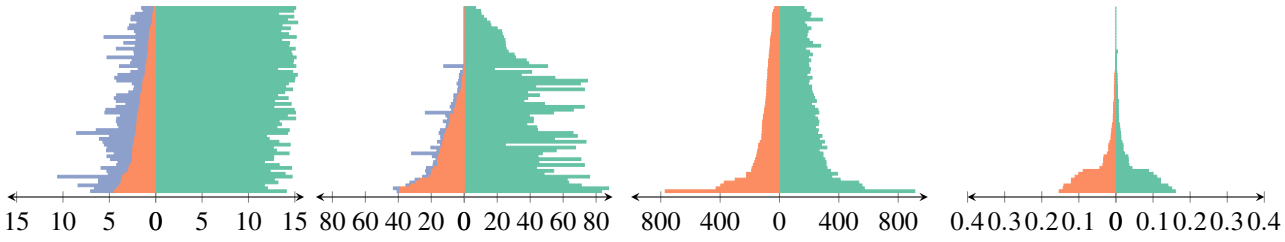
Figures 6a, 6b, 6c, and 6d show the network performance improvements that the OTTER Orchestrator is able to achieve. Each horizontal bar corresponds to one of 64 unique source-destination (Src-Dst) pairs, with the source in GCP and destination in Azure. The bar on the right represents the median performance observed for a Src-Dst pair when the OTTER Orchestrator is not invoked, *i.e.,* the default path [32] that a flow experiences when we do not specify which VPN connection to use and do not force traffic through an intermediate region (see Figure 5). The first bar on the left shows the median improvement that the OTTER Orchestrator achieves for the same Src-Dst pair when considering the best performing path across all possible combinations of VPN connections and intermediate regions. This median is computed as follows: for each round of measurements, we calculate the performance improvement of OTTER orchestration over the default path, and then use the median value across the ~20 rounds that happen over 24 hours. The second bar on the left shows the maximum improvement observed. The Src-Dst pairs in each figure are sorted by median improvement.

For example, the bottom bar in Figure 6a shows that this Src-Dst pair can achieve 14.1 Gbps, in the median across the 24 hour period, but the OTTER Orchestrator can improve that by 4.4 Gbps to achieve 18.5 Gbps in the median, and in the best case improves throughput to 21.1 Gbps. The bottom bar in Figure 6b shows a Src-Dst pair with 83.2 ms median RTT, but the OTTER Orchestrator can reduce that by 39.4 ms to achieve 43.8 ms in the median, and in the best to 42.4 ms.

Consider one Src-Dst pair example in Figure 7. From a source in GCP's US West 3 region to a destination in Azure's South Central US region, 13.6 Gbps throughput and 40.6 ms RTT is the median performance, on the typical [32] hot potato path, where traffic is handed off to Azure at the nearest location in Azure's West Central US region. OTTER has two other paths between the same source and destination that the Orchestrator can use with different RTT and throughput tradeoffs. When sending traffic via GCP's US South 1 region, the median RTT is significantly better at 35.0 ms, but at a slight degradation in throughput (13.5 Gbps). Instead, via GCP's US West 4 and Azure's West US 3 region, there is significantly higher throughput of 19.4 Gbps, but at slightly worse RTT of 42.8 ms. There is not one path with the best throughput, RTT, jitter and loss—different paths have different performance. OTTER is able to exploit the spectrum of paths to send traffic along the appropriate path for a flow, using scalable mechanisms involving VPN tunnels, VNet peering, and user-defined routes (§5).

Appendix B has additional presentations of these experimental results. We summarize our findings as:

- For flows prioritizing throughput, the OTTER Orchestrator offers higher throughput than the underlying networks. On average, it is 13% higher and in the best case 136% higher. In some cases, the throughput increases by 6-10 Gbps. The peak throughput OTTER achieves between two VMs across the clouds is >20 Gbps.
- For RTT, the average reduction is 15%, with a maximum of 56%. In some cases this is a RTT reduction of 42 ms, within the continental US.
- The average jitter reduction is 45%, with a maximum of 99%, which in some cases is a reduction of over 10 ms.
- Excluding some outliers, packet loss is reduced from 0.06% on average to less than 0.001%. In some cases



(a) Throughput (Gbps) (b) RTT (ms) (c) Jitter (μs) (d) Loss (%)

Figure 6: Improvements that the OTTER Orchestrator achieves per Src-Dst pair for each of 4 performance metrics. Bars on right show performance of default path, bars on left show median improvement and max improvement. Max improvements for jitter and loss are excluded for clarity.

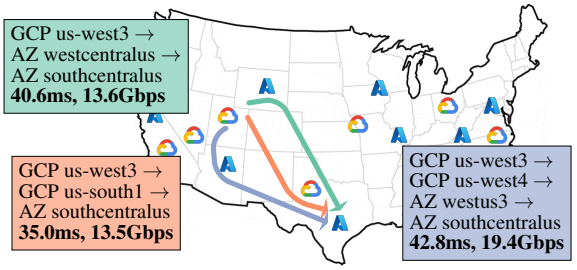


Figure 7: Examples of improvements in performance OTTER can achieve between the same Src-Dst pair.

the maximum reduction is more than 0.4%.

- Our scalable, secure, and private multi-WAN architecture achieves higher throughput than a simple architecture of VMs with public IPs. While some cloud components such as VPN tunnels can slightly worsen RTT and jitter, the OTTER Orchestrator compensates by finding better paths, providing the highest throughput, lowest RTT, jitter, or loss for any Src-Dst pair.

Prior work [40, 48, 91] has observed inefficiencies in the default intra- and inter-WAN paths provided by existing networks. Here, we comprehensively identify what OTTER can achieve across multiple objectives in the context of a deployment topology that is secure, private, and scalable, and achieves higher throughput than previously shown [40]. As seen in many TE algorithms [39, 41], the primary objective function is satisfying as much traffic demand as possible, with a possible secondary objective of minimizing RTT. This makes sense in a commercial WAN, where the cost of building and operating it has to be amortized over as much customer demand as possible. In contrast, OTTER's goal is to provide best performance to individual flows, and as such utilizes multiple paths that offer different QoS, without having to scale to the TE demands of the underlying networks. Further, OTTER utilizes virtual routing techniques (see §5), to react to real-time performance changes without requiring the underlying network infrastructure to issue route updates and converge, forcing all traffic to use those routes.

7.2 OTTER Controller Evaluation

Setup. We evaluate the Controller on the same continental US-wide, multi-WAN topology on which the Orchestrator is evaluated (§5). In Appendix F, we also evaluate synthetic smaller and larger topologies, including those that match the scale of contemporary TE systems, and show qualitatively similar results. To avoid limiting findings to specific network performance values, we evaluate the Controller on a distribution of values listed in Table 4. Path attributes are sampled at random from the distribution of measured path metrics in our multi-WAN topology, while destination and flow attributes are chosen uniformly at random from the specified range to mimic 5G traffic [79].

Flows arrive according to a Poisson process with a mean arrival rate. We evaluate multiple flow arrival rates as listed in Table 4. We use 10 seconds as the mean flow duration, and we show qualitatively similar results for varying durations in Appendix C. We randomly assign an application profile to each flow and generate a demand function for each performance metric using one of three function types: step, linear, or rational, with inflection points in each function derived from the guideline service demands in Table 5. Those demands are based on 5G application profiles from the relevant 3GPP standard [79], described in more detail in Appendix D. To account for flow diversity, we select inflection points at random from within $\pm$25% of the guideline metric. We also evaluate the effect of assigning application profiles from varying distributions based on RTT requirements, instead of random assignments, and show qualitatively similar results in Appendix E.

Heuristics. We implement several heuristics in the Controller, with different levels of optimality, compute needs, and impact on ongoing flows:

- GREEDY: Flows are allocated greedily in arrival order to the best paths based on the tolerance coefficient.
- PER: Same as GREEDY, but the Optimizer also periodically runs to shift all active flows to optimal paths. While the Optimizer is computing allocations, incoming flows are allocated greedily.

Topology			value/range		units
Num sources				8	#
Num destinations (DC)				2	#
Num destinations (edge)				6	#
Num paths per src/dst pair				8	#
Path Attributes	min	max	avg	std	units
Throughput	4.0	20.8	13.7	1.6	Gbps
RTT (DC)	69.2	150.8	92.2	18.2	ms
RTT (edge)	3.2	34.6	24.8	7.2	ms
Jitter	0.008	38.0	0.62	1.38	ms
Loss	0.0	4.3	0.06	0.16	%
Destination Attributes			value/range		units
CPU capacity (edge)			128-4096		cores
Memory capacity (edge)			128-4096		GB
Storage capacity (edge)			1024-32768		GB
Flow Attributes			value/range		units
Flow arrival rate			20K-40K		flows/s
Flow duration mean			10		s
Flow duration std			10		s
Application profile			1-8		index
Demand function			{step, linear, rational}		type
CPU requirement			1-128		#
Memory requirement			1-128		GB
Storage requirement			8-1024		GB

Table 4: Evaluation Parameters. Values are sampled at random from the specified ranges. Path attribute values are actual measurements from our multi-WAN topology.

Application name	**Throughput** (Mbps)	**Latency** (ms)	**Jitter** (ms)	**Loss** (rate)
Conversational Voice	0.3	100	-	10^{-2}
Live Streaming Video	1.5	150	-	10^{-3}
Buffered Streaming Video	15	300	-	10^{-6}
MC-PTT Voice	1	75	-	10^{-2}
Augmented Reality	20	10	-	10^{-6}
Real-time Gaming	10	50	10	10^{-3}
Remote Driving	1	5	-	10^{-4}
Intelligent Transport	1	30	-	10^{-5}

Table 5: Application profiles for flows (see Appendix D for details).

- PER+PATHPIN: Same as PER, except RTT and jitter-sensitive flows are pinned to their destinations and paths and not shifted even if the Optimizer finds a better path.
- PER+DSTPIN: Same as PER, except RTT and jitter-sensitive flows can be assigned to a new path without changing the destination.
- OPT+PATHPIN: An unattainable version of PER+PATHPIN that uses an infinitely fast optimizer. This removes the need to greedily allocate during optimizer compute time.
- OPT+DSTPIN: An unattainable version of PER+DSTPIN that uses an infinitely fast optimizer.
- OPT: An unattainable version of PER that uses an infinitely fast optimizer, which invokes a re-optimization of the network on every flow arrival and can re-allocate flows to optimal paths with no restrictions.

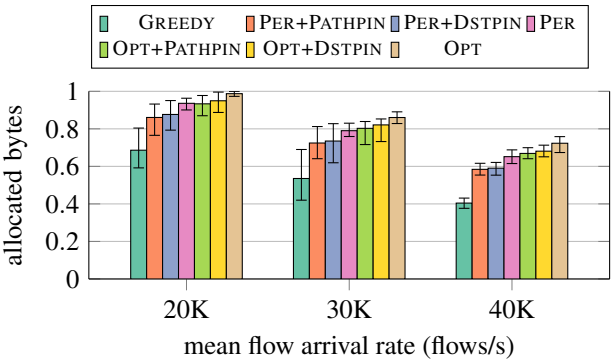


Figure 8: Average fraction of allocated bytes across all flows. Error bars show min-max over multiple runs.

Traffic allocation. Figure 8 shows the average fraction of bytes allocated across all flows for each allocation method for varying flow arrival rates. We define allocated bytes for a flow as the amount of requested bandwidth bw_f successfully allocated multiplied by its duration. We see that OTTER with periodic re-optimizing (PER) always allocates more traffic compared to the baseline GREEDY method, while still reacting quickly to on-demand flows. As expected, as the flow arrival rate increases, the number of allocated bytes decreases across all allocation methods. This is explained by the increase in resource contention when more active flows are present on networks of the same size with similar link and destination capacities. We also see that PER+PATHPIN and PER+DSTPIN on average allocate 26–45% more bytes than GREEDY, and just 10% fewer bytes than their unattainable optimal counterparts. In Appendix F, we scale to mean flow arrival rates of up to 512K flows/s without observing a large drop-off in gap-to-optimal performance.

For intuition into how the OTTER Controller allocates flows, Figure 9 has a time series of the average fraction of requested bandwidth that is successfully allocated by each method. It shows results for a mean flow arrival rate of 40K flows per second. Note that for instantaneous bandwidth allocation, we do not weight the allocation by the size of the flow, and only show the unweighted average across all flows at each point in time. OTTER's periodic re-optimizing methods exhibit a sawtooth behavior, with a jump in allocated bandwidth after each optimizer run and a steady decrease during the solve time due to falling back to GREEDY allocation for new flows. PER+PATHPIN and PER+DSTPIN provide a less optimal solution after each Optimizer run due to pinning, as shown by smaller sawtooths, but can run more frequently since the additional constraints reduce solve time.

These results also show how OTTER's Optimizer better maps flows to suitable paths without starving flows that may be considered "lower priority" in other schemes. While GREEDY also uses our notion of tolerance when ranking paths, it will greedily assign a flow to the best available path

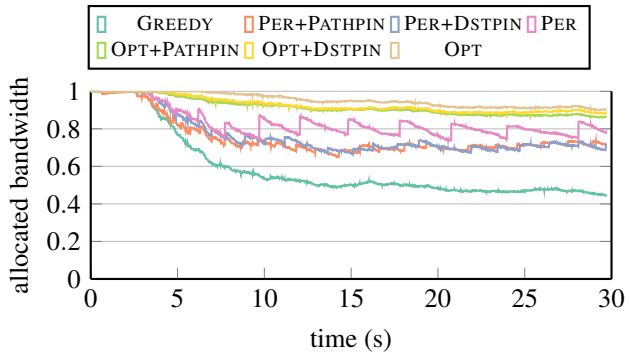


Figure 9: Average fraction of allocated bandwidth across all flows over time.

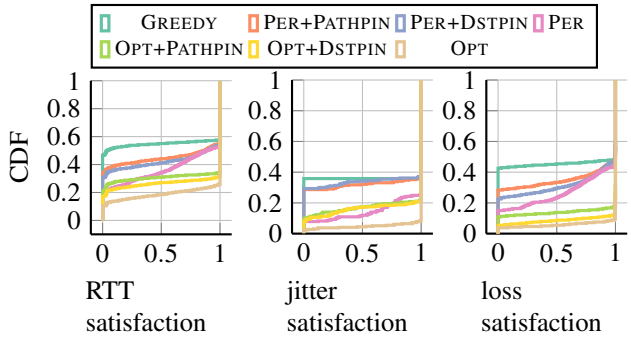


Figure 10: Service demand satisfaction for RTT, jitter, and packet loss. Value of 1 means perfect satisfaction.

even if there are alternate paths with worse performance but still just as satisfactory. The other allocation methods spread flows across available paths instead of assigning them to a single, attractive path, and those details are in Appendix G.

Service demand satisfaction. We now show that OTTER meets the design goal of placing fine-grained flow service demands. We choose a mean flow arrival rate of 40K flows/sec, which is the most difficult workload we evaluate, to show a lower bound on service demand satisfaction. All other parameters remain as in Table 4. For each flow, we compute its average "satisfaction" for each performance metric over its assigned paths, defined as the tolerance coefficient normalized to the fraction of bytes successfully allocated.

Figure 10 shows flow satisfaction distribution for RTT, jitter, and packet loss. Throughput satisfaction is the fraction of bytes successfully allocated, previously shown in Figure 8. Across all three service metrics, OTTER's periodic re-optimizing methods map a similar fraction of flows to paths with perfect satisfaction compared to GREEDY, but allocate far fewer flows to paths with extremely poor (<0.01) satisfaction. For RTT, PER+PATHPIN and PER+DSTPIN map approximately 47% of flows to paths with perfect RTT satisfaction, while GREEDY does so for 41% of flows. PER+PATHPIN and PER+DSTPIN allocate significantly fewer flows to paths that are entirely unsatisfactory (35% vs. 44%). Note that even with optimal allocation, OTTER still assigns at least 10% of flows to paths with poor RTT satisfaction (<0.01). Most of these flows belong to application profiles with service demands that are extremely difficult to meet alongside a large number of other flows.

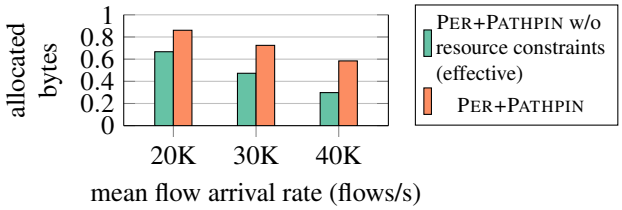


Figure 11: Impact of resource constraints on allocated bytes.

Best heuristic. Of the four realistic heuristics, we choose PER+PATHPIN for OTTER. It limits disruption of extremely performance sensitive flows, while still offering high allocated bytes and satisfaction.

Impact of edge resource constraints. In the disaggregated 5G network architecture, edge computing plays a central role in meeting service demands. However, TE systems do not account for the heterogenous availability of compute at edge sites, and may over-subscribe flows to destinations.

To demonstrate the value of explicitly considering destination resource capacities when routing 5G flows, we modify OTTER's Optimizer to ignore destination resource constraints (eliminating constraint $c4$ in Table 2). This effectively gives all destinations infinite resource capacity. The LP is restricted only by link capacities. While the unconstrained LP is seemingly able to allocate more bytes, the actual amount is much lower due to resource contention at destination sites. To estimate the amount of traffic that can actually be serviced, we reduce allocated bytes from flows that are assigned to over-subscribed destination sites. We follow random priority when de-allocating, *i.e.,* for each destination, we repeatedly select a flow at random and decrease the number of bytes allocated until the resource limitations are met. Figure 11 compares the effective fraction of bytes allocated between OTTER using PER+PATHPIN with and without edge resource constraints for different traffic loads. As we can see, ignoring destination resource constraints in the Optimizer formulation results in 23–50% fewer bytes allocated than PER+PATHPIN. This shows that the added complexity of integrating resource capacity constraints allows OTTER to better map flows to destinations that have the capacity to serve them, allowing more traffic to be supported.

8 Miscellany

Why does BGP not get in the way? Cellular network operators tend to be large ISPs that directly peer with public clouds in multiple locations. Our evaluation setup in §7 represents that, with both WANs directly peering and our use

of VPN gateways at the overlay layer. Not only do we expect 5G traffic to flow over direct connections, we expect 5G cloudification to spur operators to increase the quantity of peering. This has two implications for OTTER. First, many of the oddities of BGP in the wild do not apply here, where intermediate ASes enforce their policies between source and destination ASes. Second, we expect to see even more diversity of inter-domain paths that OTTER can exploit.

Why not optimize for WAN costs? OTTER does not explicitly incorporate monetary cost as an optimization criteria. Cloud compute and network costs tend to be uniform within countries. Most 5G network deployments are at national scales. Operators and clouds that they depend on use direct peering. Hence, optimizing for WAN costs is not a priority for OTTER and we leave that for future work.

Will OTTER negatitvely impact existing WAN TE? OTTER works within the confines of TE in both WANs — the routes available to OTTER are what the underlying TE has exposed to applications. OTTER then can only influence the traffic matrix, which TE systems effectively optimize over. If OTTER-induced demands are sufficiently high, TE systems can change the routes available to applications, including OTTER. OTTER will in turn adapt to these new routes and still pick the best ones for 5G workloads. While there is a feedback loop between OTTER and TE, it is not one that can be exploited for reducing the efficiency of existing TE. In Appendix H, we explore the value of using paths that underlying TE systems create but do not expose to applications.

9 Related Work

Overlay networks. Researchers have proposed overlay networks to improve underlying network availability, performance, or security [4, 5, 17, 22, 36, 59, 70, 77]. Recently, Skyplane [40] proposed a cloud overlay for inter-cloud bulk transfers between object stores, and selects routes that balance price and throughput. Routes and VMs are chosen using static throughput and cost measurements and participate in store-and-forward queuing. Hercules [29] is another bulk-transfer tool for overlays. CRONets [18] deploys overlay nodes on cloud networks and leverages MPTCP through overlay paths to improve throughput. CloudCast [72] focuses on latency improvement using cloud overlays.

OTTER also uses an overlay network to interconnect a 5G operator WAN and a cloud WAN. Unlike prior work, OTTER optimizes for various 5G NFs and applications, each of which can have different performance requirements, not just bulk data transfer. OTTER optimizes network paths online by using dynamic measurements for a wide range of metrics, including RTT, jitter, and loss. We achieve higher inter-cloud, per-VM throughput than prior work [40], in part because our security and privacy oriented topology design does not suffer from limits that some clouds impose on bandwidth to public IPs (Appendix B). OTTER also allocates both the path of flows as well as the compute destination.

Cloud TE. Existing cloud TE systems (such as Taiji [19], COPE [87], SWAN [39], B4 [41], BwE [46], OneWAN [45], Cascara [76], Espresso [94]) maximize network utilization or bandwidth fairness, or minimize cost for traffic grouped by coarse-grained priority classes. OTTER is not a replacement for existing TE, as underlay paths will still be subject existing TE. OTTER is an enhancement that can optimize at the granularity required for 5G applications by selecting overlay paths between multiple destinations across WANs. Additionally, OTTER performs dynamic flow placement and considers server resource capacities at endpoints, while TE periodically allocates batched traffic predictions and only considers network resource constraints. While some recent TE work [50] to optimize more fine-graned flows for applications, this optimization still takes place within a single WAN, and does not consider the multi-WAN setting of OTTER.

Multi-WAN collaboration. Unlike TE systems, OTTER optimizes traffic that spans two WANs. Prior work (Nexit [48], Wiser [47], P4P [90], Flow Director [68]) attempt to facilitate coordination between two independent entities by allowing them to exchange information while preserving privacy. Nexit and Wiser allow two ISPs to route traffic efficiently, despite competing interests. Nexit uses a negotiation-based approach, while downstream ISPs using Wiser tag routing advertisements with costs used by upstream ISPs. P4P improves peer-to-peer application performance by allowing ISPs to share performance and topology information with applications. Flow Director uses ALTO [2] to share information between ISPs and CDNs to optimize content placement. However, these works require negotiation to ensure mutually beneficial routing. OTTER makes use of the fact that, for 5G networks, these incentives are already aligned – operators pay to deploy resources on cloud networks to facilitate better QoS for 5G flows. OTTER does not require any buy-in from cloud platforms, so routing and resource placement can be done without coordination.

Better routing. Attempts to improve interdomain routing include multipath extensions to BGP [38, 92] and interdomain bandwidth reservation such as N-Tube [88]. New routing protocols that allow nodes to select paths that optimize multiple metrics include Routing on Multiple Optimality Criteria [78]. Such approaches may improve path diversity or even path selection. However, they must still optimize routing based on static metrics such as BGP AS path length or width, and cannot adjust based on dynamic metrics such as loss or latency. Deploying any of these technologies requires changes to BGP, which has proven to be a major practical impediment. SCION [95] is a novel Internet routing architecture that seeks to provide high security and availability. However, it requires ISP buy-in to deploy border routers to participate. RouteScout [7], PAINTER [43] and SCULP-

TOR [6] reroute traffic based on traffic performance degradation. However, such optimizations are performed locally or per prefix. In contrast, OTTER optimizes the entire overlay according to the performance demands of all flows, allocates per flow, and does not require deploying new devices.

Better path selection at the edge. In Tango [16], edge networks coordinate to expose more paths, enable better measurements, and route traffic efficiently. Rather than coordinate paths between edge networks, OTTER directly sets up overlay paths between the cloud provider and 5G operators by orchestrating resources on the cloud network. OTTER also routes traffic according to application-specific service demands and relies only on existing cloud network technologies. Other work [42] selects paths between a private WAN and the public Internet for video conferencing applications (e.g., Teams) based on extensive measurement data. Like OTTER, this work jointly optimizes the path and destination DC selection. However, unlike OTTER, it neither considers the general setting of 5G applications nor the coordination of paths among multiple ASNs. Any of these mechanisms that expose multiple paths to an endpoint could be utilized at the transport layer by multipath TCP.

Virtual Network Function (VNF) placement. Systems for VNF placement [62, 63, 82] optimize for metrics such as cost and CPU utilization of NFs. However, unlike OTTER, they do not consider the QoS needs of individual flows, and are evaluated on simulated setups. VNF placement systems periodically optimize for coarse-grained objectives, according to predicted future load, but OTTER dynamically places flows and resources, on-demand, according to customizable fine-grained service objectives.

10 Conclusions

The next generation of 5G networks includes new frequency bands and radio technology, equipping users with access links so fast that their E2E performance will become bottlenecked by inter-domain paths [91] rather than last mile access links. Moreover, next generation 5G networks will rely on hyperscalers to deploy software-based 5G NFs [61] and interactive applications. Thus, we need to improve the performance of paths across operator and cloud WANs.

We demonstrate the value of constructing such paths over two commercial WANs. OTTER achieves 13% higher average throughput or 6-10 Gbps in the maximum, RTT reduction as much as 42 ms within the US, jitter reduction by 45% on average, and 0.06% lower packet loss. We designed and implemented a scalable algorithm to jointly optimize the placement of workloads on available paths and compute resources. It allocates 26%–45% more bytes on the network than good baselines while also satisfying the service demands of more flows. While we focus on the urgency of 5G, our research contributions may be relevant in other upcoming access technologies such as 6G.

OTTER unlocks avenues of future work. Underlay paths that are private to one WAN can provide even more diversity in end-to-end path performance than we have shown – how can they be leveraged? How will OTTER work in topologies where a 5G operator relies on two or more hyperscalers?

References

[1] 3GPP. Service requirements for enhanced V2X scenarios. https://www.etsi.org/deliver/etsi_ts/122100_122199/122186/16.02.00_60/ts_122186v160200p.pdf, 2020.

[2] R. Alimi, R. Penno, Y. Yang, S. Kiesel, S. Previdi, W. Roome, S. Shalunov, and R. Woundy. Application-Layer Traffic Optimization (ALTO) Protocol. RFC 7285, September 2014.

[3] The Telco Global API Alliance. Quality of Demand (QoD) in CAMARA. https://camaraproject.org/quality-on-demand/, 2025.

[4] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. *SIGOPS Oper. Syst. Rev.*, 35(5), Oct 2001.

[5] David G. Andersen, Alex C. Snoeren, and Hari Balakrishnan. Best-Path vs. Multi-Path Overlay Routing. In *Internet Measurement Conference*, IMC '03, New York, NY, USA, 2003. ACM.

[6] anonymous. SCULPTOR Has Your Back(up Path): Carving Interdomain Routes to Services. In *under submission to USENIX NSDI 2025*.

[7] Maria Apostolaki, Ankit Singla, and Laurent Vanbever. Performance-Driven Internet Path Selection. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, New York, NY, USA, 2021. Association for Computing Machinery.

[8] AT&T. AT&T Moves 5G Mobile Network to Microsoft Cloud. https://about.att.com/story/2021/att_microsoft_azure.html, 2021.

[9] ATT, BT, CenturyLink, China Mobile, Colt, Deutsche Telekom, KDDI, NTT, Orange, Telecom Italia, Telefonica, Telstra, and Verizon. Network Functions Virtualisation (NFV) White Paper. In *SDN and OpenFlow World Congress*. ETSI https://portal.etsi.org/nfv/nfv_white_paper.pdf, October 2012.

[10] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for Traffic Engineering Over MPLS. In *RFC 2702*, 1999.

[11] D. O. Awduche. MPLS and traffic engineering in IP networks. *IEEE Communications Magazine*, 37(12), 1999.

[12] AWS and Verizon. Architecting Geo-Distributed Mobile Edge Application With Consul. https://www.youtube.com/watch?app=desktop&v=At6cHrL6s0Q, 2022.

[13] Azure. Azure Ev3 and Esv3-series VMs. https://learn.microsoft.com/en-us/azure/virtual-machines/ev3-esv3-series, 2024.

[14] Azure. Azure VPN Gateway SKUs. https://learn.microsoft.com/en-us/azure/vpn-gateway/about-gateway-skus, 2024.

[15] Microsoft Azure. What is Azure Programmable Connectivity? https://learn.microsoft.com/en-us/azure/programmable-connectivity/azure-programmable-connectivity-overview, 2025.

[16] Henry Birge-Lee, Maria Apostolaki, and Jennifer Rexford. It Takes Two to Tango: Cooperative Edge-to-Edge Routing. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, New York, NY, USA, 2022. ACM.

[17] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat. Opus: an overlay peer utility service. In *IEEE Open Architectures and Network Programming Proceedings. OPENARCH 2002*, 2002.

[18] Chris X. Cai, Franck Le, Xin Sun, Geoffrey G. Xie, Hani Jamjoom, and Roy H. Campbell. Cronets: Cloud-routed overlay networks. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[19] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, Kevin Cole, and Dmitri Perelman. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, New York, NY, USA, 2019. ACM.

[20] Cisco. The Cisco 5G Strategy Series: Packet Core, Transport, and Identity Management. https://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/ultra-services-platform/5g-strategy-series.pdf, 2016.

[21] Cloud and Colocation. AT&T Data Center Locations. https://cloudandcolocation.com/colocation-provider/att/, 2024.

[22] Andy Collins. The detour framework for packet rerouting. 1998.

[23] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr, 2024.

[24] Ericsson. Traffic patterns and network design: To achieve optimum 5G performance, both coverage and capacity must be available throughout the network according to location-specific needs. https://www.ericsson.com/en/reports-and-papers/mobility-report/articles/traffic-patterns-drive-network-evolution, 2024.

[25] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *IEEE Communications Magazine*, 40(10), 2002.

[26] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proceedings IEEE INFOCOM 2000*, volume 2, 2000.

[27] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. Using Deep Programmability to Put Network Owners in Control. In *SIGCOMM CCR*, 2020.

[28] The Linux Foundation. CAMARA: The Telco Global API Alliance. https://camaraproject.org/, 2024.

[29] Marten Gartner, Jean-Pierre Smith, Matthias Frei, Francois Wirz, Cédric Neukom, David Hausheer, and Adrian Perrig. Hercules: High-Speed Bulk-Transfer over SCION. In *IFIP Networking*, 2023.

[30] Google. Cross-Cloud Interconnect overview. https://cloud.google.com/network-connectivity/docs/interconnect/concepts/cci-overview, 2024.

[31] Google. Google n2-standard-32 VMs. https://gcloud-compute.com/n2-standard-32.html, 2024.

[32] Google. Routing in Google Cloud. https://cloud.google.com/vpc/docs/routes, 2024.

[33] GooRAM. 5G: What's the NRF? https://gooram.medium.com/5g-whats-the-nrf-7db7a83cb1bd, 2023.

[34] GSMA. Open Gateway API Descriptions. https://www.gsma.com/futurenetworks/gsma-open-gateway-api-descriptions/, 2024.

[35] Gurobi Optimization. Gurobi optimizer reference manual. http://www.gurobi.com, 2019.

[36] J. Han, D. Watson, and F. Jahanian. Topology aware overlay networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 4, 2005.

[37] Red Hat. Automatic & Zero-Touch Provisioning for the RAN. https://www.youtube.com/watch?v=2m1CVD5ytgI, 2023.

[38] Jiayue He and Jennifer Rexford. Toward internet-wide multipath routing. *IEEE Network*, 22(2), 2008.

[39] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, 2013.

[40] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. Skyplane: Navigating the Cost and Performance Trade-off for Cloud Bulk Transfers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[41] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *ACM SIGCOMM*, 2013.

[42] Bhaskar Kataria, Palak LNU, Rahul Bothra, Rohan Gandhi, Debopam Bhattacherjee, Venkata N Padmanabhan, Irena Atov, Sriraam Ramakrishnan, Somesh Chaturmohta, Chakri Kotipalli, et al. Saving private wan: Using internet paths to offload wan traffic in conferencing services. *arXiv preprint arXiv:2407.02037*, 2024.

[43] Thomas Koch, Shuyue Yu, Sharad Agarwal, Ethan Katz-Bassett, and Ryan Beckett. PAINTER: Ingress Traffic Engineering and Routing for Enterprise Cloud Networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23. Association for Computing Machinery, 2023.

[44] Umesh Krishnaswamy, Rachee Singh, Nikolaj Bjørner, and Himanshu Raj. Decentralized cloud wide-area network traffic engineering with BlastShield. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[45] Umesh Krishnaswamy, Rachee Singh, Paul Mattes, Paul-Andre C Bissonette, Nikolaj Bjorner, Zahira Nasrin, Sonal Kothari, Prabhakar Reddy, John Abeln, Srikanth Kandula, Himanshu Raj, Luis Irun-Briz, Jamie Gaudette, and Erica Lan. OneWAN is better than two: Unifying a split WAN architecture. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[46] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. *SIGCOMM Comput. Commun. Rev.*, 45(4), Aug 2015.

[47] Ratul Mahajan and David Wetherall. Mutually Controlled Routing with Independent ISPs . In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*, Cambridge, MA, apr 2007. USENIX Association.

[48] Ratul Mahajan, David Wetherall, and Thomas Anderson. Negotiation-Based Routing between Neighboring ISPs. In *2nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI'05, USA, 2005. USENIX Association.

[49] Mellanox. sockperf: Network Benchmarking Utility. https://github.com/Mellanox/sockperf, 2024.

[50] Congcong Miao, Zhizhen Zhong, Yunming Xiao, Feng Yang, Senkuo Zhang, Yinan Jiang, Zizhuo Bai, Chaodong Lu, Jingyi Geng, Zekun He, et al. Megate: Extending wan traffic engineering to millions of endpoints in virtualized cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 103–116, 2024.

[51] Microsoft. About NAT on Azure VPN Gateway. https://learn.microsoft.com/en-us/azure/vpn-gateway/nat-overview, 2023.

[52] Microsoft. Azure Operator Voicemail: Take the first step to move voice workloads to the cloud. https://techcommunity.microsoft.com/t5/azure-for-operators-blog/azure-operator-voicemail-take-the-first-step-to-move-voice/ba-p/3751315, 2023.

[53] Microsoft. Azure Programmable Connectivity. https://techcommunity.microsoft.com/t5/azure-for-operators-blog/azure-programmable-connectivity-powers-network-aware/ba-p/3751024, 2023.

[54] Microsoft. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db, 2024.

[55] Microsoft. Azure Operator Nexus. https://azure.microsoft.com/en-us/products/operator-nexus/, 2024.

[56] Microsoft. Microsoft Azure Private 5G Core. https://azure.microsoft.com/en-us/products/private-5g-core, 2024.

[57] Microsoft. Write network-aware applications with Azure Programmable Connectivity. https://build.microsoft.com/en-US/sessions/2c7f3bc6-e021-4f5c-8749-a2544f34395d, 2024.

[58] Netflix. Internet connection speed recommendations. https://help.netflix.com/en/node/306, 2023.

[59] NetFoundry. OpenZiti. https://docs.openziti.io, 2024.

[60] PR Newswire. AT&T to run its mobility network on Microsoft's Azure for Operators cloud, delivering cost-efficient 5G services at scale. https://www.prnewswire.com/news-releases/att-to-run-its-mobility-network-on-microsofts-azure-for-operators-cloud-delivering-cost-efficient-5g-services-at-scale-301323256.html, 2021.

[61] Nokia. Nokia 5G Core Software as a Service in practice. https://www.youtube.com/watch?v=_4DHzjrtB34, 2023.

[62] Jianing Pei, Peilin Hong, Miao Pan, Jiangqing Liu, and Jingsong Zhou. Optimal VNF Placement via Deep Reinforcement Learning in SDN/NFV-Enabled Networks. *IEEE Journal on Selected Areas in Communications*, 38(2):263–278, 2020.

[63] Jianing Pei, Peilin Hong, Kaiping Xue, and Defang Li. Efficiently Embedding Service Function Chains with Dynamic Virtual Network Function Placement in Geo-Distributed Cloud System. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2179–2192, 2019.

[64] Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. Democratizing the Network Edge. In *ACM SIGCOMM Computer Communication Review*, 2019.

[65] Larry Peterson and Oguz Sunay. 5G Mobile Networks: A Systems Approach. 2022.

[66] Phone.com. How Much Bandwidth Do I Need for VOIP? https://www.phone.com/much-bandwidth-need-voip/, 2014.

[67] Petar Popovski, Kasper F. Trillingsgaard, Osvaldo Simeone, and Giuseppe Durisi. 5G Wireless Network Slicing for eMBB, URLLC, and mMTC: A Communication-Theoretic View. In *IEEE Access*, 2018.

[68] Enric Pujol, Ingmar Poese, Johannes Zerwas, Georgios Smaragdakis, and Anja Feldmann. Steering Hyper-Giants' Traffic at Scale. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, New York, NY, USA, 2019. Association for Computing Machinery.

[69] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[70] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, USA, 2004. USENIX Association.

[71] RootMetrics. Mobile cloud gaming: the real-world cloud gaming experience in Los Angeles. https://rootmetrics.com/en-US/content/us-LA-gaming-report-2020, 2020.

[72] Noga H. Rotman, Yaniv Ben-Itzhak, Aran Bergman, Israel Cidon, Igor Golikov, Alex Markuze, and Eyal Zohar. CloudCast: Characterizing Public Clouds Connectivity. *CoRR*, abs/2201.06989, 2022.

[73] Mahadev Satyanarayanan. The Emergence of Cloud Computing. In *IEEE Computer*, 2017.

[74] Mahadev Satyanarayanan, Guenter Klas, Marco Silva, and Simone Mangiante. The Seminal Role of Edge-Native Applications. In *IEEE International Conference on Edge Computing (EDGE)*, 2019.

[75] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas Anderson. The End-to-End Effects of Internet Path Selection. In *SIGCOMM*, 1999.

[76] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective Cloud Edge Traffic Engineering with Cascara. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[77] Ramesh K. Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. Overlay Networks: An Akamai Perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*, 2014.

[78] João Luís Sobrinho and Miguel Alves Ferreira. Routing on Multiple Optimality Criteria. SIGCOMM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[79] 3GPP Technical Specification. 3GPP TS 23.501 version 16.6.0 Release 16. https://www.3gpp.org/ftp//Specs/archive/23_series/23.501/23501-g60.zip, 2020.

[80] 3GPP Technical Specification. 5G System Network Function Repository Services. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3345, 2023.

[81] Neil Spring, Ratul Mahajan, and Thomas Anderson. Quantifying the Causes of Path Inflation. In *SIGCOMM*, 2003.

[82] Jie Sun, Feng Liu, Huandong Wang, and Dapeng Oliver Wu. Joint VNF Placement, CPU Allocation, and Flow Routing for Traffic Changes. *IEEE Internet of Things Journal*, 10(2):1208–1222, 2023.

[83] Zoom Support. Zoom system requirements: Windows, macOS, Linux. https://support.zoom.us/hc/en-us/articles/201362023-Zoom-system-requirements-Windows-macOS-Linux, 2023.

[84] Qualcomm Technologies. Making 5G NR a reality. Technical report, 2016.

[85] Qualcomm Technologies. VR and AR pushing connectivity limits. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/presentation_vr_and_ar_are_pushing_connectivity_limits_web_0.pdf, 2018.

[86] Terraform. Automate infrastructure on any cloud. https://www.terraform.io/, 2024.

[87] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *ACM SIGCOMM*, 2006.

[88] Thilo Weghorn, Si Liu, Christoph Sprenger, Adrian Perrig, and David Basin. N-Tube: Formally Verified Secure Bandwidth Reservation in Path-Aware Internet Architectures. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, 2022.

[89] Xipeng Xiao, A. Hannan, B. Bailey, and L. M. Ni. Traffic Engineering with MPLS in the Internet. *Netwrk. Mag. of Global Internetwkg.*, 14(2), March 2000.

[90] Haiyong Xie, Y. Richard Yang, Arvind Krishnamurthy, Yanbin Grace Liu, and Abraham Silberschatz. P4p: Provider Portal for Applications. SIGCOMM '08, New York, NY, USA, 2008. Association for Computing Machinery.

[91] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding Operational 5G: A First Measurement Study on Its Coverage, Performance and Energy Consumption. In *ACM SIGCOMM*, 2020.

[92] Wen Xu and Jennifer Rexford. MIRO: Multi-Path Interdomain Routing. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, New York, NY, USA, 2006. Association for Computing Machinery.

[93] Zhiying Xu, Francis Y. Yan, Rachee Singh, Justin T. Chiu, Alexander M. Rush, and Minlan Yu. Teal: Learning-accelerated optimization of wan traffic engineering. In *Proceedings of the ACM SIGCOMM 2023 Conference*. Association for Computing Machinery, 2023.

[94] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *ACM SIGCOMM*, 2017.

[95] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In *2011 IEEE Symposium on Security and Privacy*, 2011.

A Multi-WAN US Topology

Figure 19 shows all the Azure and GCP cloud components (VMs, VPN gateways, etc.) and the connections between them on the continental US-wide topology we have deployed for evaluating OTTER. It is generated from the output of our Terraform deployment, using a Terraform visualizer called Rover. While difficult to read, the figure indicates the complexity and scale of our evaluation topology across 8 Azure and 8 GCP regions, as it contains many cloud components and connections between them.

Table 6 shows the 8 sets of VPN connections that we use in our evaluation topology. For example, the first row shows that GCP us-central1 is peered with Azure centralus. Each peering consists of 4 VPN connections, each with 2 tunnels, for improved throughput and resilience. This number can be increased or decreased as needed, or further augmented [30], but we have found that 8 tunnels gives us a good tradeoff

GCP Region name	GCP Region location		Azure Region name	Azure Region location
us-central1	Iowa	⟺	centralus	Iowa
us-east1	South Carolina	⟺	eastus	Virginia
us-east4	Virginia	⟺	eastus2	Virginia
us-east5	Ohio	⟺	northcentralus	Illinois
us-south1	Texas	⟺	southcentralus	Texas
us-west3	Utah	⟺	westcentralus	Wyoming
us-west2	California	⟺	westus	California
us-west4	Nevada	⟺	westus3	Arizona

Table 6: VPN connections between Azure and GCP regions.

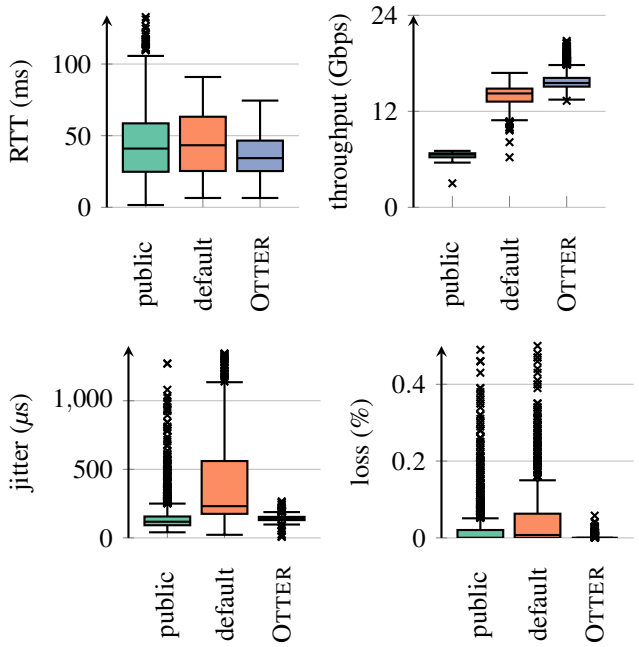


Figure 12: Network performance of paths available by different mechanisms. The jitter vertical axis is chopped at 1200 μs for clarity, and the loss vertical axis is chopped at 0.5%.

between cost and throughput. In picking which Azure region peers with which GCP region, our goal was to mimic how a 5G operator would peer with a cloud provider. Such peerings occur at carrier hotels or "meet-me" points, where two or more ASes may co-locate their routing equipment. Hence we strove to pick pairs of regions that were physically close to each other. Note that when we evaluated OTTER, there was one additional US region offered by both clouds, however we were unable to acquire sufficient compute capacity in those regions.

B Additional Analysis of OTTER Orchestrator Performance

Figure 12 summarizes the characteristics of the paths available by different mechanisms. The "default" box plots show the performance spread between all Src-Dst pairs in our topology (§7.1) across the 24 hour measurement period, when not controlling the path and allowing the two WANs to pick the path [32]. The "public" box plots show the performance spread without any of our security and privacy infrastructure (i.e., no private VNets, no VPN gateways, no VNet peering), where the source and destination use public IP addresses and nothing specified by us in between. The OTTER box plots show the best performing OTTER Orchestrator path for each Src-Dst pair for the relevant network performance metric.

We make several observations. Our secure, private, and scalable deployment topology offers far higher throughput than when using public routing. The {min, average, max} throughput for public is {3.0 Gbps, 6.5 Gbps, 7.1 Gbps}, while for default it is {6.3 Gbps, 14.0 Gbps, 16.8 Gbps}. As prior work [40] noted, some clouds such as GCP limit outbound transfers to 7 Gbps, but those limits do not apply to our use of private VNets and VPN connections. OTTER is able to further improve these numbers to {13.3 Gbps, 15.8 Gbps, 20.8 Gbps}. The median, 25th, and 75th percentile RTTs are slightly higher for default compared to public, but with far less spread above the 75th percentile. OTTER offers the lowest RTT at the 25th and higher percentiles. OTTER also offers the tightest jitter and lowest loss.

Figure 13 shows the performance improvement when comparing different mechanisms. "O-D" shows for every Src-Dst pair, for every round of measurements, the additional performance that the OTTER Orchestrator achieves compared to the default path on our topology. "O-P" provides OTTER's improvement over the public path when using public IP addresses and no private networking functions. "P-D" shows the improvement of the public path over the default private path. Figure 14 shows the same data, but as a percentage improvement. When considering the "O-D" boxes, all the values are 0% or higher, because the OTTER Orchestrator will never pick a path that has worse performance for the desired metric. The "O-P" and "P-D" boxes show that for RTT, jitter, and loss, the use of private networking functions introduces occasional performance regressions. We believe these functions are run by cloud providers in VMs, and that introduces small amounts of occasional jitter and packet loss as traffic goes up and down the software stack. Nonetheless, OTTER improves on this, by, for example, picking paths that

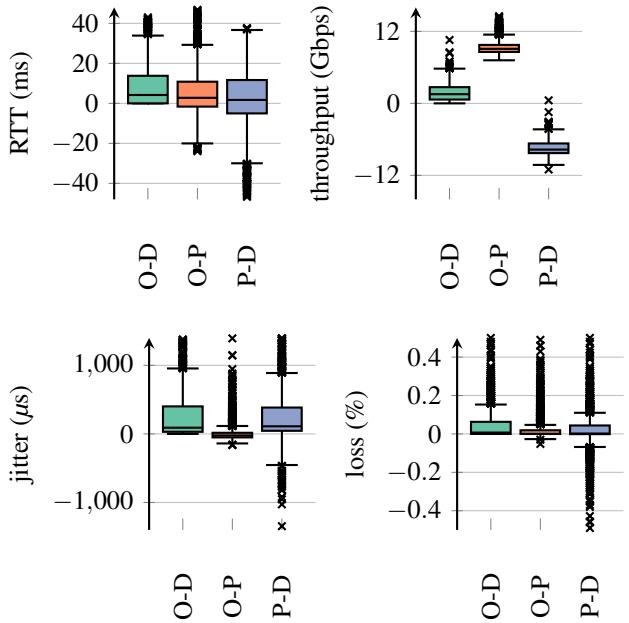


Figure 13: Improvement in network performance. On the horizontal axis, "O-D" shows the performance improvement of OTTER over default, "O-P" is OTTER over public, and "P-D" is public over default.

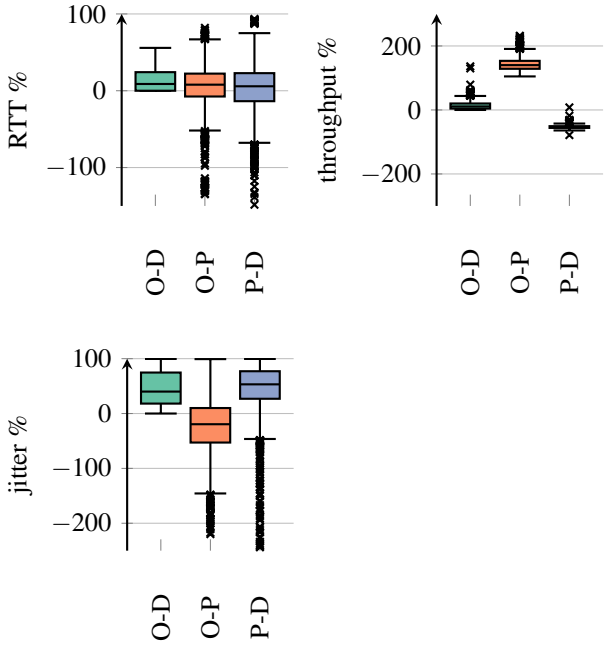


Figure 14: Percentage improvement in network performance. On the horizontal axis, "O-D" shows the performance improvement of OTTER over default, "O-P" is OTTER over public, and "P-D" is public over default. Loss is not presented because the large spread of outliers on small values makes the graph difficult to read.

go through other instances of VPN gateways that are not suffering from noise at the moment. This is seen in Figure 12, where on each metric of throughput, RTT, jitter, and loss, OTTER offers the best path.

C Evaluation of OTTER Optimizer on Different Mean Flow Durations

Table 7 summarizes the results of how well OTTER allocates flows of varying durations using the heuristics described in §7.2: GREEDY, PER+PATHPIN, PER+DSTPIN, PER, OPT+PATHPIN, OPT+DSTPIN, and OPT. These results, like results in §7.2, show that heuristics relying on a greedy allocation (GREEDY, PER+PATHPIN, PER+DSTPIN, PER) typically allocate less traffic, but are more practical for real-time decisions. We also see that with longer flow durations, the amount of bytes allocated per flow decreases for each algorithm. This is because, with longer flows, there are less resources available for later flows, because earlier flows are still active.

D Service Demand Generation

The traffic composition on 5G networks is dynamic and can evolve over time, making it difficult to predict what kinds of demands will be present. Applications will also have diverse service requirements across various performance metrics. To mimic 5G traffic in our evaluations, we generate flows by assigning them an "application profile" based on service guidelines from the 3GPP and other sources.

We selected a set of example applications outlined by the 3GPP standard [79] that represent a diverse range of performance demands that 5G networks are expected to support. These applications are listed in the first column of Table 5. In § 5.7.4 of the relevant 3GPP standard [79] are guidelines for the packet delay budget and packet error rate for each application, which we use to determine RTT and packet loss demands. These are shown in the latency and loss columns in Table 5. To obtain RTT demands, we double the one-way latency guidelines listed in that 3GPP standard. However, it does not provide information on throughput and jitter, so we infer them from other sources [1, 58, 66, 71, 83, 85]. In cases where the throughput demands are unclear or conflicting, we set a default of 1 Mbps for a single 4-tuple flow. We chose real-time gaming as one application with explicit jitter demands, doubling the 10 ms one-way jitter guideline from [71] to match the round-trip path attributes measured on our topology.

As mentioned in §4, OTTER models service demands as a vector of demand functions which map path performance characteristics to a tolerance coefficient in the range $(0, 1]$, where 1 is best. A demand function is able to capture com-

mean flow duration(s)	mean flow arrival rate (flows/s)	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	20K	0.95	0.99	0.99	0.99	0.99	1.0	1.0
5	30K	0.61	0.84	0.84	0.93	0.92	0.95	0.98
5	40K	0.60	0.78	0.82	0.85	0.85	0.87	0.91
10	20K	0.64	0.80	0.82	0.91	0.87	0.90	0.96
10	30K	0.55	0.74	0.76	0.83	0.81	0.83	0.87
10	40K	0.44	0.64	0.64	0.66	0.68	0.69	0.70
20	20K	0.50	0.71	0.71	0.79	0.79	0.79	0.84
20	30K	0.30	0.47	0.47	0.54	0.54	0.54	0.58
20	40K	0.17	0.34	0.34	0.41	0.40	0.41	0.45
30	20K	0.36	0.54	0.55	0.68	0.66	0.66	0.72
30	30K	0.21	0.38	0.38	0.47	0.43	0.44	0.50
30	40K	0.23	0.35	0.35	0.37	0.37	0.38	0.39
60	20K	0.45	0.64	0.63	0.67	0.67	0.68	0.69
60	30K	0.32	0.45	0.46	0.49	0.47	0.49	0.50
60	40K	0.21	0.32	0.33	0.35	0.34	0.35	0.36

Table 7: Fraction of allocated bytes across flows for varying mean flow durations.

plex service demands, which we believe will be necessary to capture the QoS characteristics expected by 5G applications. In our implementation, a demand function can be either a step, linear, or rational function, with inflection points randomly selected from within ±25% of the service guidelines mentioned above. For flows that are indifferent to the metric (*e.g.,* jitter), the demand function is constant at 1.

The service demand generation process is as follows: each flow is first randomly assigned an application profile. For each performance metric, we randomly assign a function type, then generate inflection points. For example, an augmented reality flow may have a step-type demand function for RTT that looks like the following:

$$\mathcal{D}_f^{rtt}(\sigma_p^{rtt}) = \left\{ \begin{array}{ll} 1, & \text{if } \sigma_p^{rtt} \leq 21.2\ ms \\ \varepsilon, & \text{if } \sigma_p^{rtt} > 21.2\ ms \end{array} \right\}$$

We then randomly select a value within ±25% of the application's guideline throughput demand.

E Evaluation of OTTER Optimizer on Different Flow Distributions

Table 8 summarizes the results of how well OTTER satisfies flow RTT requirements for varying distributions of flows, using the heuristics described in §7.2. We vary the proportion of flows with strict RTT requirements (RTT $\leq$ 60ms) and report the average RTT satisfaction for those RTT-sensitive flows. We find that for lower arrival rates, the proportion of RTT-sensitive flows does not affect their satisfaction, as there is less contention for resources. Like the results in §7.2, the GREEDY heuristic provides the lowest satisfaction. The heuristics that pin flows to either paths or destinations also provide less satisfaction than PER and OPT, as RTT-sensitive flows may get pinned to a high-RTT path, even as capacity on lower-RTT paths becomes available.

F Evaluation of OTTER Optimizer on Different Network Sizes

Table 9 summarizes the results of how well OTTER allocates flows on networks of various sizes using the same mean flow arrival rates and the heuristics described in §7.2: GREEDY, PER+PATHPIN, PER+DSTPIN, PER, OPT+PATHPIN, OPT+DSTPIN, and OPT. These results are qualitatively similar to those described in §7.2, with OPT being able to allocate more traffic compared to the other algorithms, but the PER heuristics scaling better with higher traffic load on the same network topology. We can additionally see that as we increase the number of paths between each Src-Dst pair, all algorithms are able to improve the amount of bytes allocated per flow. This is explained by the increase in path diversity, which gives more options for flows to be mapped to paths that better suit their service demands. The same effect can be seen as the number of sources and destinations increases. This increases the overall resource capacity of the network, since there are more links and compute resources for the same traffic load.

We also evaluate OTTER on larger topologies with larger flow arrival rates. We choose topology sizes to match the scale of contemporary TE systems, B4 [41] (approximately 30 DC sites globally) and SWAN [39] (evaluated with up to 40 DC nodes). In each topology, we set 75% of the destinations to be edge nodes, and set the rest to be DC nodes. We generate path, destination, and flow attributes (except flow arrival rates) using the values in Table 4. We scale up the flow arrival rates because as the topology grows, smaller arrival rates do not produce enough traffic to cause resource contention, and OTTER is able to achieve 100% allocation. We set the mean flow arrival rate to the number of source nodes times the number of destination nodes. The results are summarized in Table 10. The allocation from the OPT+PATHPIN, OPT+DSTPIN, and OPT heuristics are qualitatively similar to

% flows with RTT $\leq$ 60ms	mean flow arrival rate (flows/s)	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
30	20K	0.34	0.38	0.48	0.77	0.50	0.57	0.85
30	30K	0.33	0.36	0.43	0.65	0.43	0.52	0.71
30	40K	0.22	0.17	0.22	0.39	0.23	0.31	0.43
50	20K	0.48	0.52	0.56	0.81	0.57	0.65	0.87
50	30K	0.31	0.33	0.37	0.58	0.38	0.46	0.64
50	40K	0.20	0.21	0.27	0.46	0.27	0.34	0.53
70	20K	0.48	0.52	0.58	0.79	0.55	0.66	0.85
70	30K	0.22	0.27	0.29	0.53	0.34	0.44	0.59
70	40K	0.21	0.24	0.26	0.46	0.29	0.38	0.53

Table 8: Average RTT satisfaction for flows with varying application distributions.

		Small (4 srcs, 4 dsts)						
paths per src/dst pair	mean flow arrival rate (flows/s)	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	20K	0.18	0.36	0.36	0.38	0.36	0.37	0.40
5	30K	0.16	0.27	0.27	0.28	0.28	0.27	0.29
5	40K	0.15	0.20	0.20	0.20	0.20	0.21	0.21
10	20K	0.31	0.56	0.57	0.62	0.59	0.60	0.64
10	30K	0.27	0.43	0.43	0.45	0.45	0.45	0.47
10	40K	0.23	0.31	0.31	0.31	0.32	0.32	0.32
20	20K	0.41	0.65	0.63	0.83	0.74	0.72	0.88
20	30K	0.34	0.54	0.56	0.63	0.61	0.62	0.68
20	40K	0.20	0.36	0.36	0.46	0.45	0.45	0.53
		Medium (8 srcs, 8 dsts)						
paths per src/dst pair	mean flow arrival rate (flows/s)	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	20K	0.78	0.87	0.89	0.93	0.91	0.92	0.95
5	30K	0.34	0.56	0.57	0.67	0.63	0.64	0.72
5	40K	0.40	0.55	0.56	0.56	0.58	0.58	0.60
10	20K	0.74	0.86	0.90	0.95	0.93	0.93	0.99
10	30K	0.48	0.69	0.71	0.75	0.76	0.77	0.82
10	40K	0.52	0.68	0.69	0.73	0.76	0.78	0.79
20	20K	0.92	0.99	0.99	0.99	0.99	0.99	1.0
20	30K	0.59	0.79	0.81	0.88	0.91	0.92	1.0
20	40K	0.52	0.68	0.70	0.79	0.83	0.84	0.94
		Large (12 srcs, 12 dsts)						
paths per src/dst pair	mean flow arrival rate (flows/s)	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	20K	0.95	0.98	0.99	0.99	0.99	0.99	1.0
5	30K	0.71	0.88	0.89	0.93	0.95	0.96	0.99
5	40K	0.61	0.77	0.80	0.87	0.87	0.91	0.97
10	20K	1.0	0.99	1.0	1.0	0.99	1.0	1.0
10	30K	0.93	0.99	0.99	0.99	0.99	0.99	1.0
10	40K	0.65	0.82	0.85	0.88	0.96	0.96	0.99
20	20K	1.0	0.99	1.0	1.0	0.99	1.0	1.0
20	30K	0.95	0.99	0.99	1.0	0.99	0.99	0.99
20	40K	0.79	0.93	0.93	0.94	0.99	0.99	1.0

Table 9: Fraction of allocated bytes across flows for varying network sizes.

(12 srcs, 12 dsts)								
paths per src/dst pair	**mean flow arrival rate (flows/s)**	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	72K	0.34	0.49	0.49	0.53	0.59	0.60	0.64
8	72K	0.38	0.54	0.55	0.59	0.66	0.67	0.72
10	72K	0.45	0.61	0.63	0.62	0.74	0.77	0.81
20	72K	0.49	0.64	0.66	0.62	0.84	0.84	0.95
(16 srcs, 16 dsts)								
paths per src/dst pair	**mean flow arrival rate (flows/s)**	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	128K	0.30	0.48	0.47	0.51	0.55	0.56	0.59
8	128K	0.29	0.45	0.45	0.50	0.55	0.57	0.64
10	128K	0.30	0.46	0.47	0.53	0.60	0.62	0.69
20	128K	0.42	0.57	0.59	0.57	0.79	0.82	0.90
(20 srcs, 20 dsts)								
paths per src/dst pair	**mean flow arrival rate (flows/s)**	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	200K	0.27	0.45	0.45	0.51	0.54	0.55	0.59
8	200K	0.30	0.48	0.48	0.54	0.58	0.60	0.67
10	200K	0.31	0.49	0.49	0.54	0.62	0.62	0.70
20	200K	0.34	0.50	0.52	0.55	0.67	0.69	0.77
(32 srcs, 32 dsts)								
paths per src/dst pair	**mean flow arrival rate (flows/s)**	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	512K	0.20	0.36	0.37	0.43	0.42	0.42	
8	512K	0.23	0.39	0.39	0.45	0.47	0.46	
10	512K	0.25	0.41	0.42	0.47	0.49		
20	512K	0.27	0.42	0.43	0.42			
(64 srcs, 64 dsts)								
paths per src/dst pair	**mean flow arrival rate (flows/s)**	GREEDY	PER+PATHPIN	PER+DSTPIN	PER	OPT+PATHPIN	OPT+DSTPIN	OPT
5	2M	0.17	0.25	0.26	0.24			
8	2M	0.18	0.25	0.25	0.26			
10	2M	0.19	0.22	0.23	0.20			
20	2M	0.18	0.21	0.23	0.19			

Table 10: Fraction of allocated bytes across flows for varying network sizes, with mean flow arrival rates scaled according to network size. Some values for OPT heuristics are missing because of the excessive and unrealistic compute times these heuristics need.

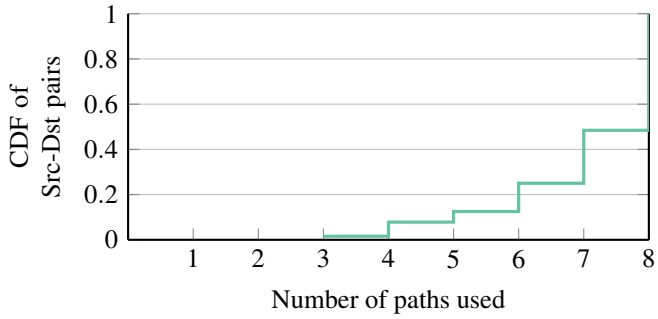


Figure 15: Number of paths between each Src-Dst pair that carries at least one flow.

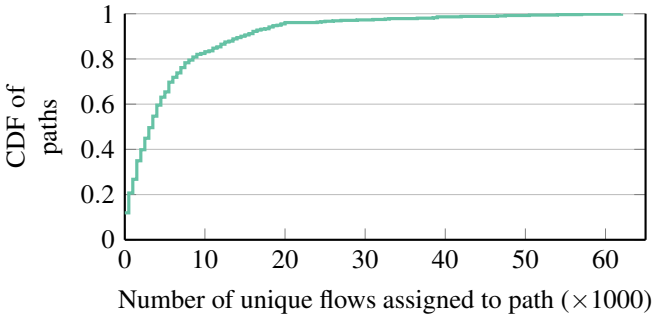


Figure 16: CDF of the number of unique flow aggregates assigned to each path in the network.

results in §7.2. As the flow arrival rate grows, there is a more significant gap between the PER+PATHPIN, PER+DSTPIN, and PER and the OPT heuristics. This is because the OTTER Optimizer has a longer runtime, which means the PER heuristics rely more on greedily allocating flows.

G OTTER's Path Usage

We present additional analysis results on the path usage of OTTER's allocation methods. This is in contrast to the service demand satisfaction and byte allocation metrics presented in the main body of the paper. We use this to demonstrate that the evaluated topologies have realistic capacity constraints and path metrics, and that the OTTER Optimizer does not simply assign the majority of flows to a single, particularly attractive, hypothetical path (*e.g.,* one that happens to have extremely low RTT, high throughput, and low loss).

Figure 15 shows the number of paths between each Src-Dst pair that is assigned at least one flow under the PER+PATHPIN allocation method. We use the default topology described in § 7.2, with eight sources, eight destinations, and eight paths between each Src-Dst pair. We can see that for more than half of all Src-Dst pairs, all eight paths are employed. In 75% of cases, at least seven paths are used. This demonstrates that OTTER does not simply favor a small number of well-provisioned paths, but that it balances traffic across those that are available to maximize satisfaction.

We further show in Figure 16 the distribution of the number of unique flows that are assigned to each path. We use a mean flow arrival rate of 40K flows per second for a period of 30 seconds from the time of the first flow. While there is a long tail in terms of number of flows assigned to each path, almost 90% of all paths in the network are allocated at least one flow within first 30 seconds.

H WAN Underlay Opportunity

The focus of OTTER is as an overlay, in setups which can be constructed by customers of public clouds without any special privileges. WANs, whether they be cloud WANs or 5G operator WANs, have a dedicated underlay network. This

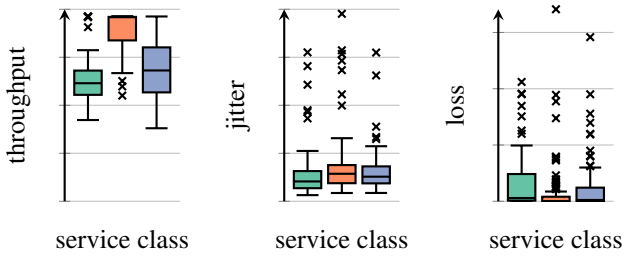


Figure 17: Relative network performance between two VMs across the US, with each box representing a different class of service. Jitter graph is chopped on vertical axis for clarity – outliers go up to 2X, 3X, 3X higher for each class from left to right.

underlay is typically comprised of layer 2 (such as MPLS or SONET) segments which are stitched together to form layer 3 links that interconnect various locations (DCs and/or PoPs). Some WANs maintain multiple paths in the underlay between any two locations [45], as well as different queues at the routers in the WAN. The combination of the two can be used to create different classes of service, with different properties (such as bandwidth, RTT, jitter, loss, reliability, cost). While typically not exposed to customers, we wonder if these underlay network capabilities could even further enhance the value of OTTER as part of a native offering by a WAN.

To briefly demonstrate the value of using these underlay capabilities, we show the relative network performance between two VMs, one in the US West coast and one in the US East coast, on a cloud WAN, over three different classes of service, on a two hour window in the middle of a US week day. We run approximately 100 iPerf3 measurements lasting 10 s each for each of UDP and TCP on each class of service.

Figure 17 shows that each class offers a different tradeoff. The class in the middle tends to offer more throughput but also slightly more jitter than the other two. The class on the right offers more throughput and less loss than the left, but more uncertainty in the throughput. As shown in Figure 18, these tradeoffs are not static, necessitating a dynamic approach to allocating traffic to underlay offerings. RTT is also different and time-varying between these classes – the

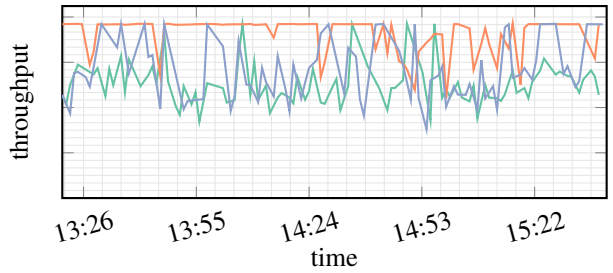


Figure 18: Relative throughput between two VMs in the US. Each line represents a different class of service.

minimum RTT is the same for all three, but the maximum is 1.7× higher for the middle class.

We have removed the units on the vertical axes on the graphs as these results are obtained from proprietary access to underlay paths that are not available to the public. However, the key takeaway is visible, that there are different underlay paths offering different tradeoffs in network performance. As future work, we are considering how to construct hybrid paths across one WAN with only overlay paths and another WAN with access to underlay paths.

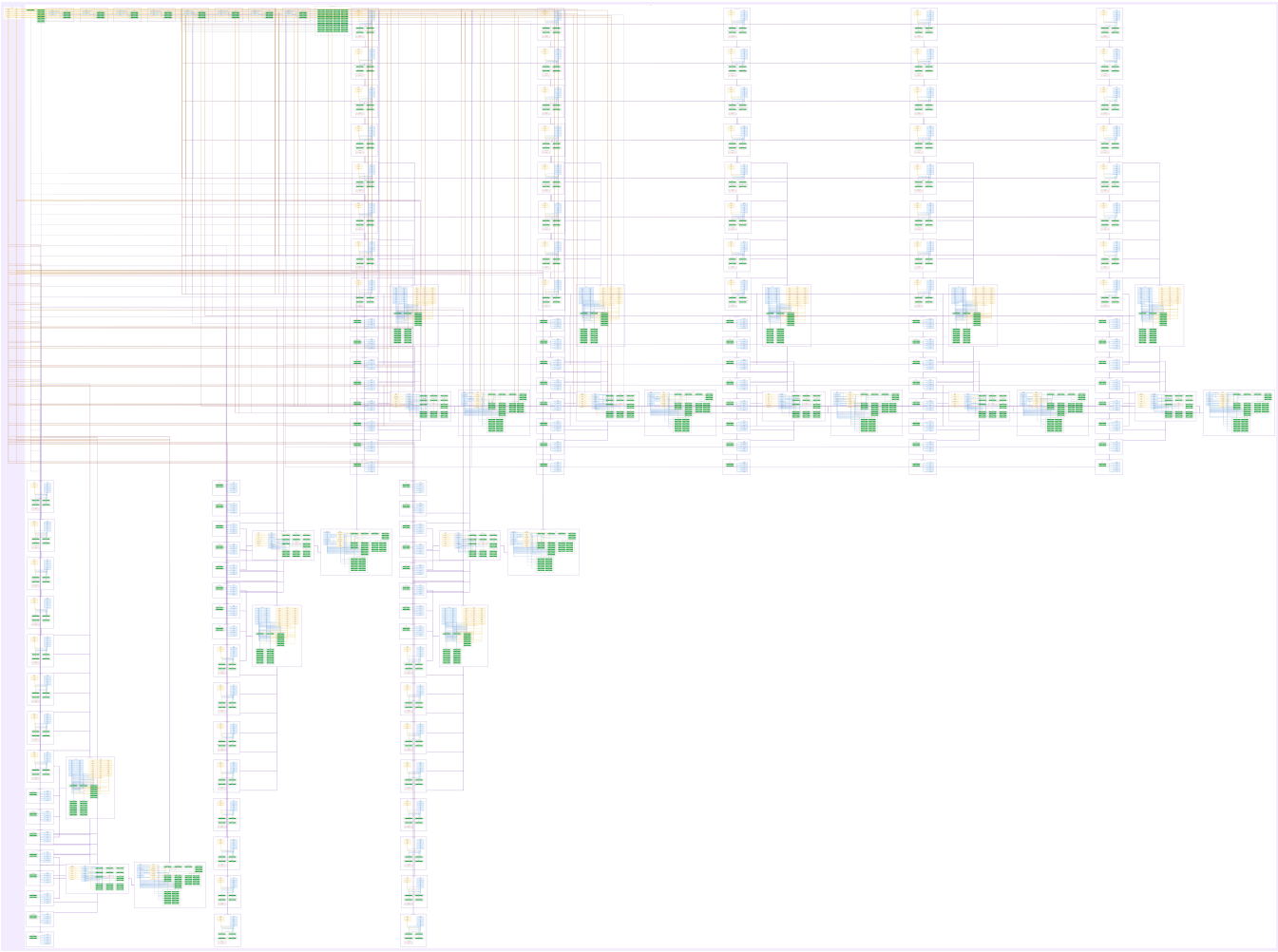


Figure 19: OTTER multi-WAN topology deployed on continental US scale across Azure and GCP.

Verifying maximum link loads in a changing world

Tibor Schneider
ETH Zürich

Stefano Vissicchio
University College London

Laurent Vanbever
ETH Zürich

Abstract

To meet ever more stringent requirements, network operators often need to reason about worst-case link loads. Doing so involves analyzing traffic forwarding after failures and BGP route changes. State-of-the-art systems identify failure scenarios causing congestion, but they ignore route changes.

We present Velo, the first verification system that efficiently finds maximum link loads under failures *and* route changes. The key building block of Velo is its ability to massively reduce the gigantic space of possible route changes thanks to (i) a router-based abstraction for route changes, (ii) a theoretical characterization of scenarios leading to worst-case link loads, and (iii) an approximation of input traffic matrices. We fully implement and extensively evaluate Velo. Velo takes only a few minutes to accurately compute all worst-case link loads in large ISP networks. It thus provides operators with critical support to robustify network configurations, improve network management and take business decisions.

1 Introduction

Link loads—how much traffic crosses each link—are a key indicator of network performance. High link loads, in particular, increase the likelihood of congestion, packet drops, and inflated delays. To meet stringent service-level agreements, operators often need to reason about the worst-case load that every link can realistically experience during network operation – e.g., to check that all loads are below a safety threshold, and adapt routing configurations if they do not.

Identifying worst-case link loads is hard, though. Operators often have tools to measure traffic and model traffic patterns, even long term [26, 35]. However, determining worst-case loads requires to scrutinize traffic forwarding for a *huge range of possible events*. During network operation, links and nodes fail, and BGP routes for external destinations appear, disappear and change. Even a single failure or 3-4 route changes at specific border routers can overload links, as our case study on a real ISP network shows (§7).

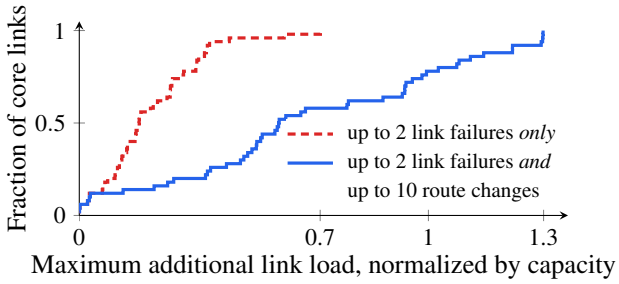


Figure 1: Additional link loads can double due to route changes compared with link failures only. This plot shows the additional traffic on all the core links of an ISP network.

Reasoning about link loads is beyond the capabilities of existing network verifiers. Most of them [1, 4, 14, 39, 40, 42, 45, 50] do not support *performance* requirements such as maximum link loads, but restrict to *functional* requirements, such as the absence of blackholes and forwarding loops.

A couple of recent contributions [7, 27, 44] focus on assessing properties on link loads. Yet, they only consider network failures and assume fixed external routes, fitting controlled network environments such as data centers. For most Internet-connected networks, *both* failures and route changes must be considered, *jointly*, to provide guarantees on link loads. As an illustration, Figure 1 depicts the additional load on the core links of a real ISP network, when simulating all one and two link failures. In the presence of up to ten route changes (solid curve), most links reach higher load than without route changes (dashed curve), and their load can be twice as big. In other words, ignoring route changes leads to vastly underestimating maximum link loads.

This paper presents Velo, the first verification system that efficiently computes the individual worst-case loads of *all* links for arbitrary link failures and external route changes. Velo does not depend on how paths are computed, and hence it works in both traditional (e.g., IGP/BGP-based) and SDN networks; it also supports typical traffic engineering technologies such as ECMP and tunnelling (e.g., MPLS).

In each run, Velo takes as input: router configurations, BGP routes, and a traffic matrix. The traffic matrix captures the specification (such as the worst-case pattern seen in the past, or the expected future traffic) for that run. Different traffic scenarios can be verified in separate runs. As in [44], operators can also input the maximum traffic volume additional to the given matrix, e.g., to model uncertainty. Further, Velo can limit its computations to user-defined subsets of failures and route changes that model realistic scenarios. For example, operators can specify a maximum number of simultaneous route changes, and define per-network and per-router stable destinations, for which route changes are not allowed.

As such, Velo supports many practical use cases including: *(i)* verification of configurations before deployment; *(ii)* adaptation of BGP preferences to make the worst-case routing states less likely; *(iii)* fine-tuning of network monitors to raise alerts only when there is a significant risk of congestion; *(iv)* support for business decisions and routing policies. Our case study (§7) exemplifies some of these use cases.

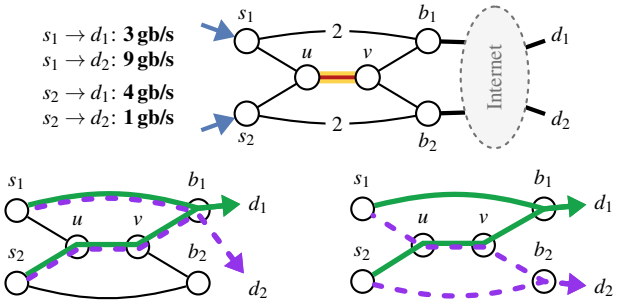
Challenge It is infeasible to iterate over all possible route changes and failure scenarios. For each destination, external networks may advertise new routes, modify existing ones, or completely withdraw them. New and modified routes have attributes (e.g., AS path or BGP communities) with a very large set of possible values. When factoring in possible failures, not only does the search space grow, but it also becomes more challenging to navigate. Indeed, failures generally affect paths to multiple destinations, hence destinations cannot be treated as independent from each other across failures. This also prevents us from extending prior verification work [4, 15, 45, 50].

Solution We greatly reduce this search space in three steps.

First, we provide a compact abstraction for capturing route changes. We do not explicitly model possible external routes, e.g., symbolically as in [4]. Instead, we model possible routes for a destination as *sets of egress routers*. Each set captures the many equivalent routes inducing the same link loads.

Second, we design Velo to explore a *minimal* set of states guaranteeing correctness. We prove that for common intra-domain routing schemes (e.g., shortest-path routing), each link is maximally loaded when only one egress router is used per destination. This enables Velo to compute all the worst-case link loads in *polynomial* time. In the presence of paths deviating from the above schemes (e.g., for traffic engineering), Velo extends its search to *minimal* sets of egress routers, ensuring limited impact on Velo's efficiency.

Third, we propose an efficient technique to approximate the input traffic matrix by combining destinations with similar traffic patterns, while also providing strong accuracy guarantees. We formalize the traffic matrix approximation as a clustering problem, and prove that the clustering error bounds the approximation error of the worst-case link load.



(a) Both destinations d_1 and d_2 are advertised only to b_1.

(b) Destination d_1 is advertised to b_1 and d_2 is advertised to b_2.

Figure 2: Network that routes traffic along the shortest paths. Subfigures (a) and (b) show the forwarding paths for two possible routing advertisements of destinations d_1 and d_2. All link weights are 1, except those from s_i to b_i are 2.

Results We implement a prototype of Velo and make it publicly available[1]. Our evaluation on real network topologies shows that Velo computes the worst-case load of every link in large networks and the entire BGP routing table, *within minutes*, even considering up to two simultaneous link failures. Velo's computed maximum loads are also highly accurate, within 1% of the actual ones, and this across a wide range of traffic matrices, from heavy-tailed to almost-uniform.

2 Overview

Velo finds the worst-case load for each link by exploring the input space of failures and route changes. Consider the example in Figure 2. Routers s_1 and s_2 receive traffic for two remote destinations d_1 and d_2. Depending on the received BGP routes, d_1 and d_2 can be reached via either b_1 alone, b_2 alone, or both simultaneously. Both s_1 and s_2 send packets to the closest egress among b_1 and b_2 that can reach the destination (i.e., implementing hot-potato routing).

Let's initially focus on the link (u,v), assuming no failures. The link load depends on the BGP routes received by b_1 and b_2. For example, if only b_1 receives BGP routes for d_1 and d_2, packets from s_2 crosses (u,v) but those from s_1 do not, resulting in a total load of 5 gb/s—see Figure 2a.

To compute the worst-case link load, we conceptually need to check all the combinations of each destination being reachable from b_1, b_2, or both. For example, if b_2 receives the same BGP route as b_1 for d_1 and still no route for d_2, the load of (u,v) decreases, because the traffic from s_2 to d_1 is moved to the path (s_2,b_2). Figure 2b shows the worst-case scenario: when d_1 is only reachable from b_1 and d_2 only from b_2, then 13 gb/s are sent over (u,v) since both demands $s_1 \to d_2$ and $s_2 \to d_1$ cross that link.

[1] Available at https://github.com/nsg-ethz/velo (GPLv3 license)

Consider now any other link, for example, (s_2, b_2). Clearly, Figure 2b is not the worst-case scenario for (s_2, b_2); its load is higher if s_2 uses b_2 as next hop for both d_1 and d_2.

Real networks are much bigger than Figure 2, and typically have many more border routers and millions of destinations, making the problem challenging to solve. For example, iterating over all the possible BGP routes received by any border router for each destination does not scale. Worse yet, every possible failure affects the network topology, and so requires to re-assess the impact of each possible route change.

2.1 Problem Statement

We now describe Velo's problem and provide the intuition of how Velo manages to accurately and efficiently solve it. We refer to any router that sends traffic outside the network as *egress router*. For example, in Figure 2b, b_1 is the only egress router for d_1, and b_2 is the only egress router for d_2. An *egress change* occurs when the set of egress routers for one destination changes, such as between Figure 2a and Figure 2b.

Problem: Given *(i)* the network configuration, *(ii)* current BGP routes, *(iii)* traffic per destination, *(iv)* constraints on route changes, and *(v)* a space of failure scenarios, find the worst-case load for each link in the network, along with the egress changes and the failure scenario causing such loads.

We provide more details on Velo's inputs in the following.

Network configuration We define a network configuration as the network topology and the collection of all router configurations. We support a wide range of networking paradigms and protocols and present an abstraction in §3. Operators can provide either the *current* topology and router configurations as input, or alternative ones (e.g., for what-if analyses).

BGP routes Velo takes the current BGP routes as input. We define a BGP route as a BGP advertisement from an external network concerning a single destination and including its BGP attributes. Routes generally change over time.

Yet, not all route changes are equally likely, and some are not worth considering in practice. For example, at operational timescales, receiving new routes for a few destinations is more likely than the same occurring for all Internet prefixes.

For this reason, Velo allows to limit the set of route changes verified in any Velo's run. Operators can specify the maximum number k of egress changes, where the *number of egress changes* is the number of destinations featuring one or more egress changes. Operators can also provide constraints on specific destinations (e.g., high-volume IP prefixes) for which routes can or cannot change, and on the possible egress routers per destination (e.g., enforcing that BGP customers always advertise only their own prefixes).

Failure scenarios Contrary to route changes, network failures typically affect many forwarding paths for many (if not all) destinations. Velo takes as input a description of the space of failure scenarios to explore. This can be the set of up to l simultaneous link or node failures [42, 44], and can include Shared Risk Link Group (SRLG) information [53]. The only constraint here is that for each failure scenario, Velo can derive a new topology from the original one.

Traffic In each run, Velo takes a traffic matrix as input. Previous work often defines a traffic matrix as the traffic volumes per pair of ingress and egress routers, describing where traffic enters and exits the network. We instead define a traffic matrix as traffic volumes for each pair of ingress router and destination prefix. We do so because we are interested in assessing the impact of BGP routes which are per-destination.

While Velo accepts a single matrix as input, operators can specify the current traffic matrix, or, for more conservative analyses, they can provide Velo with a matrix derived from one or multiple congestion events recorded in the past. They can also run Velo multiple times on different matrices.

Velo models uncertainty on inter-domain traffic in the same way as QARC [44]. It indeed allows operators to specify the total amount of traffic, across all the destinations, that may need to be forwarded in addition to the input traffic matrix.

2.2 Velo in a nutshell

Verifying link loads requires exploring a gigantic search space. Besides the sheer number of routes and route attributes, link failures prevent destinations from being verified independently (i.e., one at the time). Velo addresses these challenges by massively reducing both the search space and the number of destinations. We now provide the intuition of how it does so, referring to the following sections for details.

Reducing the search space In §4, we prove that the worst-case link loads occur when for each destination, all the network routers select the same BGP route. Indeed, the worst-case load for (u, v) in Figure 2 occurs when all routers select b_1's route for d_1, and b_2's route for d_2. This key insight enables us to explore, for each failure scenario, only $|P| \cdot |N_B|$ different states, where P is the set of destinations and N_B is the set of egress routers—a polynomial search space instead of an exponential one.

The above property holds for all strictly isotone routing protocols (such as shortest path routing) that usually govern intra-domain routing. We further extend our technique to support exceptions to strict isotonicity, such as MPLS tunnels for traffic engineering, by computing the minimal set of additional states to explore in these cases (§4.1.2).

Velo iterates over all the input failure scenarios, one by one. We experimentally show (§6) that Velo is anyway more efficient than approaches attempting to prune failure scenarios, like QARC, even when considering combinations of up to 3 or 4 simultaneous failures. Intuitively, this happens because our search space reduction enables Velo to analyze any failure scenario *very* quickly.

Reducing the number of destinations In §5, we show that we can combine destinations with *similar* traffic distributions across the same ingress routers, and obtain an approximated traffic matrix that contains much fewer destinations. This approximation significantly improves Velo's running time at the expense of potentially degrading its accuracy. Yet, we present a modified k-means clustering algorithm that computes an approximated traffic matrix with provable bounds on the accuracy loss. In §6.2, we experimentally show that the approximation is highly accurate for both realistic (heavy-tailed) and unrealistic (almost-uniform) traffic matrices. We show that this clustering technique is more efficient and general than a simple approach based on heavy-hitters, as it achieves tighter error bounds by considering fewer destinations for all the traffic matrices used in our experiments.

To illustrate our approximation, consider a third destination d_3 in Figure 2 with traffic $s_1 \to d_3$ of 0.4 gb/s and $s_2 \to d_3$ of 0.3 gb/s. d_3 is also reachable via b_1, b_2, or both. The traffic for d_1 (3 gb/s from s_1 and 4 gb/s from s_2) is ten times larger than for d_3, but their distributions are similar. Considering d_1 and d_3 jointly would yield a maximum load on link (u,v) of 13.3 gb/s instead of 13.4 gb/s when analyzing d_1 and d_3 individually. Velo accurately bounds the error to 0.1 gb/s.

Dealing with traffic variability Velo's input can include an amount y of traffic additional to the input traffic matrix. There are infinite traffic matrices compatible with any given y; obviously, we cannot analyze all of them. Two key observations instead inspire Velo's approach. First, allowing y additional traffic increases the worst-case load of any link by up to y compared to its maximum load for the input traffic matrix. Second, for any link, its maximum load increases exactly by y whenever a router r sends *all* its traffic to a destination d across that link: in this case, indeed, increasing the traffic from r to d by y increases the load on that link by y.

Based on these observations, Velo accounts for y additional traffic by first computing the maximum link loads for the input traffic matrix, and then increasing the maximum link loads by y. This approach may overestimate the maximum load of a link if no router forwards all its traffic for any single destination over that link in the worst-case scenario for that destination. This tends to occur rarely in practice (less than 0.1% in our experiments in §6), implying that Velo is fully accurate in the vast majority of scenarios.

3 Model and Notation

We model the network as a graph $G = (N, E)$ with routers N connected by edges E. Border routers $N_B \subset N$ are connected to external networks that advertise BGP routes to destinations P, that is, remote IP prefixes populating the routing tables of the routers. For each destination, the network selects a set of *preferred routes*, corresponding to a set $B \subseteq N_B$ of border routers that will be used as *egress routers* (i.e., to forward traffic towards that destination). This selection can be done by a central controller [10] or in a distributed fashion using iBGP [37]. In the case of iBGP, we assume that *(i)* BGP attributes that affect routes' preference are not modified on any iBGP session, and *(ii)* all routers eventually learn their preferred routes. Several techniques are available to avoid iBGP visibility problems [36, 47, 48] or to prove the absence thereof [4, 16, 45, 49].

By focusing on egress routers per destination, we hide most of BGP's low-level details (e.g., its decision process). For any given destination d, we must only consider *(i)* the border routers that *can* receive a route for d, and *(ii)* the possible subsets of border routers that can receive *preferred* routes for d. For the former, we check whether configured eBGP ingress route maps discard all routes for d. For the latter, we analyze the attributes modified by ingress route maps, mainly the Local Preference, that affect the BGP decision process. Other attributes like the AS path can take arbitrary values, including those that result in a tie during route selection.

We denote the input traffic matrix as $\mathbf{M}$. Each element $\mathbf{M}_{s,d}$ is the ingress traffic that enters the network at router $s \in N$ with destination $d \in P$. We define a column $\mathbf{M}_d$ of $\mathbf{M}$ as the total traffic for destination d, as sourced from all ingress routers in N. The network sends traffic $\mathbf{M}_d$ for d over the computed paths for egress routers in B, resulting in load $load(e, \mathbf{M}_d, B)$ on any link $e \in E$.

Intra-domain routing Forwarding paths are computed from each router to its selected egress router. Many intra-domain routing approaches exist. Some rely on a central controller [18, 20], while others use distributed Interior Gateway Protocols (IGPs) [33, 34]. Static routes, source routing or tunnelling can also be used to forward traffic [26].

We model the intra-domain path computation using routing algebra [41], consisting of weights W, a binary operation $\oplus$ and an order relation $\succcurlyeq$. For example, the commonly used shortest-path-based IGPs [33, 34] are modeled with weights corresponding to per-link IGP costs, a binary operation that adds costs, and a binary relation that selects the shorter path.

We write link weights as $w : E \mapsto W$. We denote the optimal s-t path in the routing algebra as $p^*_{s,t} = (s, n_1, \ldots, n_i, t)$ with path weight $w_{s,t} = w(s,n_1) \oplus \cdots \oplus w(n_i,t)$. We assume that traffic is split equally across all next-hops of each router, i.e., the first hops of its optimal paths. This is consistent with traffic distribution in the presence of ECMP [26].

Link weights W are *strictly isotone* if $a \succ b$ implies both $a \oplus c \succ b \oplus c$ and $c \oplus a \succ c \oplus b$ for any $a, b, c \in W$. This property is both necessary and sufficient to ensure that any sub-path of an optimal path is itself optimal, allowing the paths to be computed using a generalized Dijkstra algorithm (if they exist) [41]. Note that strict isotonicity is common in intra-domain routing regardless of the forwarding paradigm (e.g., hop-by-hop vs. source routing vs. SDN).

Strictly isotone weights offer limited flexibility to achieve traffic engineering [13], so operators sometimes configure *exception paths* (e.g., MPLS RSVP-TE tunnels [3]) that carry packets over manually defined sequences of edges. We denote the set of exception paths as ρ that take precedence over shortest paths.

4 Finding Maximum Link Loads

We achieve scalability thanks to a highly efficient algorithm for finding the worst-case link load in any input topology, allowing us to iterate over a large number of failure scenarios. For any given topology, the worst-case load of a link is the sum of its worst-case loads for each destination considered independently: this is because BGP routes are specified per destination, and so are forwarding paths. To find the worst-case link loads in each topology, we thus search for the worst-case egress routers per destination, one destination at a time. Hereafter, we explain how Velo performs such a search for a given topology and a single destination.

4.1 Per-Destination Worst-Case Link Loads

The basic building block of our approach is to efficiently find the set of egress routers for a single destination that maximizes the load of a specific link. Reasoning on the worst-case egress routers per destination compared to the huge and possibly infinite space or BGP routes and their attributes is already a massive optimization. Yet, naively exploring the space of egress routers still requires exploring $O(2^{|N_B|})$ states.

In the following, we explain how we reduce our search space to $O(|N_B|)$ states whenever internal paths are computed from strictly isotone weights (§4.1.1), and slightly more states in the presence of a few exception paths (§4.1.2).

4.1.1 Strictly Isotonic Routing

Let $G = (N, E)$ be a network routing traffic matrix $\mathbf{M}$ according to protocol $(W, \oplus, \succcurlyeq)$ where W is strictly isotone. We focus on one destination d and the maximal load on link e.

Theorem 1 *For any non-empty set of egress routers $B \subseteq N_B$ and any link $e \in E$, there exists a single egress $b \in B$ that, if chosen as the unique egress for destination d, causes at least the same load on e as the set B. Formally:*

$$\forall e \in E : \exists b \in B \text{ s.t. } load(e, \mathbf{M}_d, B) \leq load(e, \mathbf{M}_d, \{b\})$$

Algorithm 1: Find the worst-case link load for all links simultaneously if all traffic is forwarded along optimal paths.

Data: Graph $G = (N, E)$, border routers $N_B \subseteq N$, link weights $w : E \mapsto \mathbb{R}$, traffic $\mathbf{M}_d$
Result: maximum link loads $y[e]$ for all links $e \in E$

$y[u, v] \leftarrow 0 \; \forall (u, v) \in E$
for $b \in N_B$ **do**
  $l[s] \leftarrow \mathbf{M}_{s,d} \; \forall s \in N$
  $G_{dag} \leftarrow ForwardingDAG(G, w, root = b)$
  for $u \in TopoSort(G_{dag})$ **do**
    $d_{out} \leftarrow |out(G_{dag}, u)|$
    for $v \in out(G_{dag}, u)$ **do**
      $l[v] \leftarrow l[v] + l[u]/d_{out}$
      $y[u, v] \leftarrow \max(y[u, v], l[u]/d_{out})$

Intuitively, Theorem 1 holds by the main property of strict isotonic routing protocols; the sub-path of any optimal path is also optimal. All traffic crossing the edge $e = (u, v)$ must follow a path that is also optimal for v, allowing us to reduce the set of egresses to the one selected by v. In case a router splits its traffic equally among multiple paths, we show that the fraction of traffic it sends across the edge e will not decrease.

Proof of Theorem 1. Consider any edge $e = (u, v)$, and let $N_e \subseteq N$ be the set of all nodes that forward traffic along e. Further, let b be an egress router preferred by v, and pick any router $n \in N_e$.

First, observe that the weight of n's optimal path towards any egress router in B, namely $w_{n,B}$, is equal to the weight of its path to b via e, i.e., $w_{n,b} = w_{n,u} \oplus w(e) \oplus w_{v,b}$. Hence, $w_{n,B} = w_{n,b}$ due to the strict isotonicity.

We now show that the total amount of traffic n sends across edge e cannot decrease when only egress b is available. We show this by comparing n's next-hops for egresses B, i.e., $nh_{n,B}$, with those for the single egress b, i.e., $nh_{n,b}$. First, we argue that $nh_{n,b} \subseteq nh_{n,B}$, and thus, the traffic n sends to any next-hop in $nh_{n,b}$ does not decrease. This follows from the fact that any optimal path of n for b is also optimal for B.

Second, we show that $nh_{n,b} \cap N_e = nh_{n,B} \cap N_e$, proving that the fraction of traffic n sends across the edge e cannot decrease. Consider any next-hop $x \in nh_{n,B} \cap N_e$: n can still reach the destination d via egress b on an optimal path via x. This is because $x \in N_e$, and thus, $w_{x,B} = w_{x,b}$. □

Theorem 1 implies that the worst-case load of any link for any destination is obtained when the entire network chooses a single egress router. More formally,

$$\max_b load(n, \mathbf{M}_d, \{b\}) = \max_B load(n, \mathbf{M}_d, B).$$

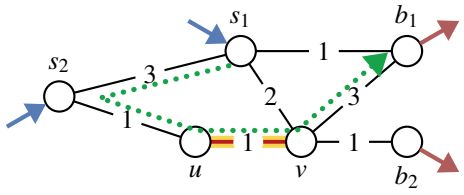


Figure 3: Example where considering less combinations of egress routers than Velo in the presence of exception paths leads to underestimating maximum link loads.

This observation directly enables us to reduce the number of explored states to $O(|N_B|)$. Indeed, Velo iterates over the border routers in N_B. For each border router b, Velo creates the forwarding graph rooted at b, which is a directed acyclic graph (DAG). It then computes the corresponding link loads by traversing the DAG in topological order and pushing traffic from each node to its outgoing edges. At the end of the iteration, it reports the worst-case scenario for each link.

Algorithm 1 shows our pseudo-code. Note, that the second-to-last line ensures traffic is split equally among next-hops, i.e., implementing ECMP. Its computational complexity is $O(|N|^2 \log|N| + |N||E||P|)$, because the two most expensive operations are (i) computing $|N_B|$ forwarding DAGs with Dijkstra's algorithm [41], and (ii) traversing a DAG in topological order for each destination prefix $d \in P$.

4.1.2 Exception Paths for Traffic Engineering

Theorem 1 does not hold in the presence of exception paths. Take the example depicted in Figure 3, with two egresses b_1 and b_2, and two ingress routers s_1 and s_2. Shortest-path routing is always used, except s_1 forwards traffic for b_1 on the path (s_1, s_2, u, v, b_1), drawn as a dotted line. The load on link (u,v) is maximized when both b_1 and b_2 are egress routers. Indeed, shortest paths are such that s_2 bypasses (u,v) if b_1 is the only egress, and s_1 bypasses (u,v) if b_2 is the only egress.

To deal with exception paths for destination d, we explore all single egress routers plus all the combinations of the border routers terminating any exception paths configured for d. The example in Figure 3 demonstrates that considering less egress routers can lead to incorrect results. We now show that considering only these combinations is sufficient to find the worst-case link loads. Let ρ_d be the exception paths configured for d, and let $T = \{b \mid (s, \cdots, b) \in \rho_d\}$ be the set of egress routers ending any path in ρ_d.

Theorem 2 *For any non-empty set of egress routers $B \subseteq N$ and any link $e \in E$, there exists a subset $B' = \{b\} \cup T'$, where $T' = T \cap B$, that causes at least the same load on e than the set B. Formally, for every link $e \in E$,*

$$\exists b \in B \text{ s.t. } load(e, \mathbf{M}_d, B) \leq load(e, \mathbf{M}_d, \{b\} \cup T \cap B)$$

Intuitively, Theorem 2 holds as nodes either forward via optimal paths, in which case Theorem 1 applies, or via exceptional path that are still available with the reduced subset of egress routers. We provide a proof in Appendix A.1.

Theorem 2 allows us to find the worst-case link loads by iterating over all subsets $T' \subseteq T$ and one other egress router in $N_B \setminus T$. We filter out combinations of egresses that cannot receive equally preferred routes due to BGP policies.

For each set B of egress routers computed as above, we construct the corresponding forwarding DAG by combining the forwarding DAGs rooted at all egresses in B. We then find the load resulting from traffic routed along exception paths, and push the remaining traffic through the DAG as in Algorithm 1. We provide a pseudo-code in Appendix A.2.

Computing link loads in the presence of exception paths has complexity $O(|N|^2 \log|N| + 2^{|T|}|N||E||P|)$, similarly to the complexity of Algorithm 1 but for the larger search space.

4.2 Restricting the Number of Route Changes

So far, we have discussed finding the worst-case link loads considering any set of route changes. However, part of Velo's input is the maximum number k of egress changes. Hence, for each link, we must find the k destinations that cause the highest increase of its load. To that end, we maintain for each link a heap of size k with the largest difference between the worst-case and current-state link loads. Maintaining the heap has a time complexity of $O(|E| \cdot |P| \cdot \log k)$ which is negligible with respect to the total time of the algorithm.

5 Approximating the Traffic Matrix

The algorithms presented in §4 scale linearly in the number of destinations. It is not rare that real routers have order of a million entries [19, 26], so finding the worst-case link load for so many destinations would be a limiting factor.

We greatly reduce the number of destinations by combining those with similar traffic patterns. Specifically, we approximate the input traffic matrix with a smaller one, leveraging the low effective rank that traffic matrices typically exhibit [6, 31].

We formulate this traffic matrix approximation problem as a clustering problem, and find that the clustering error bounds the *approximation error* δ; by how much the link loads differ when computed on the original and approximated traffic matrices. We envision that our approximation technique can be useful within other networking problems for which the size of traffic matrices limits scalability.

Approximation problem Given a traffic matrix $\mathbf{M} : |N| \times |P|$, we aim to find a clustering C of the destinations that reduces the size of $\mathbf{M}$ while also bounding the error induced by that reduction. More formally, we aim to compute a partition C of P, an approximate traffic matrix $\mathbf{A}$ of size $|N| \times |C|$, and

an *error bound* ε for the approximation error δ, such that

$$\delta = \max_{e \in E} |max\,load(e, \mathbf{M}) - max\,load(e, \mathbf{A})| \leq \varepsilon, \quad (1)$$

where $max\,load(e, \mathbf{M})$ is the maximum load of link e for $\mathbf{M}$.

We take the number of clusters $|C|$ as input. Increasing $|C|$ reduce both ε and δ at the cost of running time. Operators can tune $|C|$ to find a balance between accuracy and running time.

We must however be careful to cluster together only destinations for which the routing and forwarding states are equivalent. For example, we must not aggregate destinations with different intra-domain paths for the same egress point. To this end, we provide a custom definition of *Equivalence Class*, and only cluster destinations in the same equivalence class. We distribute the available clusters $|C|$ across all equivalence classes proportional to the amount of traffic they carry. Specifically, any pair of destinations in an equivalence class must satisfy the following two conditions:

1. The same egress routers are used for both destinations in the current routing state.
2. The forwarding paths for both destinations are always equal for the same set of preferred egress routers.

In the following, we detail our clustering algorithm for destinations in the same equivalence class.

Normalized clustering algorithm We cluster together destinations with similar traffic distribution across ingress routers, i.e., if similar fractions of traffic for two destinations enter from the same routers. Such destinations *likely* produce worst-case link loads for the same routing inputs.

We define the traffic distribution $\mathbf{M}_d/|\mathbf{M}_d|$ of destination d as the fraction of traffic from each ingress compared to d's total traffic load $|\mathbf{M}_d|$. We then use the L1 norm [24] to measure the similarity of two distributions, as the L1 norm quantifies their absolute difference. We run an *adaptation* of the k-means clustering algorithm on the traffic distributions: instead of minimizing the L1 distances within a cluster c_i, we weighten the distance between its cluster centroid $\mathbf{A}_i$ and a destination d according to d's total traffic. Thus, the computed clusters accurately resemble high-traffic destinations. Cluster centroids $\mathbf{A}_i$ and the clustering error ε are defined as follows.

$$\mathbf{A}_i = \sum_{d \in c_i} \mathbf{M}_d \qquad \varepsilon = \frac{1}{2} \sum_{c_i \in C} \sum_{d \in c_i} |\mathbf{M}_d| \cdot \left| \frac{\mathbf{M}_d}{|\mathbf{M}_d|} - \frac{\mathbf{A}_i}{|\mathbf{A}_i|} \right|. \quad (2)$$

Finally, Velo constructs the approximate traffic matrix $\mathbf{A} = [\mathbf{A}_1, \mathbf{A}_2, \cdots]$ directly from the cluster centroids.

Our clustering approach guarantees that the approximation error δ is bounded by the clustering error ε. Intuitively, this is because *(i)* ε is the amount of traffic in $\mathbf{M}$ that the approximation $\mathbf{A}$ does not account for, and *(ii)* ε is also the amount of additional traffic in $\mathbf{A}$ that is not part of the original matrix $\mathbf{M}$. Consequently, $max\,load(e, \mathbf{A})$ can differ from $max\,load(e, \mathbf{M})$ by at most ε. We provide a formal proof in Appendix B.

6 Evaluation

We now evaluate Velo's performance in terms of scalability (running time) and accuracy. Its correctness is instead guaranteed by our theorems in §4 and §5.

In §6.1, we discuss aspects that influence Velo's running time. We show that Velo can find the maximum loads for all the links in large networks (with close to 2 000 links) within a few hours, considering up to two simultaneous failures and up to ten changes of any external routes. As baseline, we also compare Velo against its closest related work, QARC [44], in scenarios where external routes do not change. We find that Velo runs much faster than QARC for scenarios considering up to two simultaneous failures (the most practical ones).

In §6.2, we evaluate the impact of the traffic matrix on Velo's accuracy. We show that Velo's approximation error is less than 1% even for unrealistic traffic matrices that are hard to approximate.

Implementation We implement a prototype of Velo in $\approx$ 7 000 lines of Rust,[2] including an optimized version of both Algorithms 1 and 2, and our modified k-means clustering algorithm. We run the evaluation on a server with 96 CPU threads and 384 GB of memory.

Networks We study the 75 largest networks from Topology-Zoo [23], ranging from 40 to 754 nodes with 80 to 1 790 links. For each network, we study the effect of increasing the number of border routers by randomly connecting external peers that can advertise a route for any destination to internal routers. We assign random IGP weights (between 10 and 100) to each link. To study the impact of non-shortest paths routing, we setup exception paths (i.e., traffic-engineered paths that do not follow the shortest paths) for given destinations by computing alternative paths using permuted link weights.

Traffic matrices We obtain real traffic matrices from a research network. We also generate synthetic traffic matrices following the gravity model [9, 31]. This enables us to control and vary traffic characteristics to evaluate Velo's sensitivity to different traffic patterns.

6.1 Running Time

The following factors affect Velo's running time: network size, number of simultaneous failures l, maximum number of external route changes k, number of clusters $|C|$, number of border routers $|N_B|$, and number of exception paths $|\rho|$. The specification of traffic additional to the traffic matrix instead has no noticeable impact, consistently with the inexpensive approach to deal with it in Velo (see §2).

[2] Available at https://github.com/nsg-ethz/velo (GPLv3 license).

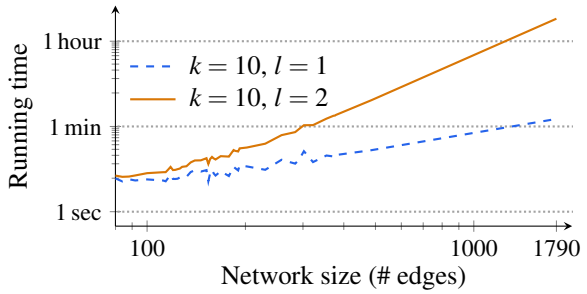


Figure 4: Velo can scale to large networks of over 1 000 links while exploring up to two simultaneous link failures. This log-log plot shows the running time of Velo for the 75 largest topologies in TopologyZoo with up to one and two simultaneous link failures l running on 96 parallel threads.

We find that the network size and the number of failures have the biggest impact on Velo's running time. We therefore divide our experiments in two sets: a fist set considering only these two factors, and another set considering the others.

Following these experiments, we compare Velo's running time against QARC [44], a state-of-the-art approach to find link load violations under varying scenarios, albeit assuming fixed external routes.

Network size and link failures In the first set of experiments, we analyze the impact of network size and maximum number of failures on Velo's efficiency across all the topologies in our dataset. We set a relatively low number of simultaneous failures ($l \leq 2$) and route changes ($k = 10$), as it is unlikely for many more links to fail and many more external routes to change at the same time [42]. We also set $|C|$ to 300, which empirically provides a good accuracy-performance tradeoff (see Appendix D.3). We finally configure 30 border routers per topology, and no exception path.

Figure 4 shows a log-log plot of Velo's running time. Varying the network size from 100 to 1 790 increases the running time by about four orders of magnitude. A similar effect is caused by considering all combinations of two link failures ($l = 2$) rather than single link failures ($l = 1$).

Yet, our results show that Velo is practical for all the TopologyZoo networks. For the largest one, Velo finds the worst-case load for all its 1 790 links within one minute for single link failures, and within three hours for two link failures. For all the other networks, Velo finds all the worst-case loads for $l \leq 2$ within two minutes.

We further break down Velo's efficiency according to its two main phases: clustering and analysis. The clustering phase dominates the running time for small networks (up to 200 edges), while the analysis phase is the dominant one for bigger networks. The asymptotic growth of the analysis time, and hence of the overall running time, is roughly cubic for $l = 2$. Full results are shown in Appendix D.2.

Finally, we approximate the running time improvement gained from reducing the search space of route changes. With 30 border routers, the naive approach requires exploring 2^{30} states—seven orders of magnitudes bigger than Velo's search space. Analyzing a network of only 80 links without the state reduction would take roughly one year for $l = 2$.

Other factors To measure the impact of the other factors, we focus on Cogent, one of the largest networks in Topology Zoo, and vary each factor, one at a time.

The number of route changes k has a small effect on Velo's efficiency. This is because for each link, we always need to compute the worst-case routing input for *all* destinations, before picking the k destinations causing the highest load on that link. Increasing k from 10 to 100 still increases the running by 1.3× but only because the algorithm must maintain larger heaps. We however note that allowing all routing inputs to change (i.e., $k = |P|$) *reduces* the running time by 2.9×, as Velo needs to maintain less state.

Increasing the number of clusters $|C|$ affects the running time, as Algorithms 1 or 2 are called for each cluster. Increasing $|C|$ from 300 to 600 or 1 000 increases the running time by 1.3× or 1.6×, respectively. Similarly, decreasing $|C|$ to 100 improves Velo's performance 1.2×. As a reference, deactivating clustering would take Velo roughly 1 000× longer to complete, assuming around 1 million destinations.

The number of border routers $|N_B|$ affects the number of states to be explored. Yet, increasing $|N_B|$ from 30 to 100 only increases Velo's running time by 1.7×.

Finally, adding exception paths forces Velo to explore additional states. Indeed, Velo must explore all combinations of distinct egress routers of these paths. Such paths are usually configured manually, and they are often used only for traffic-heavy destinations. A total of 30 or 50 exception paths increases the running time by respectively 1.4× or 3.2×.

Comparison to QARC QARC does not support changes in routing inputs. We thus compare Velo against QARC when both systems are asked to analyze link loads under l link failures. We rerun the same experiments as in [44] with the author's implementation. These experiments involves the specification of traffic between all pairs of routers, and 10% additional traffic. For Velo, we extract the traffic matrix by attaching one destination per egress router, and set $k = 0$: this effectively disables Velo's clustering algorithm, since no destinations fall in the same equivalence class (see §5).

The results for up to two simultaneous link failures are shown in Figure 5. Velo is several orders of magnitude faster in computing maximum link loads than QARC, achieving sub-second computation on all topologies. We believe that such a performance gap is mainly due to the different approaches at the core of the two systems: QARC solves an Integer Linear Program, while Velo relies on Dijkstra's algorithm.

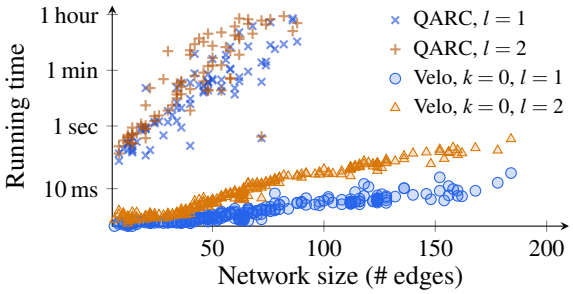


Figure 5: For small $l < 3$, Velo outperforms QARC when comparing only the current state, i.e., with $k = 0$. This figure compares the running time of Velo and QARC [44] to find link load violations for networks from TopologyZoo.

Velo is 10-100× faster than QARC even for $l = 3$ and $l = 4$ (see Figure 7). However, the performance gap between the two systems seems to progressively decrease when increasing l. We expect QARC to outperform Velo for large values of l.

In fairness, QARC guarantees exact computation of link loads, while Velo theoretically doesn't. However, in the absence of clustering, Velo is fully accurate, except possibly overapproximating additional traffic (see §2.2). This is a rare case that does not occur in these scenarios—Velo has *no approximation error* in any of these experiments.

Takeaways Velo finds the worst-case loads of all links within minutes for the most likely and practical scenarios with up to two link failures and a limited number of route changes. Because it is mostly polynomial in the network size, Velo scales to large networks with close to 2 000 links.

6.2 Accuracy

Velo's accuracy depends on the input traffic matrix, which indeed impacts the effectiveness of our clustering. We evaluate Velo's accuracy on both real inter-domain traffic data and synthetic traffic matrices, using the latter ones to measure the effects of traffic characteristics on our clustering algorithm.

We measure both the error bound ε as defined in Equation (2) and the approximation error δ from Equation (1), i.e., maximum difference in worst-case link loads between the approximate and the original traffic matrix.

We further compare our clustering algorithm with the simpler approach of picking the destinations carrying the most traffic, in the following called *Top-X*. We determine how many such destinations are needed to provide the same guarantees as Velo. That is, we find the smallest X such that the amount of traffic disregarded by Top-X equals our error bound ε. We report the ratio between this smallest X and $|C|$: such ratio relates directly to Velo's running time, given that its efficiency scales linearly with the analyzed destinations.

We represent real and synthetic traffic matrices according to the gravity model, a common technique in network traffic synthesis [38]. In the gravity model, a traffic matrix is parametrized by three distributions:

$$\mathbf{M}_{s,d} \propto \mathit{repulsion}(s) \cdot \mathit{attraction}(d) \cdot \mathit{friction}(s,d).$$

The *attraction* describes the distribution of the traffic towards the destinations: a heavy-tailed attraction results in a few destinations attracting most traffic. The *repulsion* describes the distribution of traffic originating from each ingress, symmetrically to the attraction. The *friction* describes traffic similarity across source-destination pairs: a heavily tailed friction yields a sparse matrix, where many entries are zero.

Related work has shown that all the three distributions are heavy tailed in practice [31]. Our real measurements confirm this: traffic per destination (i.e., the attraction), traffic per source (i.e., the repulsion), and traffic similarity across destinations (i.e., friction) are similar to LogNormal distributions. We thus characterize traffic matrices according to the σ parameter of attraction, repulsion and friction: σ indeed measures the skeweness of LogNormal distributions. We provide more details in Appendix C.1.

In all experiments, we pick $|C| = 300$ clusters, as we find this to be a good compromise between accuracy and running time, but we evaluate different values of $|C|$ in Appendix D.3.

Real traffic data We obtain Netflow data for four 5-minute slices from the Swiss research network Switch. These measurements include byte counters per ingress link and per destination IP, aggregated by /24 IPv4 or /48 IPv6 prefixes. The data is extremely skewed; 95% of all traffic is towards 0.23% of all destinations. We stress that this does not seem to be common in the Internet [11, 31]. For example, the top 15% destinations are reported to account for 95% of the traffic in the real traffic matrices studied in [31]. This difference might be an artifact of the destination granularity (i.e., we cannot reconstruct traffic for prefixes less specific than /24), or due to the specificities of the research network originating our data.

Both addresses and ingress links are anonymized, forcing us to randomly assign links to routers. We analyze 100 such random assignments to confirm that this has little impact on the results. We refer to Appendix C.2 for more details.

The first four rows in Table 1 show the results for all such traffic matrices. Velo achieves an error bound ε of around 5% and an approximation error δ of around 0.5%. Appendix D.4 expands on the relationship between ε and δ. Further, Velo is ≈ 3.4× more efficient that top-X despite the extreme skeweness of the data; a property that favors Top-X.

Synthetic data Based on our real traffic matrices, we synthesize matrices with similar characteristics using the gravity model with 30 equivalence classes, cf. §5. Starting from the average parameters of the real matrices, we evaluate how

real	repulsion	attraction	friction	error bound ε (0 … 10%)	approx. error δ (0 … 1%)	efficiency w.r.t. Top-X (1 … 60×)
yes	**2.79**	**2.99**	**3.99**	5.63%	0.68%	4.3×
yes	**2.82**	**2.93**	**3.96**	5.34%	0.68%	3.7×
yes	**2.79**	**3.05**	**4.03**	5.07%	0.49%	4.2×
yes	**3.16**	**2.91**	**4.21**	3.46%	0.44%	3.7×
no	**2.03**	3.01	4.02	11.04%	1.06%	10.8×
no	**2.53**	3.01	4.03	8.79%	0.82%	15×
no	**3.00**	3.00	4.01	7.34%	0.88%	24×
no	**3.47**	2.99	4.01	6.47%	0.77%	40.4×
no	2.97	**1.98**	4.02	9.1%	0.89%	63.7×
no	2.98	**2.50**	4.01	8.52%	0.88%	39.1×
no	3.00	**3.00**	4.01	7.34%	0.88%	24×
no	3.02	**3.51**	4.01	6.01%	0.72%	14.8×
no	3.04	**4.01**	4.01	4.58%	0.52%	8.9×
no	2.95	3.00	**2.51**	12.18%	0.88%	48.6×
no	2.96	3.01	**3.01**	11.07%	0.91%	36.8×
no	2.97	3.00	**3.50**	9.43%	0.85%	30.4×
no	3.00	3.00	**4.01**	7.34%	0.88%	24×
no	3.01	3.00	**4.51**	5.85%	0.67%	18.9×

Table 1: Velo's accuracy is affected by the tail distribution over per-destination traffic, i.e., the attraction, and the variation between traffic distributions, i.e., the friction. This table shows both the error bound ε and the approximation error δ, the latter on a smaller scale. The box represents the 25 and 75 percentiles, with the mean indicated by the red line. The whiskers show the minimum and maximum values. The number right to the upper whisker show the maximum number.

changing the skewness in all three dimensions affects Velo's accuracy, one at a time. We do so for six large topologies from TopologyZoo (with 140–200 nodes). For each topology and set of parameters, we generate 20 random traffic matrices.

Table 1 demonstrates that clustering is an excellent fit for approximating traffic matrices. The error bound ε is mostly below 10%, and the actual approximation error δ is always <1%. Clustering is also 5x-50x more efficient than Top-X.

As expected, decreasing traffic skeweness (for any parameter) increases both ε and δ, because it causes traffic to be more evenly spread across more sources or destinations, making it harder to capture all traffic with a fixed amount of clusters. Yet, $|C| = 300$ clusters accurately approximate unrealistic traffic matrices where traffic distributions are not very skewed at all. In contrast, Top-X needs a lot more destinations to achieve the same error bound, especially if the attraction has a less significant tail. We conclude that Velo's clustering is more effective and more general than Top-X.

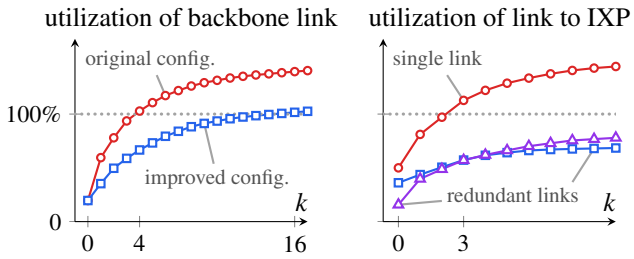
The number of clusters $|C|$ also affects the accuracy of Velo, reducing the error when increasing $|C|$. Doing so, however, also increases the running time, as discussed in §6.1. We find that Velo with $|C| = 300$ is sufficiently accurate for all matrices we evaluated, and increasing it to 1000 clusters only yields minor gains in accuracy (reducing the error bound from ≈3.5% to ≈1.8%). Appendix D.3 shows more results.

Traffic Variability We finally evaluate Velo's accuracy when traffic volumes additional to the traffic matrix is given as input (see §2). Across all the experiments described in Figure 4, Velo computes the correct maximum link loads in over 99.9% of all links, for any value of additional traffic.

Takeaways Velo is very accurate in approximating the worst-case link loads, even when using only $|C| = 300$ clusters. In all our experiments, Velo's approximation error is below 1% of the total traffic volume. Further, Velo guarantees error bounds below 12% even for unrealistic traffic matrices that are challenging to approximate.

7 Case Study

We experiment with an ISP that provides connectivity to around 100 universities and research institutions. The ISP manages 126 routers connected by 376 links. Ten routers are connected to either providers or IXPs. We generate a traffic matrix with 100 000 prefixes according to the gravity model [32] to match reported characteristics [31]. Customers only announce their respective prefixes, while all the other prefixes are reachable through neighboring providers or IXPs. In the initial network state, all links are utilized below 50%—the ISP upgrades links that maintain load above that threshold.



(a) Configuration fine-tuning (b) Network upgrade

Figure 6: Our case study demonstrates that Velo helps improving network robustness.

We run Velo on the relevant parts of the router configurations, the generated traffic matrix, and an increasing k. Within a few seconds, Velo finds that the current configuration and topology is worryingly fragile to specific egress changes: two links would exceed their capacities if egresses of only four destinations change. We now show three ways in which Velo's insight can be leveraged by operators.

Robustifying configurations One of the two fragile links is part of the ISP backbone: this link would be congested by specific egress changes for four (large) destinations. We use this insight to assess an alternative configuration with three additional MPLS paths to effectively shift traffic to underutilized links. The alternative configuration would prevent congestion on any link for up to 16 arbitrary egress changes – a much less likely scenario than 4 egress changes. Figure 6a compares the robustness of the initial and alternative configurations.

Aiding strategic decisions The second fragile link connects the ISP to an IXP. The link would be congested if traffic to just three specific destinations is sent to that IXP. Operators may think about upgrading that link; however, Velo's output highlights that other links within the ISP would also need to be upgraded to sustain the additional traffic. A better alternative is to add a link from the IXP to another, geographically close backbone router. Re-running Velo on the alternative topology reveals that $\approx 1\,000$ prefixes must be advertised at that IXP to congest either of the two links – see Figure 6b.

Velo can also guide operators' peering decisions, predicting possible configuration and infrastructural changes required by new BGP peerings.

Improving network management Velo identifies 64 pairs of prefixes and egress routers; link loads significantly increase if any such prefix is routed via the corresponding egress. This information is highly valuable for network management and operation. For example, it enables monitoring systems to focus only on those 64 pairs of prefixes and egress routers, alerting operators only for the small subset of critical events.

Velo's output can also improve automated management systems. For example, it enables traffic engineering systems (e.g., [8, 18, 20]) to optimize paths *before* link load requirements are violated. Consider the above case of the backbone link congested after four BGP egress changes. When any three egress changes are observed out of these four, an emergency recomputation of internal paths can be triggered immediately, *preventing* congestion without waiting for data-plane measurements typically collected every few minutes [20].

8 Related Work

Velo focuses on verifying link load properties, e.g., to help improving network robustness to external events. As such, it is related to prior work in network verification and routing.

Network verification Many verification systems have recently been proposed. Among them, data-plane ones [12, 21, 22, 43] target properties based on packet headers for fixed forwarding states, basically ignoring traffic and routing.

Control-plane verification systems [1, 4, 5, 14, 15, 45, 46, 54] are more related to Velo, as they admit different network settings (e.g., failures). Almost all of them however focus on properties, such as forwarding paths and routes' propagation, that cannot be directly translated into link loads. Prior works cannot be easily extended to verify link load properties, because they verify one destination at the time. As failures break the independence of destinations, thousands of destinations must be considered jointly for link load properties. Doing so with prior approaches would simply not scale: for techniques like [15], it would inflate SMT formulas by $|P|\times$, where P is the number of destinations, hence hundreds of thousands.

The few existing checkers of link load properties *do not support BGP route changes*. Jingubang [26] is a flow-level simulation tool that computes link loads for specific failure scenarios and traffic matrices. YU [27] extends the above simulator to deal with k arbitrary link failures: it still works at the per-flow level. NetDice [42] implements probabilistic verification of properties including link load ones; however, for link load properties, NetDice's approach does not scale beyond a dozen ingress-destination pairs [42]. The framework presented in [7] allows to check link utilization under arbitrary failures for network designs where configurations (e.g., routing) can be changed. Our closest related work is QARC [44], that verifies link load *violations* (without computing all link loads) upon link failures. We experimentally compare Velo against QARC in §6.1.

Routing systems Optimizing link loads has been the focus of a large body of work [2, 30, 51], especially in the area of traffic engineering (TE). TE approaches typically optimize paths based on the *current* traffic and BGP routes or, at best, variations of them [17, 25, 35]. Some of these

approaches [7, 28, 52] rely on robust optimization to *approximate* the impact of failures. Future work may try robust optimization techniques for link-load verification, although it's unclear how these techniques would guarantee accuracy and scalability when analyzing both failures and route changes.

Real-world TE systems [18, 20] update forwarding upon detecting a failure, and react to BGP route changes by re-optimizing paths according to load measurements collected every few minutes. By anticipating the most impactful failures and route changes, Velo can be integrated into TE systems to prevent excessive link loads or react faster to them, see §7.

Information on current and future traffic is key for intra-domain traffic engineering. Recently, systems (e.g., [29]) have been proposed to predict changes in incoming traffic caused by specific BGP routes *announced to* external neighbors. We envision that such systems can be used by operators to compute traffic matrices they want to verify with Velo.

9 Discussion

In this section, we discuss some of the design choices we made in Velo alongside the generality of the approach.

Why *not* modeling congestion? For each link, Velo computes the load by simply summing the amount of traffic routed over it. In practice, though, no link load could exceed its capacity: traffic would be dropped instead. This also means that the amount of traffic forwarded through the network immediately decreases for the links downstream of the congested one. Hence, when any link is congested, the link loads reported by Velo do not reflect the actual traffic volumes crossing links.

Despite this simplification, Velo's approach is practical for at least two reasons. First, Velo correctly identifies *all* the links that cannot sustain the input traffic. Second, it correctly estimates the amount of exceeding traffic, and it therefore helps quantifying how much traffic needs to be rerouted, or by how much links' capacity should be upgraded.

What protocols and features can Velo model? Despite hiding most of the complexity of BGP and intra-domain routing protocols, Velo's model is powerful enough to capture common routing architectures, including iBGP and SDN (see §3). In fact, our model only imposes two constraints on intra-domain routing. First, all routers must be able to select any of the egress routers, which means that there must be no iBGP visibility problems. Several existing techniques avoid these problems [36, 47, 48]. Second, destinations must be independent, which requires the de-activation of intra-domain routing features that tie different destinations together, such as route aggregation or conditional advertisements. The use of these features is discouraged in iBGP [37]. Additionally, we think that Velo can be extended to jointly reason about egresses of dependent destinations, if needed.

Data-plane filters like ACLs do not influence routing, and thus, Theorem 1 still applies. In their presence, Algorithms 1 and 2 can be easily extended to apply such filters by dropping (some) traffic instead of propagating it. Doing so would ensure that the worst-case link loads computed by Velo would be consistent with both routing inputs and data-plane filters.

What route changes does Velo support? Velo computes the worst-case link loads for route changes affecting subsets of destinations. In §3, we have defined destinations as the IP prefixes in the routing tables. This allows Velo to cover BGP route changes for any of these IP prefixes.

In reality, the set of IP prefixes learned via BGP can change over time. For example, at any time, the remote network owning an IP prefix, say 1.0.0.0/8, can start announcing a more specific prefix, such as 1.0.0.0/16. If this happens, BGP propagates two distinct and independent routes – e.g., one for 1.0.0.0/8 and the other for 1.0.0.0/16.

Since Velo learns the destinations from the input traffic matrix, it supports some analyses of route changes affecting the set of known destinations, provided that enough information is specified by operators. For instance, Velo can handle the previous example if the input matrix includes traffic volumes for 1.0.0.0/16 and (separately) for the other subnets in 1.0.0.0/8. This information is anyway necessary to compute link loads with distinct routes for 1.0.0.0/16 and 1.0.0.0/8.

In contrast, Velo does not currently support systematic explorations of route changes affecting the set of destinations, e.g., announcements of *any* possible sub-prefix of a current destination. We leave this direction for future work.

10 Conclusions

Velo demonstrates the feasibility of verifying worst-case link loads in large networks, despite the need to navigate a gigantic space of possible failures and route changes. Our evaluation shows that Velo is both scalable and accurate across a wide range of topologies, settings, constraints and traffic matrices.

Velo focuses on maximum link loads due to their straightforward practical relevance in assessing network performance. As our case study demonstrates, Velo readily supports operators in complex tasks ranging from improving configurations to network design and aiding business decisions.

We also see Velo as a stepping stone towards the broader goal of verifying network performance. We believe that exploring other performance aspects, such as delay or bandwidth guarantees, is an interesting avenue for future research.

Acknowledgments The research leading to these results was supported by an ERC Starting Grant (SyNET) 851809. We thank João Luís Sobrinho, our shepherd Siva Kakarla and all anonymous reviewers for their valuable feedback. We also thank Simon Leinen and Switch for providing traffic data.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, 2020.

[2] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71:1–30, 2014.

[3] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Dr. Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels. RFC 3209, Internet Engineering Task Force, December 2001.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 476–489, 2018.

[6] Vineet Bharti, Pankaj Kankar, Lokesh Setia, Gonca Gürsun, Anukool Lakhina, and Mark Crovella. Inferring invisible traffic. In *Proceedings of the 6th International Conference*, Co-NEXT '10, New York, NY, USA, 2010. Association for Computing Machinery.

[7] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust Validation of Network Designs under Uncertain Demands and Failures. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 347–362, Boston, MA, March 2017. USENIX Association.

[8] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. Mate: Mpls adaptive traffic engineering. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1300–1309. IEEE, 2001.

[9] Wenjia Fang and Larry Peterson. Inter-as traffic patterns and their implications. In *Seamless Interconnection for Universal Services. Global Telecommunications Conference. GLOBECOM'99.(Cat. No. 99CH37042)*, volume 3, pages 1859–1868. IEEE, 1999.

[10] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. The case for separating routing from routers. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, page 5–12, New York, NY, USA, 2004. Association for Computing Machinery.

[11] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions On Networking*, 9(3):265–279, 2001.

[12] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, 2015.

[13] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *Proceedings IEEE INFOCOM 2000. conference on computer communications. Nineteenth annual joint conference of the IEEE computer and communications societies (Cat. No. 00CH37064)*, volume 2, pages 519–528. IEEE, 2000.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 300–313, 2016.

[15] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. Nv: An intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 958–973, 2020.

[16] Timothy G Griffin and Gordon Wilfong. On the correctness of ibgp configuration. *ACM SIGCOMM Computer Communication Review*, 32(4):17–29, 2002.

[17] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *Proceedings of the ACM SIGCOMM 2018 Conference*, page 88–102, 2018.

[18] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[19] Geoff Huston. Measuring BGP in 2023—have we reached peak ipv4? APNIC Blog, Jan 2024.

[20] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally Deployed Software Defined WAN. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013.

[21] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, 2012.

[22] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 49–54, 2012.

[23] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[24] Gottfried Köthe and Gottfried Köthe. *Topological vector spaces*. Springer, 1983.

[25] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, 2018.

[26] Ruihan Li, Fangdan Ye, Yifei Yuan, Ruizhen Yang, Bingchuan Tian, Tianchen Guo, Hao Wu, Xiaobo Zhu, Zhongyu Guan, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, , and Ennan Zhai. Reasoning about network traffic load property at production scale. In *21st USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[27] Ruihan Li, Yifei Yuan, Fangdan Ye, Mengqi Liu, Ruizhen Yang, Yang Yu, Tianchen Guo, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, and Ennan Zhai. A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures. In *Proceedings of the ACM SIGCOMM 2024 Conference*, page 228–243, New York, NY, USA, 2024. Association for Computing Machinery.

[28] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, page 527–538, New York, NY, USA, 2014. Association for Computing Machinery.

[29] Michael Markovitch, Sharad Agarwal, Rodrigo Fonseca, Ryan Beckett, Chuanji Zhang, Irena Atov, and Somesh Chaturmohta. TIPSY: predicting where traffic will ingress a WAN. In *Proceedings of the ACM SIGCOMM 2022 Conference*, page 233–249, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Alaitz Mendiola, Jasone Astorga, Eduardo Jacob, and Marivi Higuero. A survey on the contributions of software-defined networking to traffic engineering. *IEEE Communications Surveys & Tutorials*, 19(2):918–953, 2016.

[31] Jakub Mikians, Amogh Dhamdhere, Constantine Dovrolis, Pere Barlet-Ros, and Josep Solé-Pareta. Towards a statistical characterization of the interdomain traffic matrix. In *NETWORKING 2012: 11th International IFIP TC 6 Networking Conference, Prague, Czech Republic, May 21-25, 2012, Proceedings, Part II 11*, pages 111–123. Springer, 2012.

[32] Jakub Mikians, Nikolaos Laoutaris, Amogh Dhamdhere, and Pere Barlet-Ros. Itmgen—a first-principles approach to generating synthetic interdomain traffic matrices. In *2013 IEEE International Conference on Communications (ICC)*, pages 2507–2512. IEEE, 2013.

[33] John Moy. OSPF Version 2. RFC 2328, Internet Engineering Task Force, April 1998.

[34] Dave Oran. OSI IS-IS Intra-domain Routing Protocol. RFC 1142, Internet Engineering Task Force, February 1990.

[35] Yarin Perry, Felipe Vieira Frujeri, Chaim Hoch, Srikanth Kandula, Ishai Menache, Michael Schapira, and Aviv Tamar. {DOTE}: Rethinking (predictive){WAN} traffic engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1557–1581, 2023.

[36] Robert Raszuk, Bruno Decraene, Christian Cassar, Erik Aman, and Kevin Wang. BGP Optimal Route Reflection (BGP ORR). RFC 9107, Internet Engineering Task Force, August 2021.

[37] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Internet Engineering Task Force, January 2006.

[38] Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35(5):93–96, 2005.

[39] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: Synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 33 – 49, New York, NY, 2021-08. Association for Computing Machinery.

[40] Tibor Schneider, Roland Schmid, Stefano Vissicchio, and Laurent Vanbever. Taming the transient while reconfiguring BGP. In *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.

[41] João L Sobrinho. Algebra and algorithms for qos path computation and hop-by-hop routing in the internet. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society*, volume 2, pages 727–735. IEEE, 2001.

[42] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Probabilistic verification of network configurations. In *Proceedings of the Annual ACM SIGCOMM Conference of the ACM Special Interest Group on Data Communication*, Online, 2020. Association for Computing Machinery.

[43] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 314–327, 2016.

[44] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. Detecting network load violations for distributed control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 974–988, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 94–107, 2023.

[46] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Kirigami, the verifiable art of network cutting. *IEEE/ACM Transactions on Networking*, 2024.

[47] Jim Uttaro, Pierre Francois, Keyur Patel, Jeffrey Haas, Adam Simpson, and Roberto Fragassi. Best Practices for Advertisement of Multiple Paths in IBGP. Internet-Draft draft-ietf-idr-add-paths-guidelines-08, Internet Engineering Task Force, April 2016. Work in Progress.

[48] Virginie Van den Schrieck, Pierre Francois, and Olivier Bonaventure. Bgp add-paths: the scaling/performance tradeoffs. *IEEE Journal on Selected Areas in Communications*, 28(8):1299–1307, 2010.

[49] Stefano Vissicchio, Luca Cittadini, Laurent Vanbever, and Olivier Bonaventure. iBGP deceptions: More sessions, fewer routes. In *Proceedings of the IEEE INFOCOM Conference*, 2012.

[50] Dan Wang, Peng Zhang, and Aaron Gember-Jacobson. Expresso: Comprehensively Reasoning About External Routes Using Symbolic Simulation. In *Proceedings of the ACM SIGCOMM 2024 Conference*, page 197–212, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Ning Wang, Kin Hon Ho, George Pavlou, and Michael Howarth. An overview of routing optimization for internet traffic engineering. *IEEE Communications Surveys & Tutorials*, 10(1):36–56, 2008.

[52] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. R3: resilient routing reconfiguration. In *Proceedings of the ACM SIGCOMM 2010 Conference*, page 291–302, New York, NY, USA, 2010. Association for Computing Machinery.

[53] Fatai Zhang, Oscar Gonzalez de Dios, Matt Hartley, Zafar Ali, and Cyril Margaria. RSVP-TE Extensions for Collecting Shared Risk Link Group (SRLG) Information. RFC 8001, Internet Engineering Task Force, January 2017.

[54] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic router execution. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 336–349, 2022.

A Exception Paths

In the following, we first proof Theorem 2 that pretains to we handle exception paths. We then present the algorithm that relies on Theorem 2 to find the worst-case link loads under optimal and exception paths.

A.1 Proof of Theorem 2

Proof. Consider edge $e = (u,v)$, and let $b \in B$ be an egress router preferred by v. Now, take any node $n \in N_e$ that forwards traffic along the edge e for destination d. We distinguish two cases:

1. Paths of n that traverse e are optimal (i.e., are not exception paths), and by Theorem 1, n will not decrease the traffic it sends along edge e.
2. n forwards traffic along exception paths in ρ_d. Then, n will use the same forwarding paths for egresses $\{b\} \cup T \cap B$ as all destinations of n's forwarding paths are within T and B. Thus, n will still forward the same amount of traffic over link e.

It follows that the traffic on e cannot decrease for egresses $\{b\} \cup T \cap B$, proving Theorem 2. □

A.2 Algorithm for Exception Paths

Algorithm 2 shows the algorithm for finding the worst-case loads for all network links in the presence of exception paths, i.e., paths that don't follow the optimal paths according to the strictly isotone routing protocol.

The function `selected_combinations` iterates over all combinations of egress routers within T and all individual routers in $N_B \setminus T$. It further, checks whether there exists a set of advertisements for all egresses to be selected simultaneously based on the `LocalPref` values.

Algorithm 2: Find the worst-case link load for all links for optimal or exception paths.

Data: Graph $G = (N,E)$, border routers $N_B \subseteq N$, link weights $w : E \mapsto \mathbb{R}$, traffic $\mathbf{M}_d$, paths ρ_d
Result: maximum link loads $y[e]$ for all links $e \in E$

$T \leftarrow \{b \mid (s, \cdots, b) \in \rho_d\}$
$y[u,v] \leftarrow 0 \; \forall (u,v) \in E$
$G_{dag}[b] \leftarrow \mathit{ForwardingDAG}(G, w, root = b) \; \forall b \in N_B$
for $B \in$ `selected_combinations`(N_B, T) **do**
  $y'[u,v] \leftarrow 0 \; \forall (u,v) \in E$
  $l[s] \leftarrow \mathbf{M}_{s,d} \; \forall s \in N$
  $G_{fw} \leftarrow \mathit{merge}(G_{dag}[b] \; \forall b \in B)$
  for $path = (s, n_1, \cdots, n_i, b) \in \rho_d$ **do**
    if *s forwards to b in* G_{fw} **then**
      $y'[u,v] \leftarrow y'[u,v] + l[s] \; \forall (u,v) \in path$
      $l[s] \leftarrow 0$
  for $u \in \mathit{TopoSort}(G_{dag})$ **do**
    $d_{out} \leftarrow |\mathit{out}(G_{dag}, u)|$
    for $v \in \mathit{out}(G_{dag}, u)$ **do**
      $l[v] \leftarrow l[v] + l[u]/d_{out}$
      $y'[u,v] \leftarrow y'[u,v] + l[u]/d_{out}$
      $y[u,v] \leftarrow \max(y[u,v], y'[u,v])$

B Clustering Error Bounds

In the following, we formally proof that the clustering error bounds the approximation error of the worst-case link load. To that end, we first introdoce two lemmas the pertain to linearity and the superposition of maximum link loads before proofing Theorem 3.

Lemma 1 *$\mathit{max\,load}(e, \mathbf{M})$ is linear in terms of $\mathbf{M}$, i.e.,*

$$\forall a \geq 0 : a \cdot \mathit{max\,load}(e, \mathbf{M}) = \mathit{max\,load}(e, a \cdot \mathbf{M})$$

Proof. Intuitively, Lemma 1 holds, because scaling the traffic matrix does not affect the worst-case routing state, and thus, the forwarding paths in the worst case and the link loads remain unchanged. More formally, the worst-case routing state is obtained using a set of inequalities over sums of traffic volumes (see Algorithm 1). Scaling each entry in the traffic matrix does not influence these inequalities. The link loads in that routing state are computed by adding individual entries in the matrix, which itself is a linear operation. Thus, *max load* is linear in terms of $\mathbf{M}$. □

Lemma 2 *$\mathit{max\,load}(e, \mathbf{M})$ is equal the superposition of the maximum link loads for all columns of $\mathbf{M}$. In other words,*

$$\mathit{max\,load}(e, \mathbf{M}) = \sum_d \mathit{max\,load}(e, \mathbf{M}_d)$$

Proof. Lemma 2 is intuitively true because the forwarding decision for one destination is independent of the routing state of another destination. In other words, we can tweak the worst-case routing input for destination d_1 independently of d_2 to obtain the maximum load on link e for destinations d_1 and d_2. Then, the load on link e is equal to the sum of all traffic for all individual destinations, including d_1 and d_2. □

Theorem 3 *The approximation error δ of computing the maximum link loads for the approximated traffic matrix $\mathbf{A}$ is bounded by ε. More formally,*

$$\delta = \max_{e \in E} |\mathit{max\,load}(e, \mathbf{M}) - \mathit{max\,load}(e, \mathbf{A})| \leq \varepsilon. \quad (3)$$

Proof. We prove Theorem 3 by considering each individual destination d and its contribution to both the clustering and the approximation error. To that end, we define $\overline{\mathbf{M}}_d$ as the approximation of d's traffic $\mathbf{M}_d$, and express both the approximation error δ and the clustering error ε in terms of $\overline{\mathbf{M}}$.

Since the approximated matrix $\mathbf{A}$ is of a different shape than $\mathbf{M}$, we first describe how to relate the contribution of any destination $d \in c_i$ to its cluster centroid $\mathbf{A}_i$. To that end, we define $\overline{\mathbf{M}}_d = |\mathbf{M}_d| \cdot \mathbf{A}_i / |\mathbf{A}_i|$ as the approximated traffic for destination d. Notice, that the clustering error ε, defined in Equation (2), can be rewritten as:

$$\varepsilon = \frac{1}{2} \sum_{c_i \in C} \sum_{d \in c_i} \left| \mathbf{M}_d - |\mathbf{M}_d| \frac{\mathbf{A}_i}{|\mathbf{A}_i|} \right| = \frac{1}{2} \sum_{d \in P} \left| \mathbf{M}_d - \overline{\mathbf{M}}_d \right|.$$

Further, the maximum link load $\mathit{max\,load}(e, \mathbf{A})$ can expressed in terms of $\overline{\mathbf{M}}$ by relying on the definition of $\mathbf{A}_i$ in Equation (2), and the linearity and decomposition of maximum link loads. We prove both properties in Appendix B.

$$\begin{aligned} \mathit{max\,load}(e, \mathbf{A}) &= \sum_{c_i \in C} \mathit{max\,load}(e, \mathbf{A}_i) \\ &= \sum_{c_i \in C} \sum_{d \in c_i} \mathit{max\,load}(e, \mathbf{A}_i) \cdot |\mathbf{M}_d| \,/\, |\mathbf{A}_i| \\ &= \sum_{d \in P} \mathit{max\,load}(e, \overline{\mathbf{M}}_d) \end{aligned}$$

Now, let's focus on the maximum link load for $\mathbf{M}_d$ and $\overline{\mathbf{M}}_d$ on link e. To that end, we consider traffic x_d^+ that is part in the approximation $\overline{\mathbf{M}}_d$ but not in the original $\mathbf{M}_d$, and traffic x_d^- of $\mathbf{M}_d$ that is not part in $\overline{\mathbf{M}}_d$:

$$x_d^+ = \sum_s \max(0, \overline{\mathbf{M}}_{s,d} - \mathbf{M}_{s,d}) \quad x_d^- = \sum_s \max(0, \mathbf{M}_{s,d} - \overline{\mathbf{M}}_{s,d})$$

Let's first focus on x_d^+, but the argument for x_d^- is symmetrical by swapping $\mathbf{M}$ and $\overline{\mathbf{M}}$. The difference of maximum link loads for $\overline{\mathbf{M}}_d$ and $\mathbf{M}_d$ is at most the additional traffic x_d^+ of $\overline{\mathbf{M}}_d$, i.e., $\mathit{max\,load}(e, \overline{\mathbf{M}}_d) - \mathit{max\,load}(e, \mathbf{M}_d) \leq x_d^+$. That is because $\mathit{max\,load}(e, \mathbf{M}_d)$ by definition is the maximum load achievable on link e, and adding traffic x_d^+ cannot increase that maximum by more than x_d^+. Therefore, $\delta \leq x_d^+$ and $\delta \leq x_d^-$.

Finally, we show that $\sum_d x_d^+ = \sum_d x_d^- = \varepsilon$. This is because, by definition of the L1 norm, $x_d^+ + x_d^- = \left| \overline{\mathbf{M}}_d - \mathbf{M}_d \right|$, and that the centroid $\mathbf{A}_i = \sum_{d \in c_i} \overline{\mathbf{M}}_d$ contains equal positive and negative errors, i.e., $x_d^+ = x_d^-$. This proves Theorem 3. □

C Methodology for Table 1

The following elaborates on the Methodology for Table 1. Specifically, we detail how we find the attraction, repulsion, and friction parameters from a (real or synthetic) traffic matrix, and our processing that was required to obtain the real traffic matrices from aggregated netflow statistics.

C.1 Characterizing a Traffic Matrix

We characterize a traffic matrix according to the three distributions of the gravity model. Specifically, we assume the matrix was generatred from a gravity model, with repulsion, attraction, and friction sampled independently from LogNormal distributions. We then obtain the most likely σ parameter.

To characterize the attraction, we compute the total traffic to each destionation d, normalized to have a mean of 1:

$$\forall d \in P: \ \overline{\mathbf{M}}_d = \frac{|P|}{|\mathbf{M}|} \sum_s \mathbf{M}_{s,d}.$$

We then assume $\{\overline{\mathbf{M}}_d \mid d \in P\} \sim \mathit{LogNormal}(\sigma, \mu = 0)$ to be sampled from a LogNormal distribution with $\mu = 1$ (the expected value in log-space) and find the most likely σ parameter. Similarly, we repeat the same process for the repulsion, but with traffic from each ingress router:

$$\forall s \in N: \ \overline{\mathbf{M}}_s = \frac{|N|}{|\mathbf{M}|} \sum_d \mathbf{M}_{s,d}.$$

For the friction, we use the traffic from each source and to each destination to find the variance for each element in the matrix. In other words, we find find the most likely σ assuming $\{\overline{f}_{s,d} \mid (s,d) \in N \times P\} \sim \mathit{LogNormal}(\sigma, \mu = 0)$ with

$$f_{s,d} = \frac{\mathbf{M}_{s,d}}{\overline{\mathbf{M}}_s \cdot \overline{\mathbf{M}}_d}, \quad \overline{f}_{s,d} = \frac{|N| \cdot |P|}{\sum_{s,d} f_{s,d}} f_{s,d}.$$

By dividing each element $\mathbf{M}_{s,d}$ by the total traffic from source s to destination d, we are left with the variation factor $\mathit{friction}(s,d)$ from the gravity model.

C.2 Real Traffic Matrices

The ISP samples traffic on all links towards external peers and providers, but not on links to their customers. The collected netflow data consists of byte counters aggregated by souce and destination IP addresses (either /24 or /48), and the specific egress link on which the packet was sampled, including its direction (whether it leaves or enters the ISP).

To construct the inter-domain traffic matrix (i.e., byte counters per ingress router and destination prefix), we must associate destination addresses of ingress traffic (and source addresses of egress traffic) to the customer associated with that prefix. Together with the operators, we constructed a mapping from prefixes to customers. Unfortunately, this mapping is neither complete nor unique. Some prefixes are related to multiple customers that did not obtain full /24 or /48 prefixes. Further, some prefixes could not be assigned to customers due to time constraints. This affects around 30% of the total traffic volume[3].

[3] We plan to improve this mapping for the final version.

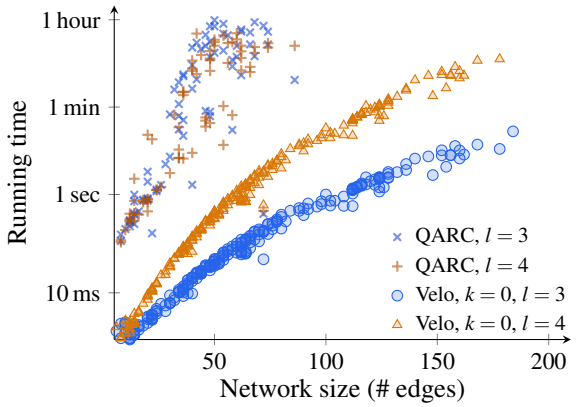


Figure 7: For $l = 3$ and $l = 4$, Velo still outperforms QARC, yet the performance gap decreases.

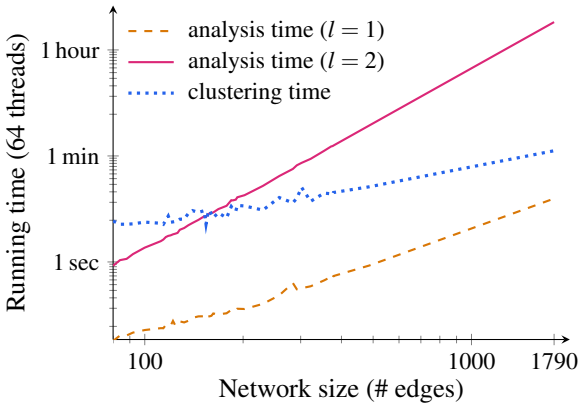


Figure 8: The running time of Velo in terms of the time to perform the clustering and the time to find the worst-case link failures.

The data further allows us to reconstruct the routing state (i.e., the selected egress routers) at time of measurement. For ingress traffic, the traffic is always routed towards the respective customer. For egress traffic, the egress interface on which the traffic is observed is the selected egress point.

D Supplementary evaluation

We provide additional measurements to support §6.

D.1 Comparison with QARC

Figure 7 compares the running time of Velo with QARC for finding worst-case link loads under three and four simultaneous link failures. In 15% of these scenarios, QARC timeouts after running for one hour, while Velo can find the worst-case within 20 minutes for networks up to twice as large.

D.2 Clustering Time vs. Analysis Time

Velo's total running time can be separated into the *clustering time* to find the approximate traffic matrix and the time to find the maximum link loads, which we call the *analysis time*. Figure 8 shows both the clustering and the analysis time for the largest 75 networks on a log-log plot, for both $l = 1$ and $l = 2$. We use the same parameters as described in §6.1. We find that the clustering time dominates the running time for small networks, while it can be neglected for large networks. Further, both the analysis time and the clustering time appear as straight lines in the log-log plot, indicating they scale polynomially in the network size. The analysis time scales roughly quadratically for $l = 1$ and cubic for $l = 2$, as Velo considers $|E|$ additional failure scenarios. Finally, the clustering time scales linearly in the network size which corresponds to the dimensionality of the clustering problem.

Table 2: Velo's accuracy and running time for varying number of clusters

		error bound ε	approx. error δ
$\|C\|$	time	0 **20%**	0 **3%**
100	178 s	16.23%	2.66%
300	210 s	7.34%	0.88%
600	268 s	4.95%	0.39%
1000	343 s	3.71%	0.22%

D.3 Performance–Accuracy Tradeoff

Velo can be configured to an arbitrary number of clusters. Increasing the number of clusters will yield more accurate results at the expense of longer running times, as Algorithm 1 is executed for each cluster.

To evaluate the performance–accuracy tradeoff, we run Velo on the same 6 topologies as in §6.2, choose the traffic matrix using the same baseline distributions, and choose $l = 2$, $k = 10$, and $|N_B| = 30$.

Table 2 shows both Velo's running time and accuracy for 100, 300, 600, and 1 000 clusters. We identify a substantial improvement in accuracy when increasing $|C|$ from 100 to 300, especially in terms of the error bounds. However, increasing $|C|$ to 1000 yields a much smaller accuracy improvement at the expense of a 1.6× increase in running time.

Based on the results in Table 2, we choose $|C|$ as a reasonable tradeoff between accuracy and running time. Yet, the number of clusters can be tuned to the operator's use case to either reduce the running time or improve the accuracy.

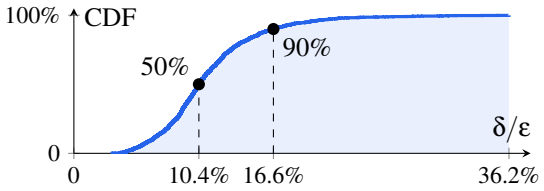


Figure 9: The approximation error δ is typically over $100\times$ smaller than the error bounds ε. This plot shows the CDF of the fraction δ/ε, a measure of how "tight" ε bounds δ.

D.4 Tightness of error bounds.

Table 1 shows that the clustering error ε typically 10 times larger than the approximation error δ. This demonstrates how our error bounds is pessimistic, as it *sums* the absolute values of all the clustering errors as if all the traffic would traverse every single link. In practice, however, differences in a single destination's traffic distribution do not necessarily cause the worst-case routing state for that destination to change. Further, if a few destinations in that cluster have different worst-case routing state, their impact on the resulting load can be both positive or negative, likely canceling each other out.

Figure 9 displays the CDF of the approximation error δ divided by the error bounds ε to visualize how tight the theoretical and real errors are. In 50% of our experiments, the approximation error δ is within 10.4% of the error bounds ε. In 90%, it is within 16.6%.

A Layered Formal Methods Approach to Answering Queue-related Queries

Divya Raghunathan
Princeton University

Maria Apostolaki
Princeton University

Aarti Gupta
Princeton University

Abstract

Queue dynamics introduce significant uncertainty in network management tasks such as debugging, performance monitoring, and analysis. Despite numerous queue-monitoring techniques, many networks today continue to collect only per-port packet counts (*e.g.,* using SNMP). Although queue lengths are correlated with packet counts, deriving the precise correlation between them is very challenging since packet counts do not specify many quantities (*e.g.,* packet arrival order) which affect queue lengths.

This paper presents QUASI, a system that can answer many queue-related queries using only coarse-grained per-port packet counts. QUASI checks whether there exists a packet trace that is consistent with the packet counts and satisfies a query. To scale on large problem instances, QUASI relies on a layered approach and on a novel *enqueue-rate* abstraction, which is lossless for the class of queries that QUASI answers. The first layer employs a novel and efficient algorithm that generates a *cover-set* of abstract traces, constructs *representative* abstract traces from the cover-set, and efficiently checks each representative abstract trace by leveraging a known result on $(0,1)$-matrix existence. The first layer guarantees no false negatives: if the first layer says "No", there is no packet trace consistent with the observed packet counts that makes the query true. If it says "Yes", further verification is needed, which the second layer resolves using an SMT solver. As a result, QUASI has no false positives and no false negatives.

Our evaluations show that QUASI is up to 10^6X faster than state-of-the-art, and can answer non-trivial queries about queue metrics (*e.g.,* queue length) using minute-granularity packet counts. Our work is the first step toward more practical formal performance analysis under given measurements.

1 Introduction

Many challenging network management tasks can be reduced to queue-related queries, as queuing delays often represent the most significant source of uncertainty for a packet on its journey [21]. For instance, a network operator can verify a latency Service Level Objective (SLO) for a path by querying the bounds for each queue's length on that path. Similarly, a decision about whether to increase the size of on-chip buffers or whether to allow certain applications to share a switch could be informed by querying how often multiple queues of the same device experience bursts concurrently [1, 2].

While there are various techniques for measuring queue behavior, fine-grained queue monitoring is often costly to collect, store, and analyze, and it may even require specialized hardware [6, 9, 16, 18, 20]. As a result, commodity data centers, ISPs, and enterprise networks often rely on simpler, always-on packet count monitors that track the number of incoming and outgoing packets at each port (*e.g.,* SNMP [7, 12, 19, 23, 25, 26]). These per-port packet counts are highly scalable and are typically collected at coarse-grained intervals, such as every few minutes. They are useful for estimating average utilization and identifying traffic patterns, but the widely held assumption is that they cannot be used for answering queue-related queries.

Upon closer examination, this assumption appears flawed. Per-port packet counts are actually closely correlated with queue lengths and can impose significant constraints on queue dynamics, hence helping answer queue-related queries. For instance, multiple packets are required for a queue to form, and bursts only occur when multiple ports simultaneously send to the same queue (fan-in). Additionally, there should be a limit to the number of concurrent bursts a symmetric device can handle. These observations motivate the question this paper seeks to answer: *Can we answer queue-related queries using only coarse-grained packet counts?*

At first glance, this task appears impossible. The first hurdle lies in the discrepancy between the query metric (queue length) and the available data (per-port packet counts). Although these two are closely related, establishing a closed-form relationship between them is not straightforward. The counts only provide the total number of packets received and sent at each port, without information on their timing, sequences, or routing (i.e., which input port's packet was forwarded to which output port and when). This information defines a *packet trace* and are

crucial for deducing queue lengths. Hence, to answer queries about queue lengths, one would need to consider a vast number of packet traces, identify those consistent with the given per-port counts, and then determine the answer based on these traces. The search space expands significantly as the monitoring interval duration increases. An alternative approach would be to leverage FPerf [4], a system that models network components and their queues in logic, and synthesizes workloads that satisfy a given query. FPerf solves a different performance analysis problem, so we modified FPerf to check if there exists a packet trace consistent with given measurements (specified as constraints over a workload) that satisfies a given query. However, FPerf does not support output and dropped packet count measurements, and quickly becomes unscalable (§7.4).

Layered approach. To navigate the search space of packet traces more efficiently, we have developed a layered approach called QUASI (**Que**ue **A**nalysis from **S**NMP **I**nformation), drawing inspiration from the abstraction-refinement paradigm [10], which has seen tremendous success in the formal methods community [8, 17]. At its core, QUASI provides Boolean answers to queries (*e.g.,* answer yes or no: could this queue have n packets?). QUASI can also handle quantitative queries by running multiple Boolean queries (*e.g.,* to answer: what is the maximum length this queue could have reached?). The first layer of QUASI performs an efficient over-approximate analysis using representations of packet traces as $(0,1)$-matrices, such that a negative result is conclusive and does not require further analysis – we prove that QUASI's first layer will not incorrectly answer a query negatively. If this first layer is inconclusive, QUASI progresses to a second layer that utilizes an exact but less scalable analysis based on use of a Satisfiability Modulo Theories (SMT) solver [5, 11]. This layered strategy allows QUASI to balance efficiency and accuracy, reserving the use of more computationally intensive methods for situations where they are genuinely necessary.

Novel abstraction. QUASI is centered around a novel, powerful, and lossless abstraction that we call *enqueue-rate*. The enqueue-rate abstraction enables QUASI to reason about multiple packet traces simultaneously, thereby improving efficiency without any loss of accuracy. This abstraction is based on the insight that the exact routing of packets between input and output ports is not relevant for answering the queue-related queries supported by QUASI.

Efficient over-approximate analysis. QUASI's first layer leverages the enqueue-rate abstraction and incorporates three key innovations that result in high efficiency. First, QUASI derives necessary conditions from the query, represented as a disjunction of concise representations of sets of abstract traces; each set can be investigated independently or in parallel with others. Second, we show that for each such set of abstract traces, we can construct a special abstract trace — termed the *most-uniform* abstract trace — that serves as a representative of all abstract traces in the set. In particular, we prove that it is enough to check just the representative abstract trace for each set. Finally, we show that we can efficiently check each representative abstract trace using a known result from combinatorics, specifically the Gale-Ryser theorem [13, 24].

Exact Analysis. QUASI's second layer also uses the enqueue-rate abstraction, using an SMT solver to precisely model the *exact* (rather than necessary) conditions for an abstract trace to be consistent with the given measurements and the query.

Summary of results. We evaluate QUASI on simulated data and compare it against a heuristic baseline and FPerf [4] on a variety of queries (both Boolean and quantitative). Our results show that QUASI is useful: QUASI's first layer finds upper bounds on maximum queue length and buffer occupancy up to 58% tighter than bounds found by a heuristic analysis within one second; QUASI's second layer finds the exact value in 25 minutes. QUASI is 10^6x faster than FPerf and supports an important class of burst-related queries that FPerf does not support. QUASI scales to realistic monitoring intervals (5 mins/300 million time steps) for queue length queries.

2 Motivation

In this section, we start by describing use cases that showcase the usefulness of provably answering queue-related queries, especially when done with coarse-grained per-port packet count measurements. Then, we explain the shortcomings of existing approaches to measuring and inferring queue dynamics.

2.1 Why are Queue-related Queries Useful?

Having visibility on queue lengths in a network (through direct or indirect measurements) is critical for multiple network management tasks. We discuss use cases related to root-cause analysis, provisioning, and performance guarantees.

First, visibility over queue lengths makes debugging network incidents easier. Consider, for example, an operator debugging an incident of packet drops on a particular network device. The operator would benefit from learning whether the drops could have been caused by buffer pressure (*i.e.,* excessive buffering on ports distinct from the one dropping), or whether they could have been from a bursty application. The operator could learn this by looking at the queue-length time series of each queue on the device. Yet access to such data via monitoring alone would require an always-on tool that monitors all queues of all devices at a millisecond (at least) granularity. Considering the cost, a tool that would *provably* answer at least some such queries using cheap coarse-grained packet counts would be useful.

Second, visibility over queue lengths simplifies network planning. Consider an operator that aims to place applications in their network avoiding bursty applications sharing the same device, or an operator investigating whether they need to upgrade on-device buffers. Observe that such a query needs to run over historical queue length data spanning a good amount

of time to contain various applications. Even if these data were collected in the past at ms granularity, it is unlikely that they would be currently available. Hence, a tool that would "piggyback" queries about queues on coarse-grained packet counts that are used for other tasks (*e.g.,* predicting utilization, traffic matrices) and hence are more likely to be available, would be very useful.

Finally, answering queue-related queries could be useful for verification. Queue management in high-speed switches involves intricate mechanisms for flow control across multiple queues on different ports. Implementation errors by equipment manufacturers can result in counterintuitive outcomes, such as high packet loss and delays even during periods of low link utilization. Investigating whether the device's working is consistent with the specification is very challenging. Accurate and fine-grained queue length monitoring would be useful but might not be reliable. Hence, it would help to cross-check those measurements with coarse-grained packet counts. In this case, a tool that can provably answer queries about what queue length values are possible given the packet counts would be useful.

2.2 Limitations of Inferring Queue Lengths

Directly measuring and storing queue lengths at a fine-grained granularity (e.g., nanosecond level) in an always-on manner is impractical [15], leading to the development of various inference methods such as Zoom2Net [15] and SIMON [14]. While these methods are innovative in that they do not require additional resources or hardware, they lack correctness guarantees. For example, Zoom2Net generates fine-grained telemetry guaranteed to satisfy selected constraints enforcing consistency with coarse-grained measurements and switch operation. However, these constraints are not sufficient, so the generated fine-grained telemetry might be impossible in practice. Also, it cannot prove, *e.g.,* that a certain queue length could not have occurred. This limitation becomes particularly problematic when an operator needs to understand the worst-case scenario or is concerned about a specific set of scenarios with some property. Consider an operator troubleshooting an incident of increased end-to-end latency: knowing the worst-case queuing delay for each device on-path (or that the delay has not exceeded some threshold) would help her safely narrow down the search to a smaller set of devices. However, if the inferred queueing delay is an underestimate, it can mislead the operator, complicating the debugging process. Beyond these limitations, both works assume access to coarse-grained measurements of queue length/delay. Zoom2Net considers periodic and maximum queue lengths (along with packet counts), and SIMON assumes end-to-end queueing delay.

3 Overview of QUASI

After formally defining our problem (§3.1), we introduce a running toy example that shows how leveraging coarse-grained packet counts can be useful in deducing queue dynamics

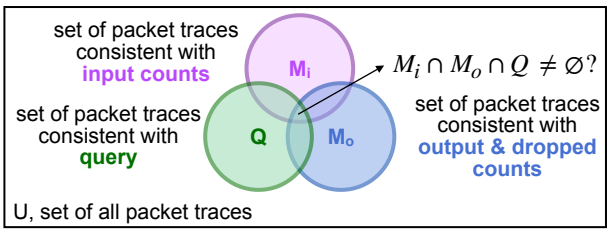


Figure 1: We recast the problem of answering a queue-related query under given measurements into the problem of checking whether the intersection of M_i, M_o, and Q is non-empty.

(§3.2). More importantly, the example showcases that even in such a seemingly simple example, answering a queue-related query is nontrivial. We use this example to illustrate the key steps in how QUASI answers queue-related queries in §4.4, after we have introduced key concepts in §4.3.

3.1 Problem Definition

We are given packet counts of received (input), sent (output), and dropped packets for each port of a switch, aggregated over a monitoring interval. We are also given a query describing an incident/property or value depending on one or more queues during the interval which can be answered positively or negatively.

Goal. Our goal is to build a system that can answer positively if there exists some sequence of packets (called a *packet trace*) which, if arrived at the switch during the interval, would be consistent with input, output, and dropped counts, and would satisfy the query. The system should answer negatively if such a packet trace does not exist. The exact syntax of the query language is shown in Fig. 4 (§4.2).

More concretely, we formulate the problem of answering a queue-related query as determining whether the intersection of three sets is non-empty, as shown in Fig. 1:

M_i : set of all packet traces consistent with the per-port input counts of all ports of a switch.

M_o : set of all packet traces consistent with the per-port output and dropped counts of all ports of a switch.

Q : set of all packet traces that make the query true.

There exists a packet trace consistent with the measurements that makes the query true if and only if $M_i \cap M_o \cap Q \neq \emptyset$.

3.2 Motivating Example

Consider a cloud operator who receives a complaint from a customer claiming that the latency between her silicon servers violated the service level objective (SLO) at some point between 9 am and 10 am on a given day. Since the customer's servers are connected through a single switch (shown in Fig. 2), the operator deduces that the only way the cloud provider might be liable is if the queueing latency at the switch exceeded a threshold close to the SLO during that time. Consider that the operator lacks access to fine-grained queue-length measurements, either because the hardware

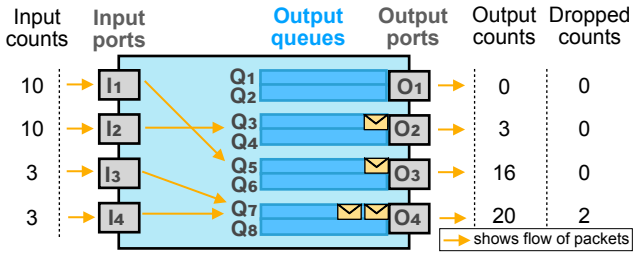


Figure 2: Output-queued switch with per-port packet counts.

does not support such fine-grained monitoring or because collecting and storing these measurements is only available on-demand due to its high cost. Instead, the operator has access to SNMP [12] packet counts *i.e.,* per-port packet counts aggregated over a monitoring interval of 20 time steps, where each time step represents either a packet received or sent.[1]

Fig. 2 shows an output-queued switch with 4 ports. Packets are received through input ports, forwarded to output queues, and sent through output ports. Each port receives/sends at most one packet in a time step, and multiple input ports can simultaneously forward packets to the same output queue. We assume packet forwarding takes negligible time. Packets are dropped if the destined queue is at its maximum size (10 packets in our example). Fig. 2 shows per-port input, output, and dropped counts, *i.e.,* the number of packets received, sent, and dropped respectively during a single interval of 20 time steps. Note that queues could store packets at the start of the monitoring interval, so the total count of sent and dropped packets can exceed the total count of received packets. The operator's query is: "Could port O_2's queues have a combined queue length of 7 packets at any time during this interval?" The operator would need to ask this query for every interval between 9 and 10 am. Next, we focus on the query for a single interval. The query is expressed in our query language as

$$\exists t \in [0,T].\ \sum_{q \in O_2} qlen(q,t) \geq 7 \qquad (1)$$

A naive observer would answer the query positively. If one were to consider port O_2 in isolation from the rest of the ports, one might think that there is a packet trace which results in a queue length of 7 packets and is consistent with its packet counts during the interval. Fig. 5 shows such a packet trace P_1, represented as a 2-D matrix with $P_1[q][t]$ showing the packets enqueued at queue q at time t, with packet colors denoting the input port each packet came from; blank entries denote no packets enqueued.

In this packet trace, O_2 receives no packets until the last 3 time steps, when it receives a sequence of 4, 4, and 2 packets. This causes the queue length to increase by 3, 3 and 1 in each time step since the port dequeues one packet per time step and the remaining packets are queued. Assuming port O_2 is initially empty, at the end of the interval, it has $3+3+1=7$ packets queued, satisfying the query. In addition, since O_2 receives packets only in the last 3 time steps, it dequeues a total of 3 packets during the interval, matching its sent count. Since port O_2 never reaches the maximum queue size of 10, it does not drop any packets, thereby matching its dropped count. However, this naive reasoning is erroneous.

[1] we have scaled down the interval to make the example easier to follow, but we have evaluated QUASI on an interval with 300 million time steps (§7.1).

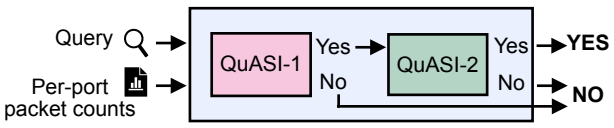


Figure 3: QUASI architecture.

The correct answer to the query is negative. In fact, there is no packet trace consistent with the observed measurements that will make the query true, *i.e.,* O_2's queue length is always less than 7 during the monitoring interval. The reason that the correct answer is negative though it appears positive at first glance is that the packet counts of other ports constrain the number of packets that could have been enqueued at O_2.

At a high level, answering queue-related queries is challenging because the queries involve quantities that are not directly measured and cannot be inferred by a simple analysis of the measurements. While a query metric (*e.g.,* queue length) is correlated with measured signals (packet counts) it also depends on information absent in packet counts, namely on the arrival sequence of packets entering the switch, and to which output queue each packet is forwarded.[2] This information is present in a packet trace, which specifies which port's packet was forwarded to which port and when. However, a huge number of packet traces could be consistent with given packet counts, and reasoning about all these packet traces is computationally expensive.

4 QUASI Design

This section describes the key concepts QUASI relies on and demonstrates their usefulness on the running example (§3.2).

4.1 QUASI's Layered Architecture

Due to the scalability limitations of an exact analysis, we use a layered approach consisting of two layers, QUASI-1 and QUASI-2, as shown in Fig. 3. QUASI-1 is a fast, over-approximate analysis which is guaranteed to never report a false negative. Hence, if QUASI-1 answers "No", QUASI reports "No" and terminates. If QUASI-1 answers "Yes", it could be a false positive, so the analysis proceeds to QUASI-2 which is a more expensive, exact analysis that reports the correct answer. QUASI's first layer is accurate enough to conclusively answer non-trivial queries in a fraction of a second, as we demonstrate in the evaluations (§7). By running the more expensive

[2] For context, Zoom2Net [15] does not deal with this challenge – it assumes there is a coarse-grained version of the signal of interest.

$$
\begin{aligned}
query &::= \{\exists|\forall\}\ t \in [T_1, T_2].\ cmp \\
cmp &::= lhs\ ?\ rhs \mid \bigwedge lhs\ ?\ rhs \\
lhs &::= \{\exists|\forall\} q \in Qs.\ m(q,t) \mid \textstyle\sum_{q\in Qs} m(q,t) \\
&\quad \mid m(Q_1,t) - m(Q_2,t) \\
m(q,t) &::= enq(q,t) \mid cenq(q,t) \mid qlen(q,t) \\
rhs &::= K \mid K.t
\end{aligned}
$$

Figure 4: Query language. T_1, T_2, K are constants, ? denotes a comparison operator, and Q_1, Q_2 denote queues in a switch.

second layer only on queries that cannot be answered conclusively by the fast first layer, QUASI can scale to *tens of thousands* of time steps. This is well beyond the capabilities of existing tools [4], which take several hours even for 10 time steps.

4.2 Query Language

A query in QUASI is a comparison (or conjunction of comparisons) involving a metric over one or more queues in a switch; Fig. 4 shows the syntax. We consider metrics (1) $enq(q,t)$: number of packets received by queue q at time step t, (2) $cenq(q,t)$: total number of packets received by queue q until time step t, and (3) $qlen(q,t)$: queue length (in packets) at queue q at time step t. Queries allow quantification over (bounded) time steps as well as queues. Our query language is expressive enough to allow interesting questions related to bursts, queue lengths, and buffer occupancy, including some queries that FPerf [4] does not support, *e.g.,* the Burst query used in our evaluation (§7). Our query language can be extended with interfaces to languages like Python or SQL to make it easier to use for operators unfamiliar with logic.

QUASI currently supports single-switch queries, but can also be used to answer some queries over multiple switches. For example, we can find the maximum queueing latency along a given network path using QUASI to find the maximum queueing latency at individual switches on the path and adding up these values. Using only QUASI-1, the answer will be an upper bound on the maximum latency along the path while QUASI-2 will report the true maximum latency.

4.3 Key Concepts

We present key concepts required to understand our approach.

Packet trace. A packet trace P is a $N_q \times T \times N$ $(0,1)$-matrix with entry $P[q][t][i]$ equal to 1 if input port i forwards a packet to queue q at time step t, and 0 otherwise. N denotes the number of switch input ports, N_q denotes the total number of output queues, and T denotes the number of time steps in the monitoring interval. Fig. 5 shows packet trace P_1 as a 2-D matrix with packet color denoting the input port it came from; blank entries are 0.

Enqueue-rate abstraction. We target queries that do not distinguish between packets in a queue based on their origin input ports. Thus, it is safe to ignore input port information in a packet trace. Our enqueue-rate abstraction defines an *abstract* packet trace that represents the *enqueue-rate*, the number of packets enqueued to each queue at each time step, ignoring the input port each packet came from. An abstract trace represents a set of packet traces, one packet trace for each labelling of packets with input ports. QUASI-2 uses the enqueue-rate abstraction to reduce the number of variables in its SMT formulation without any loss of precision due to abstraction, since our query language (Fig. 4) treats all packets in a queue as identical.

Abstract trace. An abstract trace A is a $N_q \times T$ matrix with entry $A[q][t]$ denoting the number of packets enqueued to queue q at time step t. An abstract trace represents a set of packet traces. We say packet trace P is "contained" in A iff $\forall q, t.\ \sum_{1\le i\le N} P[q][t][i] = A[q][t]$. Fig. 5 shows an abstract trace A_1 which contains both packet traces P_1 and P_2, which differ in the input ports of some packets but have the same number of packets enqueued at each queue at each time step.

Cover-set F_C. In QUASI-1, we use the enqueue-rate abstraction at a coarser granularity than in QUASI-2. The first module of QUASI-1 generates a cover-set $F_C = F_C^1 \vee \ldots \vee F_C^k$, a finite disjunction of *components* $F_C^1, \ldots, F_C^k$, formulas representing *sets* of abstract traces. Each component F_C^j is a conjunction of constraints involving enqueue-rate (enq); we do not enumerate the abstract traces satisfying F_C^j. A cover-set represents all packet traces consistent with the query and output counts and dropped counts, *i.e.,* $M_o \cap Q \subseteq C$, where C denotes the set of packet traces satisfying the cover-set formula F_C. Fig. 5 shows the cover-set that QUASI generates for our motivating example; it has only one component F_C^1 (*i.e.,* $k=1$).

Representative abstract trace. Each (non-empty) component F_C^j in a cover-set F_C has a representative abstract trace R_j that is "most-uniform" in the set of abstract traces C_j satisfying F_C^j. We prove that if R_j does not contain a (concrete) packet trace consistent with input counts, then the entire C_j has no packet trace consistent with input counts (Theorem 5.3). Representative abstract traces provide a correctness-preserving *reduction* in the search space: it is enough to consider only the representative R_j in each component. The second module of QUASI-1, MUC *constructs* a representative abstract trace R_j for a given F_C^j. For our motivating example, Fig. 5 shows the representative abstract trace R_1 for F_C^1.

Abstract trace consistency. We say abstract trace A is consistent with input counts (or is *input-consistent*) if there exists a labeling function Lbl that maps each packet in A to an input port such that: (1) packets at the same time step are assigned different input ports (an input port can forward at most one packet in a time step), and (2) each input port label i is used exactly $rcv[i]$ times, where $rcv[i]$ is port i's input count. We efficiently check input-consistency of each representative abstract trace by reducing it to checking the existence of a $(0,1)$-matrix with given row sums and column sums (Theorem 5.4) and using a known combinatorics result (Gale-Ryser theorem [13, 24]).

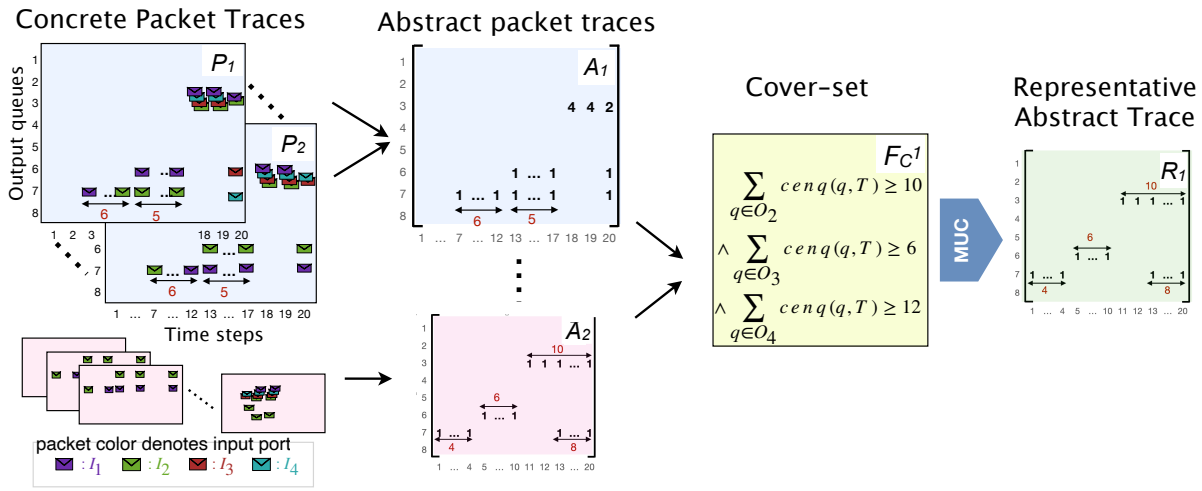


Figure 5: Multiple concrete packet traces (*e.g.,* P_1 and P_2) are mapped through the enqueue-rate abstraction to an abstract trace (*e.g.,* A_1). Multiple abstract traces (*e.g.,* A_1 and A_2) are represented by a cover-set (*e.g.,* F_C^1 with number of components $k=1$). The MUC constructs a most-uniform abstract trace which can be used as a representative in F_C^1 for checking input-consistency. For their analysis, QUASI-1 uses cover-sets and representative abstract traces, while QUASI-2 uses abstract traces.

4.4 QUASI on Motivating Example

We illustrate the key steps of QUASI on our motivating example (§3.2). While QUASI is composed of two layers, it reaches a conclusive negative answer already at the first layer, which is guaranteed to be correct. QUASI's first layer generates a formula F_C (the cover-set) that represents a set C containing all packet traces consistent with output counts, dropped counts, and the query, *i.e.,* $M_o \cap Q \subseteq C$.

$$F_C = \sum_{q \in O_2} cenq(q,T) \geq 10 \wedge \sum_{q \in O_3} cenq(q,T) \geq 6 \wedge \sum_{q \in O_4} cenq(q,T) \geq 12 \quad (2)$$

Eqn. (2) shows F_C in terms of the cumulative enqueue-rate $cenq(q,t)$, the number of packets enqueued at a queue q until time step t. F_C states that during the monitoring interval, output port O_2 must receive at least 10 packets, output port O_3 must receive at least 6 packets and output port O_4 must receive at least 12 packets.

Next, QUASI needs to determine whether each component F_C^i in F_C is consistent with input counts (here, $F_C = F_C^1$ with $k=1$), *i.e.,* whether $F_C^1 \cap M_i = \emptyset$. Note that F_C^1 itself represents a large set of packet traces, so checking each individual packet trace is impractical. QUASI constructs one representative abstract trace, denoted R_1, which satisfies F_C^1. R_1 specifies the number of packets enqueued over time at each queue (or each port, depending on the query). We prove that it suffices to check *only* R_1 for consistency with input counts: if R_1 does not contain a packet trace that is input-consistent, then no other abstract trace satisfying F_C^1 will contain a packet trace that is input-consistent (Theorem 5.3). Fig. 5 shows R_1 for our example.

To check R_1 for consistency with input counts, QUASI first checks if the total number of packets in R_1 exceeds the total number of packets received by the switch. If yes, R_1 is inconsistent with input counts; otherwise, QUASI proceeds to run an efficient matrix-based consistency check. For our example, the total number of packets in R_1 is 28, which is more than the total number of packets received (26). Hence, QUASI answers the query negatively, which is the correct answer for this example.

5 QUASI: Layer 1

QUASI-1 consists of three modules (Fig. 6):

1. **Cover-set generator (CSG)** generates a cover-set $F_C = F_C^1 \vee \ldots \vee F_C^k$ which represents all packet traces consistent with output and dropped counts that make the query true, *i.e.,* $M_o \cap Q \subseteq C$, where C denotes the set of packet traces satisfying F_C. Fig. 6 shows C with $k=4$ as a yellow circle superimposed on a Venn diagram showing the sets M_i, M_o, and Q from our problem formulation (§3.1).
2. **Most-uniform Abstract Trace Constructor (MUC)** constructs a representative abstract trace R_j (shown as a small red square in Fig. 6) that is *most-uniform* for each component F_C^j. We prove that it is enough to check *only* each R_j for consistency with input counts (Theorem 5.3).
3. **Matrix-based Consistency Checker (MCC)** determines whether a given representative abstract trace contains a packet trace that is input-consistent, *i.e.,* whether R_j intersects with M_i. We reduce this problem to checking the existence of a $(0,1)$-matrix with given row sums and column sums (Theorem 5.4) and use the Gale-Ryser theorem [13, 24] to solve it efficiently.

If some representative abstract trace R_j is found to be consistent with input counts, QUASI-1 answers the query positively. QUASI-1 answers the query negatively if every R_j is inconsistent with input counts.

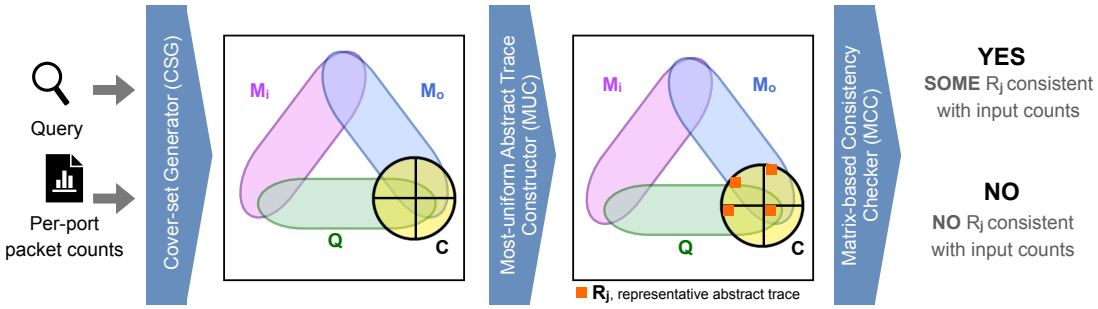


Figure 6: QUASI-1 design: (1) CSG generates a cover-set formula F_C representing a set $C = C_1 \cup ... \cup C_4$, with $C \supseteq M_o \cap Q$, (2) MUC constructs a representative R_j for each C_j, (3) MCC checks if $R_j \cap M_i = \emptyset$ for each R_j; if true, QUASI-1 reports "No", otherwise "Yes".

Theorem 5.1 (QUASI-1 correctness)**.** *If* QUASI-*1 answers a query negatively, there is no packet trace consistent with the measurements that makes the query true,* i.e., QUASI-*1 will never report a false negative.*

Proof. Assume QUASI-1 answers a query Q negatively. This means the MCC determined that each representative abstract trace R_j (for C_j) does not contain a packet trace that is consistent with input counts, *i.e.,* $R_j \cap M_i = \emptyset$. For each C_j, by Theorem 5.3, no other abstract trace in C_j is consistent with input counts, *i.e.,* $C_j \cap M_i = \emptyset$. Hence, there is no packet trace in $C = C_1 \cup ... \cup C_k$ which is consistent with input counts, *i.e.,* $C \cap M_i = \emptyset$. By Lemma 5.2, C contains all packet traces consistent with output and dropped counts that make the query true, *i.e.,* $M_o \cap Q \subseteq C$. Since $C \cap M_i = \emptyset$, it follows that $M_o \cap Q \cap M_i = \emptyset$. Hence, there exists no packet trace consistent with the measurements that makes the query true. □

Our proof depends on Lemma 5.2 and Theorem 5.3 – these are discussed in the related subsections.

False positives. If QUASI-1 answers a query positively, it is possible that no trace exists which is consistent with the measurements and query. This is because the cover-set generated by the CSG is an *over-approximation*, *i.e.,* it could include additional packet traces not consistent with output and dropped counts and/or the query. If one of these extra packet traces is consistent with input counts then QUASI-1 could incorrectly answer the query positively.

5.1 Generating a Cover-set

The CSG constructs symbolic representations (as formulas) of two sets: (1) Q', which contains all packet traces satisfying the query ($Q \subseteq Q'$) and (2) M'_o, which contains all packet traces consistent with output and dropped counts (*i.e.,* $M_o \subseteq M'_o$). It returns $F_C = F_C^1 \vee ... \vee F_C^k$, obtained by performing standard syntactical transformations on the formula for $M'_o \cap Q'$.

M'_o, necessary conditions for consistency with output and dropped counts. Each output port O_i must receive a minimum number of packets (across all its output queues) during the monitoring interval to be consistent with its output count (deq_i) and dropped count (drp_i).

$$\forall 1 \le i \le N. \sum_{q \in O_i} cenq(q,T) \ge deq_i + drp_i - ilen_ub_i \quad (3)$$

$ilen_ub_i$ is an upper bound on $ilen_i$, the number of packets queued at port O_i at the start of the interval. Note that $ilen_ub_i \le deq_i$ otherwise port O_i would dequeue more than its output count. Also, $ilen_ub_i \le L$, the maximum number of packets that can be queued at a port. Hence, we set $ilen_ub_i = min(L, deq_i)$. We can improve this bound further depending on the query, *e.g.,* if the query asks if the queue length can be always below some K.

Equation (3) is obtained from conservation of packets at an output port, assuming that output and dropped counts hold. By conservation of packets, the number of packets queued in the port at the end of the monitoring interval ($flen_i$) is equal to the difference between the number of packets received (or were queued initially) at the port during the interval and the number of packets that left the port (or were dropped) during the interval. Formally,

$$flen_i = ilen_i + cenq(O_i,T) - cdeq(O_i,T) - cdrp(O_i,T) \quad (4)$$

where $cenq(O_i, T)$, $cdeq(O_i, T)$ and $cdrp(O_i, T)$ denote the total number of packets enqueued, dequeued, and dropped respectively at port O_i from time step 0 until T. By consistency with output counts, $cdeq(O_i, T) = deq_i$; by consistency with dropped counts, $cdrp(O_i, T) = drop_i$. Using $flen_i \ge 0$ and rearranging, we get $cenq(O_i,T) \ge deq_i + drp_i - ilen_i \ge deq_i + drp_i - ilen_ub_i$. Observing that $cenq(O_i,T) = \sum_{q \in O_i} cenq(q,T)$, we get eqn. (3).

Q', necessary conditions to satisfy query.

Case 1: Query metric is *enq* or *cenq*. For queries involving the metrics enqueue-rate (*enq*) or cumulative enqueue-rate (*cenq*) the query Q is translated into an equivalent quantifier-free formula by eliminating quantifiers over queues and time steps to get Q', which is semantically equivalent to Q. Existential quantifiers are eliminated using disjunctions and universal quantifiers are eliminated using conjunctions.

Case 2: Query metric is *qlen*. Our necessary conditions need to use a different metric (*enq* or *cenq*) from the metric in

the query, so we cannot obtain Q' just by eliminating quantifiers as in the previous case. To relate queue length to enqueue-rate, we use conservation of packets (5) at the queue(s) in the query, setting $qlen(q,t)$ to an appropriate value depending on the query. We explain how we construct our necessary conditions for the query $QgeK : \exists t \in [st,end].qlen(q,t) \geq K$ which asks if queue q can have at least $K > 0$ packets at some time step $t \in [st,end]$; other queries are handled similarly.

By conservation of packets, the number of packets in the queue at time step t ($qlen(q,t)$) is equal to the difference between the number of packets received (or were queued initially) at the queue and the number of packets that left the queue (or were dropped) until t:

$$qlen(q,t) = il(q) + cenq(q,t) - cdeq(q,t) - cdrp(q,t) \quad (5)$$

where $il(q)$ denotes the number of packets in the queue at the start of the monitoring interval; and $cenq(q,t)$, $cdeq(q,t)$, $cdrp(q,t)$ denote the number of packets enqueued, dequeued, and dropped, respectively, at queue q from time step 0 until t. For the query to be true, $qlen(q,t) \geq K$ for some $t \in [st,end]$; substituting this in eqn. (5) we get

$$il(q) + cenq(q,t) - cdeq(q,t) - cdrp(q,t) \geq K$$
$$cenq(q,t) \geq K - il(q) + cdeq(q,t) + cdrp(q,t)$$

We replace each term on the right-hand-side by a lower/upper bound as appropriate: an upper bound for $il(q)$, and lower bounds for $cdeq(q,t)$ and $cdrp(q,t)$.

Upper bound on initial queue length. Note that $il(q) \leq L_q$, the maximum queue size. In some cases, we impose further constraints on the initial queue length to ensure consistency with the output count deq for q's port. For example, when $deq < T$, the number of time steps in the interval, and the output port has a single queue, the initial queue length must not exceed deq. Otherwise, the queue would dequeue more than deq packets, conflicting with the observed output count.

Lower bound on number of packets dequeued until t. We calculate the minimum number of packets that q will dequeue until t, assuming the query is true. For our example query $QgeK$ to be true, queue length at t must be at least $K > 0$. Depending on the value of K, the queue length must be non-zero for some minimum number of time steps $nz(K)$ before t, during which time it will dequeue at its minimum dequeue rate mdr. Hence, $cdeq(q,t) \geq nz(K) * mdr$. We calculate $nz(K)$ assuming the queue receives packets at the maximum possible rate (according to the input packet counts) for as long as possible until it reaches a queue length of K. We consider two (exhaustive) cases to determine the initial queue length (1) queue has a packet to send at every time step before t (queue length must become at least K at t starting from the maximum possible value of $il(q)$) and (2) queue has no packet to send at some time step before t (in this case, queue length must become at least K at t starting from a queue length of zero.)

Lower bound on number of packets dropped until t. We use $cdrp(q,t) \geq 0$. We use a tighter lower bound if the query specifies $st = end = T$ (which makes $t = T$) and queue q is the only queue in its port; then, $cdrp(q,T) = drp_i$, where drp_i is the dropped packet count for q's port.

Lemma 5.2 (CSG correctness)**.** *CSG returns a formula F_C representing a set C that contains all packet traces consistent with the query, output counts, and dropped counts,* i.e., $M_o \cap Q \subseteq C$.

Proof sketch. The set C represented by formula F_C is equivalent to $M'_o \cap Q'$ (by construction). M'_o is obtained from the packet conservation equation (4) assuming consistency with output and dropped counts, so any packet trace consistent with output and dropped counts will satisfy M'_o, *i.e.,* $M_o \subseteq M'_o$. For queries over *enq* or *cenq* metrics, Q' is obtained by eliminating quantifiers in the query Q to obtain an equivalent formula, so $Q' = Q$ in this case. For queries over $qlen$, Q' is derived starting from the packet conservation equation (5) (which is true for all packet traces) and enforcing bounds on terms assuming that the query is true. Hence, any trace that satisfies the query will also satisfy Q'. For all three metrics, $Q \subseteq Q'$. Since $M_o \subseteq M'_o$ and $Q \subseteq Q'$ it follows that $M_o \cap Q \subseteq M'_o \cap Q'$, so $M_o \cap Q \subseteq C$.

5.2 Computing Representative Abstract Traces

For each component F_C^j in the cover-set $F_C = F_C^1 \vee ... \vee F_C^k$ generated by the CSG, the MUC constructs a representative abstract trace R_j that satisfies F_C^j.

Representative is most-uniform. We characterize a representative abstract trace R_j satisfying F_C^j as being "most-uniform" among all abstract traces satisfying F_C^j. A most-uniform abstract trace has packets in columns of lowest possible *height* subject to satisfying F_C^j, where height of a column denotes the sum of its entries. Formally, for all $i \geq 1$, the total height of the i highest columns in R_j must be no more that the total height of the i highest columns in an arbitrary abstract trace A satisfying F_C^j.

Algorithm 2 (shown in Appendix A) takes as input F_C^j, a conjunction of constraints of three types: LB ($enq(q,t) \geq lb$), CLB ($cenq(q,t) \geq lb$), and CUB ($cenq(q,t) \leq ub$). It constructs an abstract trace R_j that is most-uniform by starting with the null matrix (with all entries 0) and incrementing appropriate matrix entries in order to satisfy the constraints. The procedure can be thought of as placing packets at appropriate time steps and queues in order to satisfy the given constraints. Importantly, the algorithm places each packet in a time step with lowest height (*i.e.,*, with minimum number of packets in total across all output queues) subject to upper bound constraints. This allocation strategy prioritizes placing packets in time steps that have few packets, making the resultant arrangement as uniform as possible subject to satisfying the given constraints.

Theorem 5.3 (Reduction via representative abstract trace)**.** *If the representative abstract trace R_j of a component F_C is not input-consistent, then every abstract trace A satisfying F_C^j is not input-consistent.*

Proof sketch. Consider an arbitrary A satisfying F_C^j. We show that if A is consistent with input counts then the output of Algorithm 2, R_j, must be consistent with input counts. Since A is consistent with input counts, by the definition of input-consistency, there exists some labeling function mapping each packet in A to an input port such that each input port i is used exactly $rcv[i]$ times (where rcv denotes the input packet counts) and packets at the same time step in A are mapped to different input ports. We show that is possible to transform A to R_j by shifting packets from one time step to another so that packets at the same time step continue to be mapped to different input ports after each shift. After performing all these shifts to reach R_j, the labels of packets in R_j will be a valid labeling; hence, R_j is consistent with input counts. The full proof is in Appendix B.

5.3 Checking Consistency with Input Counts

The Matrix-based Consistency Checker (MCC) checks if a given representative abstract trace R_j contains a packet trace that is consistent with input packet counts rcv. Recall that an abstract trace does not specify the input port each packet came from, and represents a set of packet traces, one packet trace for each labeling of packets with input ports.

There could be a huge number of possible labelings of packets in an abstract trace so it is impractical to iterate through all labelings to check if one is consistent with input counts. Constraint solving approaches will also have scalability limitations due to the large search space of labelings.

Reduction to $(0,1)$-matrix existence. We reduce the problem of determining if an abstract trace R_j contains a packet trace consistent with input counts to checking the existence of a $(0,1)$-matrix with given row sums and column sums. We leverage a known combinatorics result (the Gale-Ryser theorem [13, 24]) on matrix existence to solve it efficiently. The idea behind this reduction is that an abstract trace can be equivalently represented as a $(0,1)$-matrix with a row for each input port and a column for each time step. The given abstract trace R_j fixes the sum of each column and the input counts rcv fix the sum of each row of this $(0,1)$-matrix. R_j is consistent with input counts iff such a $(0,1)$-matrix exists for input counts rcv. We prove this in Appendix C.

Theorem 5.4 (Abstract Trace Consistency)**.** *An abstract trace R_j ($N_q \times T$ matrix) is consistent with input counts rcv iff There exists an $N \times T$ $(0,1)$-matrix X s.t.*

$$\forall i.\ rsum(X,i) = rcv[i] \text{ and } \forall t.\ csum(X,t) = csum(R_j,t)$$

where $rsum(X,\ i)$ and $csum(X,\ t)$ denote the sum of entries in row i and column t respectively of matrix X.

Algorithm 1 checks input-consistency of an abstract trace using the Gale-Ryser theorem [13, 24]. The Gale-Ryser theorem [13, 24] states that there exists a $(0,1)$-matrix with column sums c and row sums r, where c and r are partitions of a positive integer, if and only if r is dominated by c', where c' is the conjugate partition [3] of c.

Algorithm 1: Matrix-based consistency check

Input : R_j, abstract trace ($N_q \times T$ matrix)
rcv, input counts ($1 \times N$ matrix)
Output : True iff R_j has a packet trace satisfying rcv

```
1  Function check_consistency(R_j, rcv):
2      rt : 1 × T // Aggregate rate of R_j
3      for t = 0; t < T; t++ do
4          rt[t] ← Σ_{q∈N_q} R_j[q][t] // column sums
5      rt.sort(); rcv.sort()// descending
6      return matrix_exist(rcv, rt)
7  Function matrix_exist(r, c):
8      rsums[0] = r[0]// cumulative row sums
9      for i = 0; i < r.size(); i++ do
10         rsums[i] = rsums[i−1] + r[i]
11     c' ← conj(c)// conjugate partition of c
12     csums[0] = c'[0] // cumulative sums of c'
13     for i = 0; i < c'.size(); i++ do
14         csums[i] = csums[i−1] + c'[i]
15     for i = 0; i < r.size(); i++ do
16         if rsums[i] > csums[i] then
17             return False // c doesn't dominate r
18     return True
```

6 QUASI: Layer 2

QUASI-2 models switch operation at each time step using Satisfiability Modulo Theories (SMT) [5] constraints, with additional constraints enforcing the observed packet counts and the query. It uses the Z3 [11] solver to check if these constraints are satisfiable. If satisfiable, the query is true for some packet trace consistent with measurements; otherwise, the query can never be true for the given measurements.

Enqueue-rate abstraction enables N times fewer variables. QUASI-2 models a symbolic abstract trace which specifies the number of packets entering each output queue at each time step, requiring $O(N_qT)$ variables. In contrast, a symbolic packet trace (without enqueue-rate abstraction) specifies which input port sends a packet to which output queue at which time step and requires $O(NN_qT)$ variables.

Enqueue-rate abstraction is lossless for our query language, guaranteeing correctness of QUASI-2. Although the enqueue-rate abstraction ignores input ports of packets, it does not lose any information since our queries do not distinguish between packets in an output queue based on their input ports.

7 Evaluation

Our evaluation demonstrates QUASI 's usefulness by showing that QUASI can conclusively answer queries that cannot be

answered by our baseline, *i.e.,* a conservative heuristic analysis of the measurements. We go further than answering yes/no queries, extracting quantitative bounds on metrics like queue size and buffer occupancy from per-port packet counts. We show that QUASI 's bounds are significantly tighter than the bounds inferred by our baseline, demonstrating that QUASI can infer useful information about metrics (like queue size) that are not directly measured. We study how QUASI scales with the number of time steps in a monitoring interval and compare it with an existing performance analysis tool, Fperf [4].

Our measurements admit interesting scenarios. We collect per-port packet counts using the ns-3 simulator [22] with a star topology consisting of a switch with 8 ports, with hosts sending randomized UDP traffic. Each switch output port can queue a maximum of 250 packets and the total buffer size is 2000 packets. We ensure our collected measurements allow the possibility of packets accumulating in queues, which leads to interesting performance scenarios like bursts and increased queueing latency. We evaluate QUASI on a set of 25 monitoring intervals of duration 100 time steps, each interval corresponding to a different set of measurements. We use *example* to refer to one monitoring interval. We use a smaller interval ($T = 10$) for comparison with FPerf, and several intervals of increasing duration to evaluate QUASI's scalability.

We select relevant, non-trivial queries. Our queries are drawn from real-world use cases: (1) BurstOccurrence ("Could a burst of rate R and duration D occur sometime during the interval?") is useful for troubleshooting poor application performance, (2) QlenK ("Could the queue length at port O_i be at least K?") is useful for checking latency SLO compliance, (3) MaxQlen ("What is the maximum queue length at port O_i during the interval?") is useful for estimating queueing latency, and (4) MaxBuff ("What is the maximum buffer occupancy during the interval?") is useful for buffer provisioning. Our queries involve multiple performance metrics (queue length and enqueue-rate) and the MaxBuff query involves multiple queues. We demonstrate that our selected queries cannot be easily answered by a heuristic analysis of the measurements, implemented in our HEURISTIC baseline: HEURISTIC fails to answer many queries that QuASI conclusively answers.

We use a strong baseline. As a baseline, we implement a conservative analysis, HEURISTIC, that answers a query negatively if it conflicts with a set of heuristics/rules that must always hold. HEURISTIC incorporates several non-trivial rules based on switch operation, including accounting for ports not included in the query. For burst queries, it uses rules: (1) number of packets in burst cannot exceed total number of packets received by the switch, (2) rate of incoming packets to an output queue cannot exceed the number of input ports that receive packets during the interval, and (3) a burst cannot occur in an output port if the minimum number of packets it dequeues is larger than the port's output count. Note that HEURISTIC will never report false negatives but could report false positives.

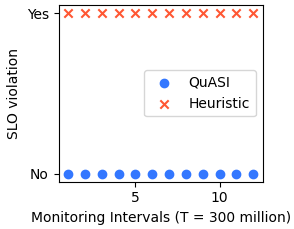


Figure 7: QUASI proves latency SLO compliance in each interval while HEURISTIC incorrectly reports an SLO violation.

7.1 Case Study: Checking SLO Compliance

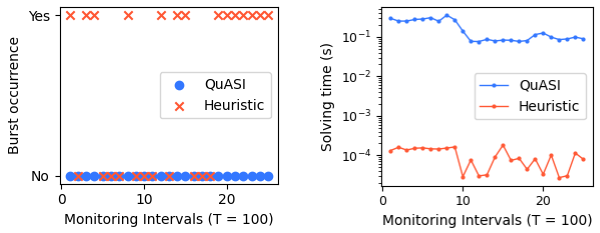
Consider, as in §3.2, a cloud operator who receives a complaint from a customer claiming the latency between the customer's servers, connected by a single switch, violated the Service Level Objective (SLO) of 289μs between 9 am and 10 am on a given day. The operator has SNMP [12] per-port packet counts over 5-minute intervals, which we collect via simulation using a switch with 8 ports, 8Gbps bandwidth, and no more than 2500 packets queued per output port. We generate randomized traffic using 1KB packets. Packet transmission time is 1μs, so a 5-minute interval has 300 million time steps.

To check for an SLO violation, we use the QlenK query which asks if a port could queue at least K packets, with K set to 290 (the SLO specifies no more than 289 queued packets). We run QUASI and HEURISTIC on each 5-minute interval in the specified hour. The results are shown in Fig. 7. HEURISTIC reports that the queue length was at least 290 packets during each interval, failing to prove SLO compliance. In comparison, QUASI reports that the queue length was always below 290 packets during each interval, thereby proving SLO compliance. QUASI-1 answers each query negatively so QUASI-2 is not run. QUASI took only 0.03s in total to prove SLO compliance, demonstrating that QUASI can scale to realistic (minute-granularity) monitoring intervals.

7.2 Burst Occurrence

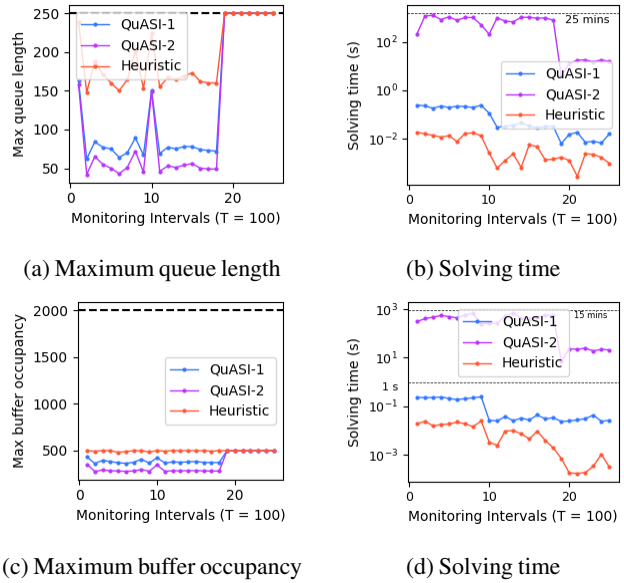
We evaluate QUASI on the BurstOccurrence query which asks if a burst of given rate and duration could occur in an interval. Fig. 8a shows the reported answer, with "yes" denoting a burst could have occurred and "no" denoting a burst is impossible. We have purposefully only included queries that could not have happened to see how often QUASI-1 would have a false positive and how long it would take for QUASI-2 to solve. Fig. 8b shows the solving time.

QUASI proves the absence of bursts in all intervals. QUASI-1 answers the query negatively for all examples, so we do not run QUASI-2 (since QUASI-1 will never report a false negative). HEURISTIC incorrectly reports that a burst is possible for 14 out of 25 examples, demonstrating that QUASI can conclusively negatively answer queries that a heuristic analysis cannot. Fig. 8b shows that QUASI is



(a) Burst occurrence (b) Solving time

Figure 8: QUASI proves the absence of bursts within 1s, while HEURISTIC incorrectly reports some bursts are possible.



(a) Maximum queue length (b) Solving time

(c) Maximum buffer occupancy (d) Solving time

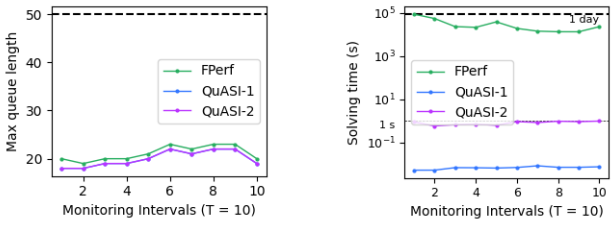
Figure 9: HEURISTIC results are consistently looser compared to QUASI. Even QUASI-1 (*i.e.,* the first layer alone) can provide tighter bounds within a fraction of a second.

efficient: QUASI-1 takes less than 1s for each example. Here we observe the benefit of a layered approach. Having a less accurate but much more scalable first layer removes the need for running the exact (but less scalable) analysis here.

7.3 Max Queue Length and Buffer Occupancy

We evaluate QUASI on queries MaxQlen and MaxBuff which ask the maximum queue length and buffer occupancy respectively during a monitoring interval.

Using QUASI to estimate quantitative bounds. We implement a binary search over the value of the metric (queue length/buffer occupancy), issuing a sequence of parameterized Boolean queries, each query asking if the value of the metric can be no less than a constant K. For MaxQlen, we use the Boolean query "$\exists t \in [1,T] \sum_{q \in O_i} qlen(q,t) \geq K$?, which can be read as "Can port O_i ever have at least K packets queued at some time during the monitoring interval?", where O_i is a specified port and K is a parameter. We initialize K to L_i, the maximum



(a) Maximum queue length (b) Solving time

Figure 10: QUASI is up to 10^6x faster than FPerf for MaxQlen.

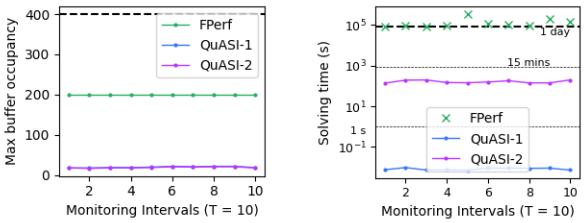
queue size at port O_i. The answer reported by QUASI (yes/no) determines how K is varied (increased or decreased) in the next Boolean query. If QUASI says "No", it means the queue length is always below K during the interval. Hence, we decrease the value of K in the next query. This process continues until the binary search terminates (QUASI zeroes in on a single value of K) or the minimum/maximum limit of the metric is reached. The final value of K is returned as QUASI's estimate of the maximum possible value of the metric for the given packet counts.

As QUASI-1 will never report a false negative, but could report a false positive, it over-estimates the value so that the final answer is an upper bound on the true maximum value. QUASI-2, which is an exact analysis, will report the correct answer to each Boolean query, so its final answer will be the true maximum value. We use the answer returned by QUASI-2 as ground truth to evaluate the accuracy of QUASI-1.

Fig. 9 shows the reported estimates of maximum queue length and buffer occupancy (lower is better as the estimates are upper bounds) and solving time for 25 monitoring intervals (each interval is a new set of measurements). We independently evaluate QUASI's two layers to show their difference in running time and the effect of false positives in QUASI-1.

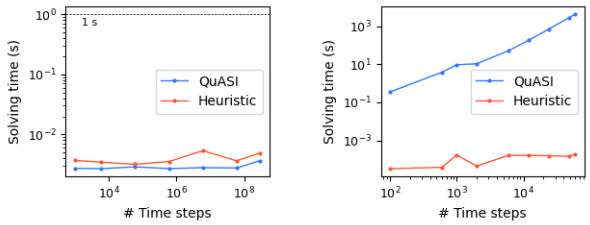
QUASI finds upper bounds within a second and the exact answer within 25 minutes for both metrics. Thanks to its layered approach, QUASI quickly finds upper bounds for both metrics (with avg. relative error of 0.25) within a second for each interval. QUASI finds the true maximum value over each interval within 25 minutes for queue length, and within 15 minutes for buffer occupancy.

QUASI-1 finds bounds up to 58% tighter than bounds found by HEURISTIC. Figs. 9a and 9c show that for both metrics, QUASI-1 reports bounds (shown in blue) that are consistently lower than bounds found by HEURISTIC (shown in orange), demonstrating that QUASI's approximate layer provides valuable insights that cannot be obtained by a heuristic analysis of the measurements. Our implementation of HEURISTIC is fair: its estimates are significantly below the trivial upper bounds of maximum queue size (250) and total buffer size (2000) shown by black dashed lines in Figs. 9a and 9c.



(a) Maximum buffer occupancy (b) Solving time

Figure 11: FPerf fails to compute the maximum buffer occupancy within a day while QUASI takes less than 15 minutes.



(a) QlenK (b) BurstOccurrence

Figure 12: QUASI takes almost constant time for QlenK and scales quadratically with interval size for BurstOccurrence.

7.4 Comparison with FPerf

To evaluate FPerf on our problem, we disable its workload synthesis step and stop after FPerf checks if some trace consistent with the base workload satisfies the query. We specify a base workload that enforces input packet counts (FPerf does not allow constraints over dequeued packets, so we do not use output and dropped packet counts). We extend FPerf to answer our quantitative queries MaxQlen and MaxBuff. FPerf does not support our BurstOccurrence query. We reduce the monitoring interval to just 10 time steps (from 100 steps in preceding subsections) so that Fperf terminates sometimes. We report FPerf's estimates after running it up to $\approx$ 1 day for each interval.

QUASI computes the maximum queue length 10^6X faster than FPerf. FPerf takes an average of 8.5 hours to find the maximum queue length while QUASI takes less than a second. The answers reported by FPerf and QUASI match when FPerf terminates within a day (*i.e.,* for 9 out of 10 intervals); the difference of 1 in Fig. 10a is because FPerf calculates queue length before dequeueing a packet rather than after.

FPerf does not complete all per-interval required queries in 1 day, reporting an upper bound $\approx$9 times larger than the true maximum buffer occupancy; QUASI finds the exact answer in under 15 minutes. We left FPerf running for more than 2 days for one example but that did not improve its estimate. As shown by the green curve in Fig. 11a, FPerf finds a very loose upper bound; the purple curve shows the exact answer found by QUASI-2. QUASI-1 is 100% accurate on all these examples, reporting the same answer as QUASI-2 (which we regard as ground truth).

7.5 Scalability

QUASI 's verification time increases as the number of time steps in the monitoring interval increases. This is because a larger number of time steps leads to larger abstract trace matrix dimensions in QUASI-1 and more variables in QUASI-2's SMT encoding. Increased switch bandwidth (which decreases packet transmission time) and longer monitoring intervals both lead to more time steps in the monitoring interval. We evaluate QUASI's scalability with the number of time steps in the monitoring interval for two queries: QlenK and BurstOccurrence; and report solving time in Fig. 12. QUASI-1 answers all queries negatively, so QUASI-2 does not run.

QUASI's solving time scales quadratically with number of time steps for BurstOccurrence. For the BurstOccurrence query, QUASI generates a cover-set with $O(T)$ components, and takes about 75 minutes for 60,000 time steps. For QlenK, QUASI's solving time remains about the same as the monitoring interval size is increased, answering queries for intervals with millions of time steps within 1 sec. This is because CSG detects that the generated cover-set does not contain any packet traces, so subsequent components in QUASI-1 are skipped. These queries are not trivial: HEURISTIC incorrectly answers all of them positively.

QUASI scales to realistic monitoring intervals (5 mins) for queue length queries. QUASI answers the queue length query negatively within a second for an interval with 300 million time steps (*i.e.,* 5 minutes, as packet transmission time is 1μs). HEURISTIC answers the query positively, showing again that QUASI can quickly answer non-trivial queries.

8 Conclusion

This paper presents QUASI, a layered approach to answering queue-related queries using coarse-grained packet counts. QUASI is centered around the novel and lossless enqueue-rate abstraction. The first layer of QUASI performs an over-approximate analysis using representations of packet traces as $(0,1)$-matrices. The second layer utilizes an exact but less scalable analysis based on use of an SMT solver. QUASI is 10^6X faster than state-of-the-art while answering non-trivial queries about queue metrics.

Acknowledgments

We would like to thank the NSDI reviewers and our shepherd for their comments. We are grateful to Prof. Matt Weinberg for helpful feedback in the early stages of this work. This work was supported in part by NSF Grants CNS-2319442 and CNS-2312539.

References

[1] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. ABM: Active Buffer Management in Datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.

[2] Vamsi Addanki, Wei Bai, Stefan Schmid, and Maria Apostolaki. Reverie: Low pass {Filter-Based} switch buffer sharing for datacenters with {RDMA} and {TCP} traffic. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 651–668, 2024.

[3] George E Andrews. *The theory of partitions*. Cambridge university press, 1998.

[4] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal Methods for Network Performance Analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 645–661, 2023.

[5] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[6] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. SIGCOMM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[8] Armin Biere and Daniel Kröning. Sat-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018.

[9] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 15–29, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[12] Mark Fedor, Martin Lee Schoffstall, James R. Davin, and Dr. Jeff D. Case. Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[13] David Gale. A theorem on flows in networks. In *Classic Papers in Combinatorics*, pages 259–268. Springer, 1956.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019.

[15] Fengchen Gong, Divya Raghunathan, Aarti Gupta, and Maria Apostolaki. Zoom2Net: Constrained Network Telemetry Imputation. In *Proceedings of the ACM SIGCOMM 2024*, 2024.

[16] In-Band Network Telemetry A Powerful Analytics Framework for your Data Center.

[17] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In *Handbook of Model Checking*, pages 447–491. Springer, 2018.

[18] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–8, 2018.

[19] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202–208, 2009.

[20] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 516–529, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Nick McKeown, Guido Appenzeller, and Isaac Keslassy. Sizing router buffers (reduxß). *ACM SIGCOMM Computer Communication Review*, Oct. 2019.

[22] Ns3 network simulator. https://www.nsnam.org/.

[23] Paula Roquero and Javier Aracil. On performance and scalability of cost-effective snmp managers for large-scale polling. *IEEE Access*, 9:7374–7383, 2021.

[24] H. J. Ryser. Combinatorial properties of matrices of zeros and ones. *Canadian Journal of Mathematics*, 9:371–377, 1957.

[25] Jurgen Schonwalder, Aiko Pras, Matus Harvan, Jorrit Schippers, and Remco van de Meent. Snmp traffic analysis: approaches, tools, and first results. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 323–332. IEEE, 2007.

[26] Yang Zhou, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong. Evolvable network telemetry at facebook. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 961–975, 2022.

A MUC Algorithm

The MUC uses Algorithm 2 to construct a most-uniform abstract trace for a given cover-set component.

Algorithm 2 takes as input a cover-set component, consisting of three types of constraints, and the total number of packets received in the interval (P). It then constructs a representative abstract trace starting from an empty abstract trace (with no packets) and iteratively placing packets to be enqueued at appropriate output queues at specific time steps in order to satisfy the given constraints.

First, lower bound (LB) constraints are satisfied (lines 2 to 4); the **get_lb** function returns the lower bound lb given queue q and time step t. The algorithm sets $R[q][t]$ to lb; this is the minimum number of packets needed to satisfy this constraint.

Second, cumulative lower bound (CLB) constraints are satisfied (lines 6 to 11). Note that there are many ways to distribute packets in an interval $[1,t]$ to satisfy such a constraint of the form $cenq(q,t) \geq lb$. Algorithm 2 uses the **fill** function, which places packets until the given lower bound constraint is satisfied, or until upper bound constraints prevent filling additional packets. Each packet is placed at time step $place \in ts$ which has the minimum number of packets in total (across output queues); ts is the set of time steps in $[1,t]$ that can take at least one packet without violating upper bounds. This allocation strategy prioritizes placing packets in time steps that have few packets, making the resultant arrangement as uniform as possible subject to satisfying the given constraints.

If the **fill** function returns abstract trace R that does not respect the lower bound lb, it means the upper bound constraints did not allow placing any more packets. In this case, there is no way to satisfy the lower bound constraint while respecting the upper bound constraints, so allocation fails and the algorithm returns *False*.

Algorithm 2: MUC

```
Input  : LB, lower bounds at given time steps
         CLB, lower bounds until given time steps
         CUB, upper bounds until given time steps
         P, total number of packets received
Output : R_j, most-uniform abstract trace
         for given constraints; False if it does not exist
1  int R[N_q][T] // Empty abstract trace
2  for t,q ∈ LB do
3      lb ← get_lb(LB,t,q)
4      R[q][t] ← lb
5  time_pts ← get_sorted_time_pts(CUB,CLB)
6  for t in time_pts do
7      for q ∈ [1,N_q] do
8          lb ← get_lb(CLB,t,q)
9          R ← fill(R,t,q,lb,CUB,N)
10         if tot(R,t,q) < lb then
11             return False
12 R ← fill(R,T,any_q,P,CUB,N)
13 if tot(R,T) > P then
14     return False
15 return R
16 Function fill (R, end, q, lb, CUB, N):
       // Ensure Σ_{t=1}^{end} R[q][t] ≥ lb
17     while tot(R,end,q) < lb do
18         ts ← ts_with_space(R,end,CUB)
19         if |ts| = 0 then
20             break
21         place ← minHeightIndex(ts)
22         R[q][place] ← R[q][place] + 1
23     return R
```

After satisfying CLB constraints, the algorithm fills any remaining packets to make the total number of packets in R equal to P (line 12); the argument any_q indicates the packets can be filled in any queue subject to satisfying the upper bound (CUB) constraints. If the total number of packets exceeds P, there are insufficient packets to satisfy the constraints and the algorithm returns *False* (line 14). Otherwise, R is returned.

If Algorithm 2 returns an abstract trace, it is most-uniform. The algorithm ensures that its output R satisfies the given constraints (*i.e.,* the cover-set component). The **fill** function always places a packet in a time step with the fewest possible packets subject to satisfying the constraints, hence ensuring the output R is most-uniform. The algorithm does not return an abstract trace if the given cover-set component is unsatisfiable, *i.e.,* if the lower bound constraints cannot be satisfied along

with the upper bound constraints or if the number of packets received P is insufficient to satisfy the given constraints.

B MUC Correctness

We present the proof of Theorem 5.3, which states that if the representative abstract trace R_j of a component F_C is not input-consistent, then every abstract trace A satisfying F_C^j is not input-consistent.

We begin by defining some notation. For an abstract trace A, $csum(A, t_1)$ denotes the sum of entries in column t_1 of A. We will also use the term "height of t" to mean the sum of entries at column t. We use R_j to denote the most-uniform abstract trace returned by Algorithm 2 for constraints F_C^j. For an abstract trace A, $ht_A \in \mathbb{N}^T$ denotes the row vector of column heights (where height of a column is the sum of its entries), sorted in non-increasing order, *i.e.,* $ht_A[1] \geq ... \geq ht_A[T]$. We use ht_{R_j} to denote the vector of sorted column heights for R_j. We use $v[:i]$ to denote the vector of the first i entries of v; and $\sum v[:i]$ to denote the sum of the first i entries in v.

Next, we define a transformation of an abstract trace, called a balancing shift, which preserves input-consistency.

Definition B.1 (Balancing shift)**.** *For an abstract trace A, a balancing shift moves a packet from time step t_1 to time step t_2 where $csum(A, t_1) > csum(A, t_2)$.*

A balancing shift moves a packet from a column of higher height to a column of lower height. A balancing shift from t_1 to t_2 requires that the height of t_1 is larger than the height of t_2, and will never increase the height difference between t_1 and t_2.

Lemma B.2 (Balancing shifts preserve input-consistency)**.** *For any abstract trace A, let A' be the abstract trace obtained by performing a balancing shift on A. If A is input-consistent, then A' must also be input-consistent.*

Proof. Assume that A is input-consistent. Then there is a labeling Lbl that associates each packet in A with an input port such that packets at the same time step in A are mapped to distinct input ports. Let the balancing shift be from t_1 to t_2 where $csum(A, t_1) > csum(A, t_2)$. Since column t_1 has at least one more packet than column t_2 and packets at the same time step are mapped to different input ports, it is always possible to choose a packet p at t_1 that has an input port distinct from all packets at t_2. (If this were not the case, it would mean that at least two packets at t_1 were mapped to the same input port, which is a contradiction). Shift packet p from t_1 to t_2 to obtain abstract trace A'. The balancing shift did not change the number of input port labels, so A' will have each input port label appearing as many times as the port's input count. Moreover, A' has packets at each time step mapped to distinct input ports by Lbl, so A' is input-consistent. □

We now show that the abstract trace R_j returned by Algorithm 2 is the "most-uniform" in C_j, the set of abstract traces satisfying a cover-set component F_C^j.

Lemma B.3. *The abstract trace R_j returned by Algorithm 2 is the most-uniform in C_j,* i.e.,*for any abstract trace $A \in C_j$, the total height of the i largest columns in R_j must be no more than the total height of the i largest columns in A, for any $i \geq 1$.*

$$\forall A \in C_j.\ \forall i \geq 1.\ \sum ht_{R_j}[:i] \leq \sum ht_A[:i] \tag{6}$$

Proof. The proof is by contradiction. Assume R_j is not most-uniform (*i.e.,* eqn. (6) does not hold). Then, there exists some $A \in C_j$ and some $i \in [1,T]$ such that

$$\begin{aligned} &\sum ht_{R_j}[:i] > \sum ht_A[:i]\ \text{, and} \\ &\forall k < i.\ \sum ht_{R_j}[:k] \leq \sum ht_A[:k] \end{aligned} \tag{7}$$

Note that $ht_{R_j}[i] > ht_A[i]$; otherwise, $\sum ht_{R_j}[:i]$ could not be greater than $\sum ht_A[:i]$. Let h denote the value of $ht_A[i]$. Consider the total number of packets that A and R_j pack in columns of height no more than h.

Let P denote the total number of packets in A and R_j.

$$P = \sum ht_A = \sum ht_{R_j} \tag{8}$$

Then, A packs more than $P - \sum ht_A[:i]$ packets in columns of height $\leq h$. Since $ht_{R_j}[i] > ht_A[i]$ and ht_{R_j} is sorted in non-increasing order, R_j packs no more than $P - \sum ht_{R_j}[:i]$ packets in columns of height $\leq h$. From eqn. (7), $\sum ht_{R_j}[:i] > \sum ht_A[:i]$, so R_j packs fewer packets than A in columns of height $\leq h$. Algorithm 2 constructs R_j by placing each packet at the minimum possible height subject to satisfying the given constraints. This allocation strategy packs as many packets as possible in columns of low height. Since A satisfies the constraints F_C^j, the total number of packets in columns of height $\leq h$ in A must be no more than the corresponding number of packets in R_j. Assuming there exists an $A \in C_j$ such that R_j is less uniform than A (eqn. (7)) implies that A packs strictly more packets than R_j in columns of height $\leq h$, which is a contradiction. □

We now present the proof of Theorem 5.3. We show that if some $A \in C_j$ is input-consistent, then the most-uniform abstract trace R_j must also be input-consistent. We define a procedure `Relabel` that transforms the labeling function f_L for A to construct a valid labeling function for R_j, thereby showing that R_j is input-consistent.

Proof. Assume there is some $A \in C_j$ which is input-consistent. Let f_L denote the labeling function which assigns each packet in A to an integer denoting an input port. Function f_L satisfies two conditions: (1) number of input port labels is equal to the port's input count; and (2) packets at the same column in A are mapped to distinct input port labels.

We define a procedure `Relabel` which transforms the labeling function f_L into a valid labeling function f'_L for the most-uniform abstract trace R_j. `Relabel` operates on the sorted height vectors ht_A and ht_{R_j} for A and R_j respectively.

`Relabel` consists of T steps. In step i, `Relabel` sets entry i of ht_A equal to entry i of ht_{R_j}; we use $ht'_A[i]$ to denote the value of $ht_A[i]$ after modification. At each step, `Relabel` stores the number of extra packets $excess[i] = \max(0, ht_A[i] - ht_{R_j}[i])$. If $ht_A[i] < ht_{R_j}[i]$ it uses the extra packets so far to make $ht_A[i]$ equal to $ht_{R_j}[i]$ (we will show that there will always be enough extra packets). After T steps, each entry of ht'_A will be equal to the corresponding entry of ht_{R_j}, *i.e.,* $ht'_A = ht_{R_j}$ at the end of the procedure.

We will now show that `Relabel` transforms ht_A into ht_{R_j} by only using balancing shifts.

By Lemma B.3, $\sum ht_{R_j}[:i] \leq \sum ht_A[:i]$ for all $i \in [1,T]$. In step 1, we will not have a deficit of packets since $ht_{R_j}[1] \leq ht_A[1]$.

Consider any step $i \geq 2$ in which there is a shortage of packets in ht_A, *i.e.,* where $ht_A[i] < ht_{R_j}[i]$. Since $\sum ht_A[:i] \geq \sum ht_{R_j}[:i]$ and $\sum ht_A[:i-1] \geq \sum ht_{R_j}[:i-1]$, it is guaranteed that the total excess packets so far, $ht_A[:i-1] - \sum ht_{R_j}[:i-1]$, is no less than the deficit at i, $ht_{R_j}[i] - ht_A[i]$.

$$\sum ht_A[:i] \geq ht_{R_j}[:i] \tag{9}$$

$$\sum ht_A[:i-1] + ht_A[i] \geq ht_{R_j}[:i-1] + ht_{R_j}[i] \tag{10}$$

$$\sum ht_A[:i-1] - \sum ht_{R_j}[:i-1] \geq ht_{R_j}[i] - ht_A[i] \tag{11}$$

Hence, we will be able to make entry $ht_A[i]$ equal to $ht_{R_j}[i]$ by using some of these extra packets.

Any extra packet is from some entry $t < i$ where $ht_A[t] > ht_{R_j}[t]$ originally (*i.e.,* before `Relabel` changes ht_A). Given that there is a deficit at entry i in A and both ht_A and ht_{R_j} are sorted in non-increasing order,

$$ht_A[t] > ht_{R_j}[t] \geq ht_{R_j}[i] > ht_A[i] \tag{12}$$

which implies that $ht_A[t] > ht_A[i]$. Hence, shifting an extra packet to i is always from a column with greater height to a column of lower height, *i.e.,* it is always a balancing shift.

Since the only moves that `Relabel` uses are balancing shifts, by Lemma B.2, it is always possible to choose an extra packet to shift from entry t to entry i such that entry i will have each packet mapped to a distinct input port. After `Relabel` terminates, the resultant vector ht'_A will have each entry i mapped to $ht'_A[i]$ distinct input port labels. Since a balancing shift does not change any input port label (it just moves labeled packets), ht'_A will be labeled with each input port a number of times equal to the port's input count. Since $ht'_A = ht_{R_j}$, each entry in ht'_A corresponds to some column in matrix R_j. Using the labeling over ht'_A, associating each entry with its corresponding column in R_j, we obtain a labeling that associates each packet in abstract trace R_j with an input port label satisfying the requirements of a valid labeling function. Hence, R_j is input-consistent. □

C MCC Correctness

We present the proof of Theorem 5.4, which reduces the problem of checking input-consistency of an abstract trace to checking the existence of a $(0,1)$-matrix with given row sums and column sums.

Proof. $\Rightarrow$: Assume abstract trace R_j is consistent with input counts rcv. Then, there exists some labeling Lbl that associates each packet in R_j with an input port. We construct a $N \times T$ $(0,1)$-matrix X from R_j and Lbl with each entry $X[i][t]$ defined as follows:

$$X[i][t] = \begin{cases} 1 & \text{if } Lbl \text{ uses port } i \text{ at } t \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Since Lbl uses each port i exactly $rcv[i]$ times, X will have exactly $rcv[i]$ 1s in row i, *i.e.,* the sum of row i will be $rcv[i]$ for all $i \in [1,N]$. Since Lbl labels all packets at the same time step in R_j with different ports, for each $t \in [1,T]$ the t^{th} column of X will have $csum(R_j, t)$ 1s, *i.e.,* $csum(X, t) = csum(R_j, t)$. Hence, R_j being input-consistent implies there exists a $(0,1)$-matrix X with its row sums determined by rcv and its column sums determined by R_j.

$\Leftarrow$: Assume there exists a $(0,1)$-matrix X with its row sums equal to the input counts rcv and its column sums equal to the column sums of abstract trace R_j. We construct a function Lbl from X. Let $AvlPorts(t)$ denote the set of indices of rows which have 1 in the t^{th} column of X. For each $t \in [1,T]$, let Lbl pick any $1-1$ mapping from $AvlPorts(t)$ to packets in the t^{th} column of R_j. Since $csum(X,\ t) = csum(R_j,\ t)$, $|AvlPorts(t)| = csum(R_j,\ t)$ so each packet enqueued at t can be labeled with a distinct input port. Since Lbl only uses ports from $AvlPorts(t)$ in each time step t (*i.e.,* only ports corresponding to entries of 1 in X for that time step), Lbl will use port i the number of times row i has entry 1 in X. Since $rsum(X,\ i) = rcv[i]$ for all $i \in [1,N]$ Lbl will use each port i exactly $rcv[i]$ times. Since Lbl satisfies the required properties in the definition of input-consistency and labels each packet in R_j, R_j is consistent with input counts rcv. □

Runtime Protocol Refinement Checking for Distributed Protocol Implementations

Ding Ding, Zhanghan Wang, Jinyang Li, Aurojit Panda
NYU

Abstract

Despite significant progress in verifying protocols, services that implement distributed protocols (we refer to these as DPIs in what follows), e.g., Chubby or Etcd, can exhibit safety bugs in production deployments. These bugs are often introduced by programmers when converting protocol descriptions into code. This paper introduces *Runtime Protocol Refinement Checking* (RPRC), a runtime approach for detecting protocol implementation bugs in DPIs. RPRC systems observe a deployed DPI's runtime behavior and notify operators when this behavior evidences a protocol implementation bug, allowing operators to mitigate the bugs impact and developers to fix the bug. We have developed an algorithm for RPRC and implemented it in a system called Ellsberg that targets DPIs that assume fail-stop failures and the asynchronous (or partially synchronous) model. Our goal when designing Ellsberg was to make no assumptions about how DPIs are implemented, and to avoid additional coordination or communication. Therefore, Ellsberg builds on the observation that in the absence of Byzantine failures, a protocol safety properties are maintained if all live DPI processes correctly implement the protocol. Thus, Ellsberg checks RPRC by comparing messages sent and received by each DPI process to those produced by a simulated execution of the protocol. We apply Ellsberg to three open source DPIs, Etcd, Zookeeper and Redis Raft, and show that we can detect previously reported protocol bugs in these DPIs.

1 Introduction

Distributed protocols, implemented in lock services [11, 21, 100], distributed databases [85, 99], etc., lie at the heart of all fault-tolerant applications. Ensuring that these services, which we refer to as *distributed protocol implementations* (DPIs), correctly implement distributed protocols remains challenging. Popular DPIs have shipped with application visible bugs: *e.g.,* prior versions of Etcd [21] would return stale values in some scenarios due to a bug [25] in its implementation of a linearizable read protocol [95]. Similarly, prior versions of Zookeeper [100] would, in some scenarios, lose updates during leader elections due to a protocol implementation bug [101]. Both systems are widely used, and both bugs can result in application bugs such as data corruption.

In this paper we develop an approach, *runtime protocol refinement checking* (RPRC), that notifies administrators when bugs such as the ones above occur, allowing administrators to mitigate the bug's effects and programmers to fix the bug. We started developing RPRC because we observed that despite the widespread use of verification tools [3, 4, 10, 41, 42, 55, 60, 65, 68, 90–92, 94] that prove the safety of distributed protocols, and testing tools [6, 15, 35, 41–43, 48, 66] that check implementation correctness, deployed DPIs continue to exhibit bugs. This is because ensuring that programmer correctly implements a verified protocol is challenging, and detecting (and eliminating) all bugs with testing is infeasible. Recently, IronFleet [34], Verus [47], Goose [12] and others, have proposed using static refinement proofs to connect verified protocols to implementations. However, to ensure feasibility (of refinement proof generation and checking) these frameworks must limit how code is written, making it hard for developers to produce verified, high-performance DPIs.

RPRC provides an alternate approach for connecting statically verified protocols to an implementation: RPRC systems observe a DPI's runtime behavior, and notify administrators when the implementation's behavior deviates from what is required by the protocol. Analyzing runtime behavior allows RPRC tools to avoid making assumptions about how the protocol is implemented or what libraries are used. Indeed, *Ellsberg*, the RPRC system described in this paper, treats the DPI as a blackbox, assumes no access to the DPI's internal state, and does not impose any limits on how DPIs are implemented. Note, RPRC systems do not replace static protocol and implementation verification [10, 34, 47, 49, 55, 68, 88, 91, 92] (they cannot statically generate proofs to show that a protocol is correct, or that an implementation refines the protocol) and fuzz testing tools [6, 41–43, 48, 66] (they do not try to maximize coverage and find all bugs before an implementation is deployed), but rather complement them.

This paper describes an RPRC algorithm that we have implemented in a system called Ellsberg. We designed Ellsberg to work with existing unmodified DPIs, to minimize communication and processing overheads, and to ensure that it had no effect on a DPI's failure guarantees. These design goals allow Ellsberg to be used in deployment and to find bugs that were missed during testing.

Ellsberg (Figure 1) consists of a set of co-located instances that run alongside (that is, co-located with) each DPI process. An Ellsberg instance has access to a trace of incoming and outgoing messages sent by its colocated process, but has no access to its internal state. Furthermore, Ellsberg instances are designed to work without communicating

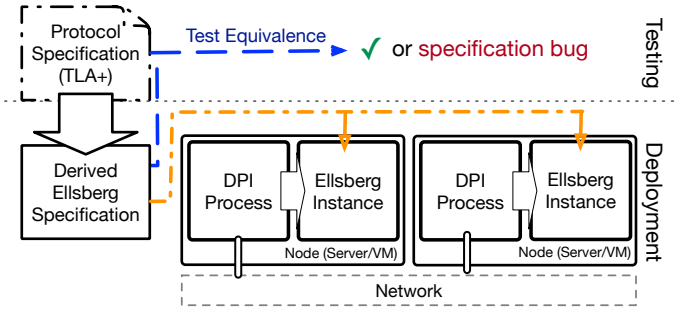


Figure 1: *An overview of* Ellsberg*, which runs an instance colocated with each DPI process, and compares the colocated process's behavior to a correct protocol implementation. We assume the TLA+ protocol specification is provided by the protocol designer, and programmers derive a* Ellsberg *specification from this specification.*

with each other: each instance processes the trace produced by its colocated process and can independently generate a notification when it observes an implementation bug. Our design builds on the observation that a DPI correctly implements a distributed protocol if and only if all processes executing the DPI have correctly implemented the protocol, allowing us to check implementation correctness without requiring additional communication or coordination. In cases where the protocol has been proven to maintain global safety properties (*e.g.,* agreement or linearizability), Ellsberg's local checks are sufficient to ensure that administrators are notified when these properties are violated.

Administrators and programmers use Ellsberg (Figure 1) as follows: before deployment, the administrator (or DPI programmer) translates the protocol the DPI purportedly implements into an Ellsberg specification and tests it (§4) to check correctness. When deploying the DPI, the administrator colocates an Ellsberg instance, configured with protocol specification, with each DPI process, and configures the DPI process so that the colocated instance has access to a trace of all messages received or sent by the process (§3.1). Each instance uses the specification and trace to simulate a correct implementation of the protocol and reports a bug if the process's behavior diverges from the simulated behavior.

We needed to address two challenges when developing Ellsberg: (a) **Concurrency within DPI processes.** Many DPI implementations are concurrent, and to avoid false notifications, Ellsberg needs to ensure that its simulation state corresponds to the DPI's protocol state. However, the DPI's protocol state depends on the order in which messages are processed, and the trace available to Ellsberg does not reveal this order. Consequently, Ellsberg instances need to consider all simulation states reachable by reordering messages in the trace, which can grow exponentially over time. Therefore, to be feasible Ellsberg needs to efficiently explore the set of reachable states; and (b) **Incremental checking,** by which we mean the ability to check DPI correctness without needing to process the entire trace or requiring access to messages sent or received in the future. Incremental checking prevents us from using existing approaches for efficiently processing multiple simulation states: these approaches assume access to a complete trace (including future messages), or assume that messages are annotated with a vector clock [78, 80] that records processing order.

Ellsberg addresses these challenges using a novel incremental RPRC algorithm (§3.4): each Ellsberg instance maintains the set of simulation states the associated DPI process can be in, and uses breadth first search to update the instances states and check DPI correctness. We use two optimizations to speed up breadth first search: we prune branches whose execution produces a simulation state that is guaranteed, based on the observed trace, to differ from the DPI's protocol state; and we identify messages whose effect will not change after being reordered with future messages and apply them immediately.

We have evaluated Ellsberg using three open-source DPIs: Etcd and ZooKeeper, which are consistent key-value stores that are used by projects such as Kubernetes for configuration and state management; and Redis Raft, an extension of the Redis key-value store that adds fault-tolerance. Etcd and Redis Raft implement the Raft [64] consensus protocol, and ZooKeeper implements the Zab [38] atomic broadcast protocol. Our evaluation (§6) shows that Ellsberg can detect bugs in these systems, that it can do so with minimal impact on response latency (we observed a worse case impact of 11% on the 99th percentile latency), no impact on throughput, and has modest processing and memory requirements.

Although the RPRC approach is general, and can be used with any distributed protocol that assumes fail-stop behavior and the asynchronous (or partially synchronous) model, it has an important limitation: it only suffices to detect protocol bugs that lead to changes in either the content or order of messages sent or received by a DPI process.We cannot detect several important classes of bugs, including ones that lead to livelock or deadlock, corrupt stored data, or degrade performance, and must rely on tools developed by prior work (§2.1) to detect these bugs. Nevertheless, as our evaluation (§6) shows, safety bugs of the kind we find do occur in practice.

2 Current Approaches and Related Work

Our goal is to catch protocol bugs in deployed DPIs that can lead to violations of safety properties. Before describing our approach, RPRC, we first review existing approaches, focusing on those that prevent or detect safety bugs. We categorize these work into five classes:

Trace Validation. A recently developed approach [15, 35] validates totally ordered traces generated during the testing against a TLA+ specification. This approach seeks to check protocol refinement at test time. However, it can only check execution traces generated by tests and, consequently can miss bugs. By contrast, Ellsberg is designed to minimize overheads, allowing it to be used in deployment and uncover bugs that were not found during testing. As a result, both approaches are complimentary: trace validation identifies bugs before deployment, while Ellsberg identifies bugs in

deployment.

Trace-validation also makes different assumptions than Ellsberg. It assumes that there is a close correspondence between implementation and specification, assuming that the the implementer used the TLA+ specification as a blueprint when writing code ([15, §1]) or that the specification was reverse engineered from the implementation. For DPI implementations, this requirement means that the DPI must log all changes to state variables, and log 'linearization points' that correspond to TLA+ state transitions. Thus, in contrast to our approach, DPIs must be implemented (or updated) for use with trace validation. Howard et al [35, §6.1] report that they needed to add 15 additional log statements to capture linearization points, which requires understanding both the protocol specification and the implementation. Similarly, for specifications, requiring close correspondence between implementation and specification means that specifications must be changed as applications are optimized or changed, and cannot necessarily be reused across applications. Indeed, the CCF effort had to develop a new detailed Raft specification [35, §4, §5] to use trace validation. By contrast, Ellsberg does not require close correspondence between the specification and the traces, and in our evaluation, we reuse the same Raft specification (modulo a few changes to account for the use of different reconfiguration protocols) for both Etcd and Redis Raft.

Furthermore, the trace-validation algorithm relies on two assumptions that make it challenging to use this approach in deployment: It assumes that (a) checks are run after the test has terminated; and (b) the test produces a totally ordered output. These assumptions require additional coordination – *e.g.,* Microsoft CCF [35, §6.1] uses a test driver to serialize node execution and a synchronized global clock to order process logs, and to decide when a test has completed – which affects a DPI's fault tolerance guarantees and performance.

Protocol Verification. Prior work has described several strategies to prove a distributed protocol's safety properties. Verdi [88], Diesel [81] and others [89] provide frameworks that make it easier for users to write proofs that can be mechanically checked by a proof assistant (*e.g.,* Coq [84]). Ivy [67, 68] provides an interactive tool to help users produce inductive invariants that can be used to prove the protocol's safety properties with an SMT solver. Recent works [55, 69, 91, 92] have extended Ivy's approach and automated the process of inferring inductive invariants.TLA+ [10, 62, 94] allows users to model check distributed protocol specification to find safety violations. Each of these strategies offers varying levels of automation and imposes different requirements on how protocols are specified. While these systems can prove that a protocol is correct (i.e. it maintains desired safety properties), they cannot prevent implementation bugs.

Use refinement proofs to prove implementations correct. This approach tries to derive the correctness of DPI implementations from verified protocols using refinement proofs. IronFleet [34] takes as input safety properties (referred to as a high-level specification), a protocol specification together with an implementation, and prove using Dafny [49] that (a) the protocol specification satisfies the specified safety properties, and (b) the implementation refines the protocol and thus also meets the safety properties. Recently, pretend synchrony [86] has shown how to reduce the proof burden for IronFleet by restricting the communication primitives used by the program. Concurrently, Verdi [88, 89] and Diesel [81] implement refinement proof using Coq's Ocaml extraction capabilities. All of these frameworks produce provably correct DPIs, however, to ensure that refinement proof generation and checking is feasible, they need to limit how DPI code is written, what libraries are used, and how the code is optimized [51].

Furthermore, these approaches make assumptions about the runtime environment, including assumptions about system call semantics and the network. Prior work [29] has shown that these assumptions are sometimes wrong, thus provably correct DPIs can still violate safety. Therefore, RPRC systems are useful even when using refinement proofs to produce provably correct implementations.

Use model checking and fuzzing to test existing implementations. Nearly all deployed DPIs use unit and integration tests to catch bugs. However, as with any large software system, developer provided tests cannot provide sufficient coverage. Prior work has developed tools for automatically generating tests. These tools can be classified into ones that use fuzzing or randomized testing [6, 43, 66], and ones that use model checking [3, 4, 6, 41, 42, 48, 60, 87, 90]. Many of these approaches [3, 43, 48, 66] are explicitly designed to be used with existing DPIs. Recent work from Majumdar and Niksic [58] has argued that randomized testing approaches are effective in finding DPI bugs. However, as is common with testing approaches, these tools cannot guarantee that all bugs are found. RPRC complements testing, allowing developers and administrators to discover bugs that occur when DPIs are deployed, but were not found during testing. However, note that as is the case with any runtime verification tool, bugs discovered using RPRC in production can also be found by a fuzzer that runs for long enough and explores enough of the implementation's code paths.

Runtime verification to find bugs in DPIs. Runtime verification approaches, *e.g.,* Dinv [33], D3S [50], Pip [77], Aragog [93], Oathkeeper [53] and Hydra [76], check program properties at runtime. These approaches are designed to check global properties, *e.g.,* agreement or state consistency, which are often expressed as predicates over the state of all (or some subset) of DPI processes. Consequently, these tools check program properties by first collecting a consistent snapshot of the relevant DPI state [13] and then evaluating a predicate on this state (see Franclanza et al's survey [31] for details). The use of DPI state snapshots allows these tools to check richer properties without concerns about decidability or scaling, *e.g.,* Aragog [93] checks performance and probabilistic staleness properties. However, the need to collect consistent state snapshots

adds coordination and communication overheads, and makes verification in the presence of DPI process failures challenging. Recent work has focused on reducing these overheads by minimizing the amount of process state collected [53, 76, 93], using fork-join parallelism to distribute predicate checking [33, 50, 76, 77, 93]. Other work has used multivalued logics [9, 14] to mitigate the effect of failures on runtime verification.

Our approach: Runtime protocol refinement checking (RPRC). RPRC seeks to combine mechanisms from both refinement proofs and runtime verification. Similar to refinement proofs, RPRC aims to ensure that each DPI process correctly implements a protocol, whose safety has verified using other approaches. However, unlike traditional refinement proofs which do so statically, RPRC's refinement checking occur at runtime to compare each DPI process' behavior to that of its protocol specification. Thus, RPRC avoids the feasibility concerns that lead tools like Verdi, Dafny and Verus to impose constraints on how DPIs are implemented. Additionally, runtime refinement checking does not require strong assumptions about system call semantics that are required by existing refinement tools. Unlike runtime verification, RPRC checks that DPI implementation refines a distributed protocol rather than checking that a safety property holds. Refinement can be checked locally without cross-node communication and coordination, and without imposing any limits on fault tolerance. By contrast, checking properties requires gathering that state at all processes, which necessitates coordination and is unavailable under failures.

2.1 Other Related Work

The tools discussed above focus on detecting safety bugs in DPIs. Below we discuss tools that look beyond safety to other issues including performance bugs, non-fail-stop failures and how to fix bugs once they are found. We discuss these below:

Distributed tracing tools. Distributed tracing [7, 20, 30, 57, 70, 82] is the most widely deployed approach to finding bugs in deployed DPIs. These tools generate logs that trace requests through the system, and several tools utilize these trace logs [8, 17, 40, 45, 46, 56, 61] to debug performance problems and failed user requests. These tools are valuable for debugging bugs, but cannot detect them.

Checking behavior under failures. Several tools and engineering practices, including chaos engineering [79], lineage driven fault injection [3–5, 59], crash consistency checking [2], and analysis tools for partial and correlated failures [37, 52, 96] have been developed to improve DPI behavior when recovering from crashes. We assume a fail-stop model, and do not currently handle these failures.

Performance and configuration bugs. Recent work has also developed tools and approaches to identifying performance bugs including gray failures [37], metastable failures [36], and bugs due to misconfiguration [97].

Record-replay based debuggers. Bugs, once found, need to be further diagnosed for debugging. Prior work has proposed several approaches for debugging DPIs [1, 8, 52, 80]. As we discussed previously, these include record-replay [8, 78, 80] based debuggers which, similar to RPRC, also need to consider multiple interleavings when reproducing bugs.

3 Ellsberg Design

We now describe Ellsberg, a RPRC system that we have implemented. We first state our assumptions about DPIs and the deployment environment (§3.1) and describe a ticket-lock service that we use as a running example in this section (§3.2). Then, we discuss the protocol specification a user must provide to Ellsberg (§3.3), and our RPRC algorithm (§3.4).

Overview. Ellsberg (Figure 1) performs runtime refinement checks on whether a deployed DPI correctly implements a protocol by comparing each DPI process's behavior against a user-provided protocol specification (§3.3). To do so, we require administrators to co-locate an Ellsberg instances with each DPI process, and to configure the deployment so that the Ellsberg instance has access to a trace of all messages sent and received by its co-located DPI process. At a high-level, the Ellsberg instance uses the following RPRC algorithm to check its co-located DPI process: it checks refinement by using the incoming messages in the trace to simulate potential protocol executions (using a protocol specification) and decides refinement check has failed if an invalid or out-of-order going message appears in the trace. It notifies administrators when the refinement check fails. Because Ellsberg must account for internal concurrency within a DPI, when simulating the protocol it maintains multiple simulation states, and uses breadth first search to consider multiple event interleavings (or schedules). Ellsberg uses outgoing messages in the trace to prune the set of simulation states that must be maintained, and the event interleavings considered.

3.1 Assumptions and Guarantees

Ellsberg assumes that the DPI and protocol work under the asynchronous (or partially synchronous) model and fail-stop failures. We do not consider byzantine failures.

When checking refinement, Ellsberg instances assume that the user-provided protocol specification is correct. We provide a testing tool (§4), that administrators can use before deploying Ellsberg to check the specification's correctness using the protocol's TLA+ specification.

We assume that the Ellsberg instance and its co-located DPI processes share fate, i.e. if an Ellsberg instance fails, its co-located DPI process also fails. We also require that the trace faithfully records all messages sent and received by its co-located DPI process, and is incrementally updated, *i.e.,* messages are added as the DPI process sends or receives them. Messages in the trace need to be annotated with the connection on which they were sent or received. We impose some ordering requirements on the trace. Specifically, we require that messages received over the same connection appear in order in the trace, and that outgoing messages follow program order, *i.e.,* they appear in the trace in the order in which they are sent, and that any incoming messages processed by the

```
1  struct State {
2      highest: int,
3      current: int,
4      held: bool
5  }
6
7  fn assign(state: State) {
8      let ticket = state.highest;
9      state.highest = state.highest + 1;
10     respond(Assigned(ticket))
11 }
12
13 fn acquire(state: State, ticket: int) {
14     if state.current ==  ticket{
15         state.held = true;
16         respond(Acquired(ticket))
17     }
18 }
19 fn release(state: State) {
20     let ticket = state.current;
21     state.current = state.current + 1;
22     state.held = false;
23     respond(Released(ticket))
24 }
```

Listing 1: *Example ticket-lock server.*

DPI before producing an outgoing message m appear before m in the trace. Our prototype uses an IPC channel, that connects each Ellsberg instance to its colocated process, to collect the trace (§5). Other approaches, *e.g.,* one where a network proxy mirrors messages to the Ellsberg instance, can also be used instead. Beyond the trace, Ellsberg has no access to DPI state: it cannot read the DPI process's memory, nor does it know the order in which received messages are processed.

Guarantees. When our assumptions are met, Ellsberg guarantees that it will generate an alert shortly after a DPI processes sends a message that is inconsistent (either because of its contents, or the order in which it was sent) with the specification, *i.e.,* shortly after a refinement violation is observed. Consequently, Ellsberg does not raise false-alarms because alerts are only generated when a violation is observed. However, similar to trace validation [15, 35], Ellsberg cannot prove that an implementation is correct because (a) it cannot detect refinement violations that do not result in the DPI sending out an inconsistent message; and (b) it is not guaranteed to have observed all possible DPI behaviors.

3.2 Running Example: Ticket Lock Service

We use a simple, non-fault tolerant, ticket lock-service as an illustrative example in this section. The service, shown in Listing 1, runs on a single server and maintains two integers: *highest* tracking the largest ticket the service has previously granted, *current* tracking what client should go next, and *held* which tracks whether a client holds the lock. The service implements three RPC calls:

1. `assign` (line 7), which assigns the client a ticket, increments *highest* so that clients get unique tickets, and responds with the assigned ticket number.
2. `acquire` (line 13), which takes as input a ticket number (*ticket*, previously acquired using `assign`) and checks if it is the client's turn to acquire the lock ($current == ticket$). If it is the client's turn, the service sets *held* to true and responds with an *Acquired* message.
3. `release` (line 19), which releases the lock by incrementing *current*, setting *held* to false, and responds with a *Released* message.

For ease of exposition, we assume that clients using this service are correct, and that a client does not call `release` unless it holds the lock. A correctly implemented version of this protocol guarantees mutual exclusion, ensuring that no-more than one client holds the lock at a time. We use this simple protocol to illustrate how to specify a protocol in Ellsberg, but we use real DPIs (Etcd, Zookeeper and Redis Raft) for the evaluation (see Appendix C for specifications).

3.3 Definitions and Specification

We model a distributed protocol as a collection of communicating processes, each of which is specified by an I/O automaton [27, 54]. More concretely, Ellsberg requires users to specify the distributed protocol as a state machine (executed by each process), whose execution is driven by a sequence of *events* that are *applied* to it. There are two types of events: the delivery of a message, and timeouts. When an event is applied, a process can update its state and send 0 or more messages.

The terms **pending events**, **reachable states** and **inducing states** are defined as follows:

Pending Events: A pending event is one that can be applied to a process, and consists of previously-received messages that the have not yet been processed, and timeouts. Timeouts in a DPI process are not recorded in the trace, and our model assumes that a timeout is always pending. We use the term *schedule* to refer to a sequence of events.

Reachable state: Given a process p in state s, we say that state s' is reachable if and only if there exists a schedule of events (some of which might not yet be received) that when applied to s result in state s'.

Inducing states: Given a message m, we say that state s is m-inducing (we drop the prefix $m-$ when the message is clear from the context) if and only if there exists state s' and event e, such that applying e to a process in state s' results in the process *sending* the message m and then transitioning to state s. In the ticket-lock example (Listing 1), the m-inducing state for an `Acquired` message for ticket 2 is any state s, where $s.current = 2$. Observe that in general, a message m does not have a unique inducing state, and some messages might have an unbounded number of inducing states: in the previous example, any state s for which $s.current = 2$ is m-inducing, regardless of $s.highest$'s value.

3.3.1 Specification

Ellsberg requires users to provide two inputs: a specification for the distributed protocol the DPI purportedly implements and mapping functions that allows Ellsberg to map network messages sent or received by the DPI into messages that appear in the specification. In the remainder of this section, we only refer to messages that appear in the protocol specification. In Listing 2, we use the ticket-lock service's (§3.2) specification to illustrate the parts of a Ellsberg specification.

A Ellsberg protocol specification consists of:

```
1  struct ProtState {
2      highest: int,
3      current: int,
4      held: bool,
5  }
6
7  fn equal(s: ProtState, s': ProtState) -> bool {
8      return s.highest == s'.highest
9        && s.current == s'.current
10       && s.held == s'.held
11 }
12
13 fn infer_inducing(m: Message) -> ProtState {
14     return match m {
15         Acquired(tkt) =>
16             ProtState{highest: None,
17               held: true, current: tkt},
18         // ...
19     }
20 }
21
22 fn apply(s: ProtState, e: Event) -> ProtState {
23   match e {
24     case Message(Acquire(tkt)) => {
25         if s.current ==  tkt{
26             return ProtState{highers
                    : s.highest, current:tkt, held:true}
27         } else {
28             return s
29         }
30     }, // ...
31   }
32 }
33
34 fn reachable(s: ProtState,
35              pending: Set[Set[Message]], e: Event,
36              p: ProtState) -> bool {
37    if state(s).current > p.current {
38     return false
39     } // ...
40   }
41 }
42
43 fn apply_asap?(s: ProtState, m: Message) -> bool {
44     return m.type == Acquire && m.tkt < s.current
45 }
46
47 fn lookahead_type(m: Message) -> Message_type {
48     if Type(m) == Acquired {
49         return Assigned
50     }
51     // ...
52 }
```

Listing 2: *User-provided protocol specification for the ticket-lock server*

(i) The definition for a *protocol state structure*, `ProtState` (line 1), that Ellsberg uses to track the current simulation's protocol state, and a function to create the simulation's initial protocol state. For notational convenience, all elements of the `ProtState` are nullable and we use this to encode sets of states.

(ii) A transition function, `apply` (line 22), that takes as input a protocol state s, a collection of sets of pending events and an event e, and returns the protocol state produced by applying e to s. We guarantee that all events passed to the transition function are pending for s. For notational convenience, we use $apply(s,\Sigma)$ to represent the output of applying the sequence of events in the schedule Σ to s.

(iii) A function, `equal`, to check equality between two protocol states s and s'. Two states are considered equal if the distributed protocol exhibits the same behavior for both. In other words, we assume that $equal(s,s') = true$ implies that for any schedule Σ, applying the schedule to both states results in final states that are equal and produces the same sequence of outgoing messages. In the ticket-lock example (line 7) the function compares *current*, *highest* and *held*.

(iv) An inference function, `infer_inducing`, that given an *outgoing* message m returns a `ProtState` of all m-inducing states. This function populates only those values that can be inferred from m, leaving the others unknown. Line 13 shows this function for our running example: we need to consider each type of outgoing message sent by the service when writing it.

(v) A `reachable` function that takes protocols state s, the set of pending messages, and a target partial protocol state S, and returns true if there exists s' such that $s' \in S$ and s' is reachable from $apply(s,e)$. We assume that this function over-approximates reachability, *i.e.,* it might return true even if there exists no reachable s' such that $s' \in S$, but never incorrectly returns false. For brevity, we only show a portion of the reachable function for the running example, which returns false if the state's *current* value is larger the target state (line 34).

(vi) A function, `apply_asap?`, that we use to reduce the simulation's memory requirements and runtime by reducing the number of schedules that need to be considered (§3.4.2). This function takes a protocol state s and an event e, and returns true if and only if we can safely apply the event immediately, *i.e.,* we do not need to consider schedules that reorder e with respect to other events (including ones that occur in the future). We define `apply_asap?`(s,e) to be correct if it meets the following two requirements:

  (a) **Event e can be reordered.** For any schedule Σ where $e \in \Sigma$ and message m, if $apply(s,\Sigma)$ is m-inducing, then so is $apply(apply(s,e),\Sigma - e)$.

  (b) **Event e does not block outgoing messages.** For any schedule Σ where $e \notin \Sigma$ and message m, if $apply(s,\Sigma)$ is m-inducing, then $apply(apply(s,e),\Sigma)$ is also m-inducing.

  These requirements ensure that applying e to protocol state s does not change the set of reachable m-inducing states, and Ellsberg can avoid exploring schedules that reorder e. An acquire message for a ticket smaller than the server's current ticket meets these requirements in our example (line 43).

(vii) An optional function `lookahead_type` that we use to reduce the number of m-inducing states found by the `infer_inducing` function, thus reducing simulation time. This function takes as input a message m and either returns `None` (indicating there is no such type) or a message type. As we explain later (§3.4.2), this function is used in an optimization to improve Ellsberg's efficiency, but its semantics do not affect correctness. Line 47 shows this function for the lock service: when

processing an outgoing acquire message Ellsberg looks ahead to find the next assigned message. This is because we cannot infer the value of the `highest` field from an acquire message, but can from an assign message, and considering both reduces the number of m-inducing states,and thus schedules that Ellsberg must explore.

In §4, we describe how a user can derive Ellsberg specifications from existing TLA+ specifications used to prove protocols correct, and test the derived specification. However, note that deriving a Ellsberg specification from a TLA+ specification is simpler than deriving an implementation: unlike implementations, Ellsberg specifications do not need to interact with communication libraries, consider failure handling, or implement application logic. Furthermore, Ellsberg specifications are written assuming sequential execution, and operators do not need to protect against data races or other concerns common to concurrent code. Consequently, it is easier to derive a correct Ellsberg specification than an implementation from a TLA+ specification.

3.4 The Ellsberg Algorithm

We now detail the Ellsberg algorithm. Concurrency and incrementally checking DPI correctness (so Ellsberg can report bugs soon after they are observed) were the two main challenges we addressed when developing this algorithm.

Concurrency is challenging because the protocol's behavior depends on the order in which events are applied, but Ellsberg does not know the order in which the messages and timeouts were processed by its associated DPI process. Similar to other work, we resolve this by considering multiple schedules consisting of different interleavings of incoming messages in the trace and timeouts.

Incremental checking makes constructing multiple schedules challenging because the DPI process might process a received message m, which appears in the trace, after message m' that has not yet been received, and thus does not appear in the trace. We resolve this challenge using the assumed the trace ordering property that guarantees that any received messages processed by the DPI process before it produces an outgoing message m must appear before m in the trace, and that outgoing messages appear in program order. Thus, when checking an outgoing message m, Ellsberg only considers schedules that contain incoming messages appearing before m in the trace. For notational convenience, we refer to the trace before m as the m-prefix. Furthermore, if the trace contains a sequence of outgoing messages $m_0, m_1, \ldots, m_n$, the simulation needs to only consider schedules which use incoming messages in the m_0-prefix before producing outgoing message m_0, messages in the m_1-prefix before producing outgoing message m_1, etc.

Ellsberg uses this observation to implement an incremental checking algorithm (Listing 3): Each Ellsberg instance maintains a set of simulation states S, which we define as a protocol state with a set of collections of pending message (line 2). For notational convenience, we use s_{sim}

```
1
2  struct SimState {
3      state: State,
4      pending: Set[Set[Messages]]
5  }
6
7  // The instance's current simulation state.
8  S: Set[SimState]
9
10 fn prune_asap(s: SimState) -> SimState {
11     for e in pending(s) {
12         if apply_asap?(state(s), e) {
13             return prune_asap(apply(s, e))
14         }
15     }
16     return s
17 }
18
19 // Process incoming message m in the trace.
20 fn process_incoming(m: message) {
21     for s in S {
22         // Check if m should be applied immediately.
23         if apply_asap?(state(s), m) {
24             S.replace(s, prune_asap(apply(s, m)))
25         } else {
26             s.add_pending_message(m)
27         }
28     }
29 }
30
31 // Process outgoing message m in the trace.
32 fn process_outgoing(m: message) {
33     // Get m-inducing states.
34     let target = infer_inducing(m)
35     // Check if we need to lookahead.
36     let m_lookahead = if lookahead_type(m) {
37         infer_inducing(
38             find_next_of_type(lookahead_type(m)))
39     } else {
40         None
41     }
42     // Updated simulation state.
43     let S' = new Set[SimState]
44     for s in S {
45         // Find reachable m-inducing states from s.
46         S_s = find_reachable(s,
47                   target, m_lookahead)
48         S' = S'.union(S_s)
49     }
50     if S'.is_empty() {
51         // No reachable m-inducing state indicates
52         // divergence b/w spec and implementation.
53         raise Error
54     } else {
55         S = S'
56     }
57 }
```

Listing 3: *Algorithm for checking a single outgoing message.*

to refer to simulation states, and $state(s_{sim})$ to reference the underlying protocol state. Initially S contains a single element: a simulation state with the initial protocol state and an empty pending message set. The algorithm iterates through the trace, and adds incoming messages (with one exception discussed in §3.4.2) to the pending message set of all simulation states $s_{sim} \in S$ (line 20). It processes an outgoing message m by finding all m-inducing simulation states that can be reached from any $s_{sim} \in S$ using s_{sim}'s pending messages and zero-or-more timeouts (line 46, §3.4.1). The algorithm notifies an error (line 53) if no m-inducing simulation states are reachable, otherwise it updates S to be the set of reachable m-inducing simulation states (line 55).

For example, if the ticket-lock service sends a message m indicating that the client with ticket 2 has acquired

```
1
2  fn find_reachable(s: SimState,
3         target: ProtState,
4         ahead: Option[ProtState]) -> Set[SimState] {
5      // The queue of SimState's to explore.
6      let to_explore = new Queue[SimState]
7      // Previously explored SimState's
8      let explored = new Set[SimState]
9      // The set of reachable m-inducing SimStates.
10     let reachable_inducing = new Set[SimState]
11     to_explore.insert_tail(s)
12     while !to_explore.is_empty() {
13         s' = to_explore.remove_head()
14         explored.add(s')
15         // Is s' m-inducing?
16         if state(s') in t {
17             reachable_inducing.add(s')
18             continue
19         }
20         for e in pending(s') {
21             // Is target reachable from apply(s', e)
22             if reachable(state(s'), pending(s'), e,
                    target) &&
23                (ahead.is_none() ||
24                 reachable(state(s'), pending(s'), e,
                        ahead)) {
25                 s'' = prune_asap(apply(s', e))
26             }
27             // Equivalent to enqueued state?
28             if to_explore.contains_state(s'') {
29                 // Add pending messages
                        in s'' to existing state.
30                 merge_pending(
31                    to_explore.get_state(s''), s'')
32             } else if !explored.contains_state(s'')
                    {
33                 // Add s'' to the exploration queue.
34                 to_explore.insert_tail(s'')
35             } else {
36                 // Add s'' to the exploration queue
37                 // if pending events differ from
38                 // previously explored.
39                 if !pending(explored.get_state(s''))
40                     .contains(pending(s'')) {
41                     to_explore.insert_tail(s'')
42                 }
43             }
44         }
45     }
46     return reachable_inducing
47 }
```

Listing 4: *Ellsberg's breadth-first search algorithm to find reachable m-inducing simulation states (passed in as argument* `target`*) starting from simulation state* `s`.

the lock ($m = Acquired(2)$) message, and S contains a single simulation s_{sim} for which $state(s_{sim}).held = true$ and $state(s_{sim}).current = 1$, then Ellsberg would throw an error if $pending(s_{sim})$ did not contain a release message.

Observe that because the trace is in program order, and the algorithm processes the trace iteratively, Ellsberg only considers incoming messages (and zero-or-more timeouts) in the trace's m-prefix when processing outgoing message m Furthermore, since outgoing messages are processed iteratively in order, the algorithm checks both the order and contents of outgoing messages sent by the DPI process.

3.4.1 Finding reachable states

The algorithm described above needs to find m-inducing simulation states reachable from the current simulation state s_{sim}. To do so, it first uses the `infer_inducing` function to compute *target* (Listing 3 line 34), which is the partial protocol state derived from outgoing message m. The algorithm then calls `find_reachable` with s_{sim} and *target* as arguments to find the subset of simulation states that are both consistent with the partial state *target* and reachable from the current simulation state s_{sim}. To make our state exploration efficient, we must design `find_reachable` to work efficiently by returning the smallest set of simulation state. In particular we require that if $s,s' \in find_reachable(s_{sim},t)$ then $equal(s,s') = false$. We had to solve two challenges when designing the `find_reachable` algorithm to meet this requirement:

Challenge 1: Many schedules can lead to the same m-inducing state. The schedule used to reach a simulation state s_{sim} dictates its pending message set. But there exist cases where `find_reachable` can reach two (or more) simulation states s_{sim} and s'_{sim} with the same underlying protocol state (*i.e.,* $equal(state(s_{sim}),state(s'_{sim})) = true$) but different pending message sets $pending(s_{sim}) \neq pending(s'_{sim})$. Which of these simulation states is returned can affect correctness. In this case, our handling depends on why multiple schedules reach the same semantically equivalent state:

a. There might exist cases where a protocol state s can be reached through two schedules Σ and Σ', and Σ is a subsequence of Σ' . This often happens because applying some message m has no effect on the protocol state s. For example, consider the ticket-lock server (Listing 1) with simulations state s_{sim} such that $state(s_{sim}).current = 1$ and $pending(s_{sim})$ contains an acquire message m_a for ticket 2 ($m_a = acquire(2)$). Applying m_a to s_{sim} does not change its protocol state because the client cannot acquire the lock, and $equal(state(apply(s_{sim},m_a)),state(s_{sim})) = true$. But, the pending set of $apply(s_{sim},m_a)$ does not contain m_a and is a subset of s_{sim}'s pending set.
   However, including the simulation state with a smaller pending state in `find_reachable`'s return can lead to false alarms: consider the case where s_{sim}'s pending set is $\{m_a,m_r\}$ where $m_r = release()$, and we are searching for an $Acquired(2)$-inducing state. It is clear that an inducing state is reachable from s_{sim} using the schedule $\Sigma = [m_r,m_a]$, but is not reachable from $apply(s_{sim},m_a)$, whose pending set does not include m_a.
   Therefore, in this case `find_reachable` should add the simulation state with the *largest pending set*[1] to the set of returned state. We ensure that this is the case by having `find_reachable` use *breadth-first search* (Lines 12–45) to find reachable target states. Breadth first search ensures that shorter schedules, and thus simulation states with larger pending message sets, are explored before simulation states with smaller pending message sets.

b. Semantically equivalent states might also be reached through two schedules Σ and Σ' neither of which is a subsequence of the other. We resolve this by having simulation states in Ellsberg track a set of pending message sets, allowing us to use a single simulation state to track

[1] Σ is a subsequence of Σ', and thus the pending set of the simulation state reached from Σ must be a superset of the other.

the pending message sets in this case (Lines 31 and 40).

Challenge 2: Terminating the search. Our algorithm searches through schedules that consist of pending incoming messages and zero-or-more timeout. These schedules can have unbounded length, because timeouts are always enabled. This poses a challenge, since `find_reachable` might explore unnecessarily long schedules and might not terminate. We address this problem in two ways: (a) We avoid redundant explorations by tracking simulation states that have already been explored by `find_reachable` (line 32). During breadth-first search `find_reachable` checks if a simulation state s_{sim}'s protocol state ($state(s_{sim})$) is the same as that of a previously explored simulation state (line 40, the `contains_state` function finds simulation states with equal protocol states): if not, it enqueues that simulation state for exploration, and otherwise terminates exploration for that simulation state. (b) Before scheduling exploration for simulation state $apply(s_{sim},e)$, we use the `reachable` function in the specification to first check if a target m-inducing state can be reached from s_{sim} (Line 22). We prune exploration from $apply(s_{sim},e)$ if the `reachable` function returns false.

3.4.2 Optimizing State Exploration

We use two performance optimizations:

Avoiding redundant exploration. The `find_reachable` algorithm explores different schedules which reorder incoming messages and zero-or-more timeout events in its search process. However, if the current simulation state s and a pending message m meet the `apply_asap?` requirements (§3.3) then when m is applied does not affect the search.

Therefore, we use the `apply_asap?` function to improve exploration efficiency in two places: (a) When processing an incoming message m from the trace (Listing 3), Ellsberg checks whether it can be applied immediately to a simulation state $s_{sim} \in S$, and if so it uses the recursive function `prune_asap` (Line 10) to replace s_{sim} with the result of applying $apply(s_{sim},m)$ and all other messages that can be applied immediately (Line 24); and (b) in the `find_reachable` (Listing 4) algorithm when adding a new simulation state s_{sim} to the set of states that need to be explored (Line 25) we again use `prune_asap`. Note, that while this optimization is similar in principle to partial-order reduction [28] (POR), our technique is designed for the incremental setting.

In Appendix A we describe how we can mechanically check that `apply_asap?` is correct, and thus the use of this approach does not come at the cost of correctness.

Looking ahead to the next outgoing message. The number of m-inducing states found by `find_reachable` is determined by the outgoing message m being considered: an outgoing message m that reveals little about the DPI's protocol state, *e.g.,* client response messages, may allow for a large set of reachable m-inducing states. Consequently, when processing m, `find_reachable` might return a large set of simulation states that the Ellsberg instance would use when it next processes an outgoing message. However, the time

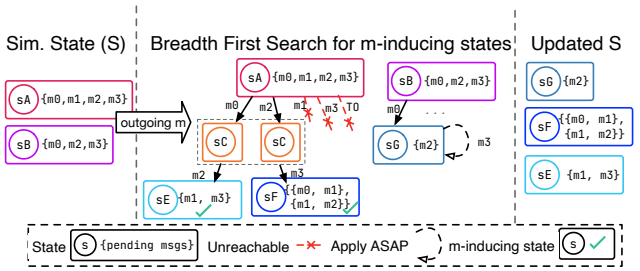


Figure 2: *An overview of how a Ellsberg processes outgoing message m using BFS to find all m-inducing states reachable from the current simulation state.*

taken to process an outgoing message depends on the size of the simulation set S because it determines the number of `find_reachable` calls made (Listing 3 line 44). Smaller simulation sets are thus preferable. Ellsberg implements an optimization that can reduce the set of states returned by `find_reachable` in some cases. Specifically, the optimization "looks ahead" to the next message m_T of type T that occurs after m in the trace and passes to `find_reachable` an optional argument (`ahead`) containing the m_T-inducing state, allowing `find_reachable` to prune any simulation states $s_{sim} \in S$ from which an m_T-inducing state is not reachable (Listing 4 line 24).

We use the `lookahead_type` function in the user-provided specification (§3.3, Listing 2) to decide when to use this optimization. As a reminder, the `lookahead_type` takes as input the outgoing message m that Ellsberg is processed, and either returns `None` (indicating lookahead should not be used) or a message type T. If the function returns a type T, Ellsberg scans the trace, starting at m to find the next outgoing message m_T of type T (Listing 3 line 38) and passes the m_T-inducing state to `find_reachable`. The `find_reachable` function then only returns simulation states that are both m-inducing and m_T-inducing, a smaller set than the set of all m-inducing sets. Note that our use of `lookahead_type` ensures that its output has no impact on correctness: if the function returns type T, then any simulation state s_{sim} pruned by `find_reachable` would have been pruned by a future call to `find_reachable` when considering the next outgoing message of type T.

3.5 Algorithm Summary and Generality

In summary, each Ellsberg instance works as follows: it maintains a set of simulation states S and iterates through the trace. An incoming message m in the trace is processed (Listing 3 Line 20) by iterating through S, and checking for each simulation state $s_{sim} \in S$ if the message should be immediately applied: if so, s_{sim} is replaced by $apply(s_{sim},m)$, if not m is added to s's pending set. An outgoing message m (Figure 2) in the trace triggers a check to determine if the DPI is buggy (Listing 3 Line 32), which involves iterating through the simulation states $s_{sim} \in S$ and using the `find_reachable` (Listing 4) algorithm to determine if an m-inducing state is reachable from s_{sim}. Ellsberg reports a bug if no m-inducing states are reachable from any simulation state $s_{sim} \in S$, otherwise S is replaced by the union of all reachable m-inducing simulation states.

When does Ellsberg detect a bug? Ellsberg? Ellsberg only has visibility into what messages have been sent or

System	Ellsberg LOC	Baseline	Baseline LOC
ETCD	952	CCF Raft [35]	1503
RedisRaft	894	CCF Raft	1503
Zookeeper	989	ZooKeeper TLA+ [38]	1615

Table 1: *Lines of code in Ellsberg specification when compared to those used by existing tools. For baselines, we use the Raft specification from Microsoft CCF [35] Etcd and RedisRaft, and a recent implementation derived TLA+ specification [38] for ZooKeeper.*

received by DPI thus far, and hence, at any point in time, it can only detect bugs that are apparent from the trace prefix provided to it thus far. Furthermore, even for this prefix, our approach checks if there exists any schedule (*i.e.,* a total order) of incoming messages that when applied to the specified distributed protocol produces the sequence of output messages observed in the trace prefix. So, as we noted earlier (§3.1) Ellsberg can only detect a bug once it observes an output message that is inconsistent with the specification.

Generality & Applicability. Our algorithm assumes that the protocols being checked can be specified as a collection of processes, each of which is an I/O automaton. Concretely, this assumption means that events (received messages or timeouts) must be applied atomically, and a messages behavior cannot be influenced by a message being applied concurrently. This condition holds for many but not all distributed protocols, *e.g.,* it does not hold for a distributed concurrency control protocol where multiple transactions can be executed simultaneously and have their results validated after the fact.

An additional protocol assumption is required to make incremental checking feasible: protocols must include one or more messages that (perhaps together) allow Ellsberg to infer the entire state of a DPI process. Absent such messages, Ellsberg must maintain an ever increasing set of partial states and consider a growing number of schedules when checking outgoing messages, making the use of Ellsberg infeasible in practice. These messages exist for most protocols, including the RSM protocols we use in our evaluation. However, we have encountered one case where this assumption does not hold: databases that use multi-version concurrency control, and whose messages do not include information about what version a transaction read from or updated. In this case applying Ellsberg requires us to consider all possible orderings for concurrent transactions, incurring similar complexity as prior transaction verification approaches [83, 98].

4 Writing Ellsberg Specifications

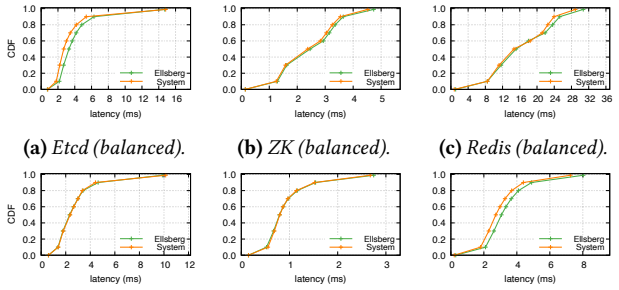
To use Ellsberg, users or protocol authors need to translate existing specifications, *e.g.,* ones used to verify safety, into Ellsberg specifications. Our evaluation uses specifications derived from TLA+ protocol specifications: for Raft we used the TLA+ specification [63] provided by the Raft authors, while for ZooKeeper we used an implementation derived TLA+ specification [38].

Table 1 shows the length of our specifications ('Ellsberg LOC' column). We use the same Raft specification for Etcd and Redis Raft, the difference in length is because they support different reconfiguration protocols. To put our specifications length into context, we also report the length of specifications used by other tools: for the two Raft DPIs we compare to the Raft specification used by Microsoft CCF (written specifically for use with that tool), and for ZooKeeper we compare to the implementation derived TLA+ specification we used. Note, that the Microsft CCF Raft implementation supports a different (and simpler) reconfiguration protocol than Etcd, but the protocols are comparable. Our specification is comparable in length to these baseline specifications, and indeed a few lines shorter. Appendix C provides additional details about the Ellsberg specifications, including a detailed breakdown of lengths for each part (Appendix C.1).

We adopted the following approach to derive the `ProtState` structure, and the `equal`, `infer_inducing`, and `apply` functions from TLA+:

- **`ProtState`**. The `ProtState` structure contains all per-server variables that appear in the TLA+ specification. We derived it by identifying all variables in the TLA+ specification, filtering out those that are global variables (*e.g.,* `messages` in the Raft specification [63]) or ghost variables (*e.g.,* `voteResponded` in the Raft specification), and adding the remaining variables to the structure.
- **`equal`**. The equal function merely checks that all state variables have the same value, and the language (or runtime) provided equality function suffices.
- **`apply`**. The `apply` function matches over all possible state machine transitions, and updates a provided `ProtState` structure. We derived it by walking through the TLA+ specification to find transitions (*e.g.,* `ClientRequest(i, v)` in Raft) and copying over any state updates that appear in the transition.
- **`inference`**. The `inference` function takes an outgoing message and tries to infer the process' DPI state. To derive this function, we first identify transitions that send messages (*e.g.,* by finding transitions that call `Send` in the Raft specification).Given these transitions, we use three techniques to infer state variables values: (i) If the outgoing message directly uses a state variable, *e.g.,* term, we can infer the state value directly from the message; (ii) If the outgoing message contains a value computed from a state variable, *e.g.,* last log index, we use the inverse computation to infer the state value; and (iii) If the outgoing message is sent only when a particular state variable has a given value, then we can infer the value of the state variable when we observe the message: *e.g.,* we can infer that Raft a node sending a `RequestVoteRequest` message is a candidate.

We had to write de novo `apply_asap?`, `reachable`, and `lookahead_type` functions. Of these, the `apply_asap?` function can be *mechanically proven correct* (described in Appendix A), and the `lookahead_type` function has no impact on Ellsberg's correctness (§3.4.2).



(a) *Etcd (balanced).* **(b)** *ZK (balanced).* **(c)** *Redis (balanced).*

(d) *Etcd (read-heavy).* **(e)** *ZK (read-heavy).* **(f)** *Redis (read-heavy).*

Figure 3: *Response latency CDFs from a 5-node cluster when just the DPI is deployed (System) and when a colocated* Ellsberg *instance is deployed (*Ellsberg*). 'ZK' refers to Zookeeper.*

Testing Ellsberg specifications. We also provide a testing tool, inspired by Mocket [87], to check that the derived Ellsberg specification matches the original TLA+ specification, and does not have translation bugs.

Our testing tool takes as input a TLA+ specification, and generates valid message traces (that is message traces that do not lead to safety violations) using this specification. Message traces are generated from the TLA+ specification as follows: first, the tool uses TLC's [44] bounded model checking functionality to generate a transition graph for a bounded number of steps; next, it performs depth first traversal of the transition graph and keeps track of the sequence of states along each traversed path; finally, it translates the sequence of states into message traces. The last step builds on the observation that TLA+ specifications model messages sends and receives as modifications to a message vector, allowing our tool to translate state sequences into sequences of message sends (or timeouts).

The tool then uses the generated traces to exercise Ellsberg specification, and reports a bug if the Ellsberg specification would have (falsely) notified an error for the trace.

5 Implementation

We implemented Ellsberg in Go, and our current prototype requires protocol specifications to be written in Go and compiled with Ellsberg. We evaluated our prototype using specifications for Raft [64] and ZAB [38], and used Etcd, ZooKeeper and RedisRaft as DPIs.

The Raft specification we use consists of ∼500 lines of code, and the ZooKeeper specification consists of ∼2000 lines of code. We describe how we developed these specifications from existing machine checked specifications in Appendix C. We used the Ellsberg testing tool (§4) to test both specifications. For Raft, our test ran 15 steps of bounded model checking (this was sufficient to cover all protocol states), and validated the specification using 8,750,468 message traces; while for Zookeeper, our test ran 20 steps of bounded model checking (again, sufficient to cover all protocol states) and validated the specification using 1,904,456 message traces.

Overall, the Ellsberg implementation (including both specifications) consists of ∼4500 lines of code. Of this total, the testing tool consists of about 500 lines of Python code, and the rest is Go code for the Ellsberg instance and both specifications. In addition to the protocol specification, users must also provide Ellsberg with DPI specific code to deserialize and map implementation messages to specification messages. The mapping function for Etcd is 356 lines of Go code, for ZooKeeper it is 370 lines of Go code, and for RedisRaft it is 380 lines of Go code.

Our prototype uses an IPC channel to get the trace, and we modified the network APIs for both DPIs to update the trace. Doing so required adding logic (to forward messages) to 11 functions in Etcd, 31 functions in ZooKeeper, and 10 functions in Redis Raft. We were careful to ensure that no additional timing or ordering information was sent over this IPC channel. We used IPC for simplicity, but our implementation can adopt alternate approaches, *e.g.,* having a service proxy [19] copy messages to Ellsberg.

6 Evaluation

We evaluated Ellsberg using three widely-used open source DPIs: Etcd, Zookeeper, and Redis Raft. Our evaluation focused on three questions: (a) Can Ellsberg detect bugs at runtime? (§6.2) (b) Can Ellsberg be deployed in production? (§6.3) (c) How does Ellsberg perform: how long does it take Ellsberg to process a trace, and the impact of our design (§3.4) on performance? (§6.4)

6.1 Setup and Workload

Our evaluation deployed Ellsberg on 3- and 5-node DPI clusters. Each cluster node was a C6525-25G instance in CloudLab [18], with a 16-core 3GHz AMD EPYC 7302P processor, 128 GB of RAM, two 25Gbps NICs, and ran Ubuntu 20.04. We deployed a DPI process and Ellsberg instance on each node, and used `taskset` to limit the Ellsberg instance to two cores. Ellsberg instances were configured to process trace messages every second.

We exercised the DPIs using workloads derived from YCSB [16]. We use two benchmarks: a *balanced* workload with 50% reads and 50% writes, and a *read-heavy* workload with 95% reads and 5% writes. For both workloads, keys are drawn uniformly at random from among 1 million possible keys. We used 120 concurrent DPI clients as load generators, we found that this maximized observed throughput for all DPIs. To minimize performance variance, we disabled checkpointing for all DPIs.

In the interest of space, we use *ZK* to refer to Zookeeper and *Redis* to refer to Redis Raft in figure captions in this section.

6.2 Can Ellsberg detect bugs?

We evaluated Ellsberg's ability to detect protocol implementation bugs by reproducing previously reported bugs in Etcd, Redis Raft and Zookeeper, and checking whether Ellsberg notified when the bugs occurred. Due to space constraints we provide the bugs and their root causes in Appendix D, but summarize them in Table 2. The bugs we tested on including bugs in the linearizable read protocol implemented by Etcd [25] and Redis Raft [72]; a stale-read

System	Bug	Type
Etcd	741 [25]	Linearizable read
	7331 [24]	Stale read after election
	12133 [22]	Reconfiguration
	7280 [23]	Reconfiguration
Zookeeper	1154 [101]	Data inconsistency
Redis Raft	17 [71]	Reconfiguration
	19 [72]	Linearizable read
	52 [74]	Lost updates
	256 [73]	Reconfiguration
	Unreported	Reconfiguration

Table 2: *Bugs used to evaluate Ellsberg's bug detection capability.*

bug in Etcd [24] because a new leader's commit index might lag behind the previous leader's commit index; a bug in Zookeeper's [101] leader election protocol (leading to data corruption); a bug in Redis Raft's caches that leads to lost updates [73]; and reconfiguration bugs in Etcd [22] and Redis Raft [71]. During testing, we also identified an unreported bug in Redis Raft's reconfiguration protocol implementation.

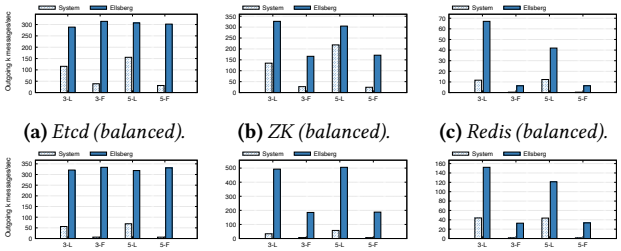
6.3 Can Ellsberg be used in production?

We demonstrate that deploying Ellsberg in production is feasible by showing that it has low resource requirements, and does not impact DPI performance. In terms of resources: our evaluation uses `taskset` to limits each Ellsberg instance to 2 cores. Furthermore, we found (explained in detail in Appendix E) that across all our evaluations, each Ellsberg instance tracks one simulation state on average (*i.e.*, $|S| = 1$) and has between 0 and 5 pending messages, and thus has minimal memory requirements. Additional, Ellsberg does not use the network by design. We thus conclude that Ellsberg has minimal resource overheads.

In terms of DPI performance impact, Figure 3 shows the performance of the DPIs with ('Ellsberg') and without ('System') a collocated Ellsberg process. We show results for a 5-node Etcd, Zookeeper, and Redis Raft cluster when running a read-heavy and balanced workload. We omit results for 3-node clusters, but they were similar. We observe that collocating Ellsberg increases tail-latencies slightly: the largest increase we observed was for Redis when running the read-heavy workload(Figure 3f) where 99th percentile latency increases by 10.7% (from 7.25ms to 8.03ms). These results show that using Ellsberg has minimal impact on response latency, and thus production deployment is feasible.

6.4 End-to-End Performance

Ellsberg's detection latency is determined by how often it checks the trace, in our evaluation we use a configuration (§6.1) where it checks the trace every second (allowing us to batch checking and reduce overheads). Therefore, we evaluate Ellsberg's performance by measuring its throughput, and report results in Figure 4. We also report the DPI's throughput when it is run without a colocated Ellsberg process: the DPI throughput both serves as a comparison point to evaluate Ellsberg's algorithmic efficiency; and



(a) *Etcd (balanced).* **(b)** *ZK (balanced).* **(c)** *Redis (balanced).*

(d) *Etcd (read-heavy).* **(e)** *ZK (read-heavy).* **(f)** *Redis (read-heavy).*

Figure 4: *Ellsberg's throughput compared to DPI throughput (System) for different workloads and DPIs. In these graphs, 3-L and 3-F respectively refer to the leader and follower in a 3-node cluster, and 5-L and 5-F refer to the leader and follower in a 5-node cluster.*

addresses whether Ellsberg can keep up with the colocated DPI process (if Ellsberg had lower throughput it might not keep up with a loaded DPI). We report results from both 3- and 5-node clusters and both workloads, and measure throughput in outgoing messages per second. The results were collected by measuring throughput in a 10-second interval over 10 experiments. We omit error bars because we observed nearly no variance across experiments.

Our results show that across DPIs and workloads, Ellsberg achieves higher throughput than the DPI: for Etcd Ellsberg has 2.0—51.7× higher throughput, for Zookeeper Ellsberg has 1.4—29.7× higher throughput, and for Redis Raft Ellsberg has 3.1—25.5× higher throughput. In practice, we found that processing a second of trace events took Ellsberg between 30 − 700ms when the co-located DPI node was a leader, 20 − 180ms otherwise. In sum, this means that in our evaluation Ellsberg notifies administrators within 1.7 second of a bug occurring, though this could likely be reduced with a different configuration. Appendix E further evaluates the impact of Ellsberg's optimizations.

7 Conclusion

This paper proposed *runtime protocol refinement checking*, an approach that combines ideas from refinement proofs and runtime verification to notify administrator of protocol bugs in deployed DPIs. Unlike static refinement proofs RPRC can be used with existing unmodified DPIs, and unlike runtime verification RPRC does not require coordination or communication. We also described an algorithm for RPRC, and our implementation of this algorithm Ellsberg, and applied it to three commonly used DPIs: Etcd, ZooKeeper and Raft. We believe that RPRC and Ellsberg provide a practical approach for improving the reliability of deployed DPIs.

Acknowledgment

We thank Wen Zhang, Joe Tassarotti, Mike Walfish and Scott Shenker for discussions that helped shape our approach presentation. We also thank our shepherd Si Liu and the anonymous NSDI reviewers for comments that further improved our presentation. This work was supported in part by NSF award CNS-2145471 and a gift from the Stellar Foundation.

References

[1] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[2] Ram Alagappan. Crash consistency: Keeping data safe in the presence of crashes is a fundamental problem. *Queue*, 20(4):107–115, 2022.

[3] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *SoCC*, pages 17–28, 2016.

[4] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven Fault Injection. In *SIGMOD*, 2015.

[5] Peter Alvaro and Severine Tymon. Abstracting the geniuses away from failure testing. *ACM Queue*, 15(5):29–53, 2017.

[6] AWS. Amazon S3 now delivers strong read-after-write consistency automatically for all applications. https://aws.amazon.com/about-aws/whats-new/2020/12/amazon-s3-now-delivers-strong-read-after-write-consistency-automatically-for-all-applications/, December 2020.

[7] Peter Bailis, Peter Alvaro, and Sumit Gulwani. Research for practice: Tracing and debugging distributed systems; programming by examples. *Communications of the ACM*, 60(7):46–49, 2017.

[8] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. Debugging distributed systems. *ACM Queue*, 14(2):91–110, 2016.

[9] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A Rosenblueth, and Corentin Travers. Decentralized asynchronous crash-resilient runtime verification. In *CONCUR*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[10] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, et al. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *SOSP*, 2021.

[11] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[12] Tej Chajed, Joseph Tassarotti, M Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in Coq with Goose. In *CoqPL*, volume 2020, 2020.

[13] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[14] Himanshu Chauhan, Vijay K Garg, Aravind Natarajan, and Neeraj Mittal. A distributed abstraction algorithm for online predicate detection. In *International Symposium on Reliable Distributed Systems*, pages 101–110. IEEE, 2013.

[15] Horatiu Cirstea, Markus A Kuppe, Benjamin Loillier, and Stephan Merz. Validating Traces of Distributed Programs Against TLA+ Specifications. *arXiv preprint arXiv:2404.16075*, 2024.

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

[17] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar, Nipun Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Bin Zhang, Guofei Jiang, and Latifur Khan. LogLens: A Real-Time Log Analysis System. *ICDCS*, pages 1052–1062, 2018.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *USENIX ATC*, 2019.

[19] Envoy Proxy. https://www.envoyproxy.io/.

[20] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.

[21] etcd. https://coreos.com/etcd/.

[22] Pending conf change is not handled correctly during campaign. https://github.com/etcd-io/etcd/issues/12133, 2020.

[23] Raft: async apply ConfChange is not safe. https://github.com/etcd-io/etcd/issues/7280, 2017.

[24] ReadIndex may read stale value. https://github.com/etcd-io/etcd/issues/7331, 2017.

[25] Consistent reads are not consistent. https://github.com/etcd-io/etcd/issues/741, 2014.

[26] etcd v3.5: Runtime reconfiguration. https://etcd.io/docs/v3.5/op-guide/runtime-configuration/.

[27] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[28] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[29] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *EuroSys*, 2017.

[30] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[31] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime verification for decentralised and distributed systems. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 176–210. Springer, 2018.

[32] Yossi Gottlieb. Introducing RedisRaft, a New Strong-Consistency Deployment Option. https://redis.com/blog/redisraft-new-strong-consistency-deployment-option/, Jun 2020.

[33] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and Asserting Distributed System Invariants. In *ICSE*, 2018.

[34] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, pages 1–17, 2015.

[35] Heidi Howard, Markus A Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. Smart Casual Verification of CCF's Distributed Consensus and Consistency Protocols. *arXiv preprint arXiv:2406.17455*, 2024.

[36] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *OSDI*, 2022.

[37] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *HotOS*, 2017.

[38] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[39] Operating etcd clusters for Kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/, retrieved in April 2023.

[40] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *SOSP*, 2017.

[41] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[42] Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007.

[43] Kyle Kingsbury. Jepsen: Distribtued Systems Safety Research. https://jepsen.io/.

[44] Leslie Lamport. TLA+ Tools. http://lamport.azurewebsites.net/tla/tools.html, 2022.

[45] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *SoCC*, 2018.

[46] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *SoCC*, 2019.

[47] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. In *OOPSLA*, 2023.

[48] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI*, 2014.

[49] Rustan Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, 2010.

[50] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.

[51] Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In *PLDI*, 2020.

[52] Chang Lou, Peng Huang, and Scott Smith. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *NSDI*, Feburary 2020.

[53] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In *OSDI*, 2022.

[54] Nancy A Lynch and Mark R Tuttle. *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[55] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *SOSP*, 2019.

[56] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *EuroSys*, 2018.

[57] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *SOSP*, 2015.

[58] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.*, 2:46:1–46:24, 2017.

[59] Christopher S Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *SoCC*, pages 388–402, 2021.

[60] Ellis Michael, Doug Woos, Thomas Anderson, Michael D Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *EuroSys*, pages 1–15, 2019.

[61] Francisco Neves, Nuno Machado, and José Pereira. Falcon: A Practical Log-Based Analysis Tool for Distributed Systems. *DSN*, pages 534–541, 2018.

[62] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of formal methods at Amazon Web Services. See http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf, 2014.

[63] Diego Ongaro. Github: ongardie/raft.tla. https://github.com/ongardie/raft.tla.

[64] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*, 2014.

[65] Lingzhi Ouyang, Yu Huang, Binyu Huang, Hengfeng Wei, and Xiaoxing Ma. Leveraging TLA+ Specifications to Improve the Reliability of the ZooKeeper Coordination Service. *arXiv preprint arXiv:2302.02703*, 2023.

[66] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of the ACM on Programming Languages*, 2:1–28, 2018.

[67] Oded Padon, Giuliano Losa, Shmuel Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. In *OOPSLA*, 2017.

[68] Oded Padon, Kenneth McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Interactive Verification of Parametrized Systems via Effectively Propositional Reasoning. In *PLDI*, 2016.

[69] Oded Padon, James R Wilcox, Jason R Koenig, Kenneth L McMillan, and Alex Aiken. Induction duality: primal-dual search for invariants. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[70] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.

[71] Possible lost updates with partitions and membership changes. https://github.com/RedisLabs/redisraft/issues/17, 2020.

[72] Stale reads in normal operation. https://github.com/RedisLabs/redisraft/issues/19, 2020.

[73] Test failure : test_transfer_unexpected. https://github.com/RedisLabs/redisraft/issues/256, 2022.

[74] Loss of committed writes, duplicate elements, dueling histories. https://github.com/RedisLabs/redisraft/issues/52, 2020.

[75] Github: RedisLabs/redisraft. https://github.com/RedisLabs/redisraft.

[76] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. Hydra: Effective Runtime Network Verification. In *SIGCOMM*, 2023.

[77] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, volume 6, 2006.

[78] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999.

[79] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. *Chaos engineering*. O'Reilly Media, Incorporated, 2020.

[80] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George C. Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing Faulty Executions of Distributed Systems. In *NSDI*, 2016.

[81] Ilya Sergey, James R Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[82] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[83] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI*, 2020.

[84] The Coq Development Team. The Coq Proof Assistant. https://coq.inria.fr/.

[85] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*, 2012.

[86] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[87] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *EuroSys*, 2023.

[88] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, 2015.

[89] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *CPP 2016*, 2016.

[90] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI*, 2009.

[91] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *OSDI*, 2022.

[92] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *OSDI*, 2021.

[93] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable Runtime Verification of Shardable Networked Systems. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[94] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In *CHARME*, 1999.

[95] Pierre Zemb. Diving into ETCD's linearizable reads. https://pierrezemb.fr/posts/diving-into-etcd-linearizable/, Sep 2020.

[96] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before you change: Preventing correlated failures in service updates. In *NSDI*, 2020.

[97] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA), 2021.

[98] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. Viper: A Fast Snapshot Isolation Checker. In *EuroSys*, 2023.

[99] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed key value store. In *SIGMOD*, 2021.

[100] Apache Zookeeper. https://zookeeper.apache.org/.

[101] Data inconsistency when the node(s) with the highest zxid is not present at the time of leader election. https://issues.apache.org/jira/browse/ZOOKEEPER-1154, 2011.

A Proving apply ASAP correct

Protocol specifications in Ellsberg include an `apply_asap?` function (§3.3), whose output we use to avoid redundant exploration (§3.4.2). In this appendix, we look at proving the correctness of this function.

We start by redefining the `apply_asap?` requirements in terms of *dominating states*.

Dominating state: We say a state s dominates state s' (or $s \sqsupseteq s'$) if and only if for any message m, s' being m-inducing implies s is m-inducing.

The `apply_asap?` function takes as input state s and an event e, and returns true if the following conditions are met:

- For all schedules Σ where $e \in \Sigma$, $apply(apply(s,e),\Sigma - e) \sqsupseteq apply(s,\Sigma)$.
- For all schedules Σ where $e \notin \Sigma$, $apply(apply(s,e),\Sigma) \sqsupseteq apply(s,\Sigma)$.

We now turn to describing our approach for proving a supplied `apply_asap?` function meets this requirement.

In our specifications (and indeed in all specifications), an $apply_asap?(s,\ e)$ consists of a set of constraints $C = \{c_0, c_1, ..., c_n\}$ over s and e, and returns true if any constraint c_i holds. For example, in Raft's $apply_asap?$ function one of the constraints deals with append entry responses, and returns true if e is an incoming *AppendEntryResponse* message with term t and the simulation state s's term is larger than t ($s.term > t$).

We prove `apply_asap?` correct by analyzing each constraint $c \in C$. More formally

Theorem 1 *An* `apply_asap?` *function specified as the set of constraints C is correct if and only if for all $c \in C, s \in S, e \in E$, $c(s,e) = true$ implies that the following two properties hold for s,e:*

- *For all schedules Σ where $e \in \Sigma$, $apply(apply(s,e),\Sigma - e) \sqsupseteq apply(s,\Sigma)$.*
- *For all schedules Σ where $e \notin \Sigma$, $apply(apply(s,e),\Sigma) \sqsupseteq apply(s,\Sigma)$.*

Proving these two conditions mechanically is challenging, as it requires reasoning about the set of all schedules Σ. Therefore, we instead prove that if four simpler properties (that can be mechanically checked) hold for any $c \in C$, then c satisfies Theorem 1:

Condition A.1 ***Order Preserving.*** *The protocol should be such that for any event e, states s_1, s_2, if $s_1 \sqsupseteq s_2$, then $apply(s_1,e) \sqsupseteq apply(s_2,e)$.*

Condition A.2 ***Constraints Preserving.*** *For any event e,e', and any state s, if $c(s,e) = true$ then $c(apply(s,e'),e) = true$.*

Condition A.3 ***Expansion.*** *For any event e and any state s that satisfies $C(s,e)$, $apply(s,e) \sqsupseteq s$*

Condition A.4 ***Reorder Safety.*** *For any event e,e', state s that satisfies $C(s,e)$, $apply(apply(s,e),e') \sqsupseteq apply(apply(s,e'),e)$*

Assume c satisfies the four conditions above, we use case splitting to prove that Theorem 1 must hold:

Case 1 $e \notin \Sigma$. Given **Statement 3**, we have $apply(s,\ e) \sqsupseteq s$. Given **Statement 1**, we have $apply(apply(s,e),\Sigma) \sqsupseteq apply(s,\Sigma)$. ■

Case 2 $e \in \Sigma$. Suppose $\Sigma = \Sigma' + e + \Sigma''$ where $\Sigma' = e_1, e_2, ..., e_k$. To simplify the expressions, we denote $apply(s,e)$ by $e(s)$. We first prove that:

Lemma 1

$$e_k \circ ... \circ \underline{e_{i+1}} \circ e \circ e_i \circ ... \circ e_1(s) \sqsupseteq e_k \circ ... \circ \underline{e \circ e_{i+1}} \circ e_i \circ ... \circ e_1(s)$$

Let $s' = e_i \circ ... \circ e_1(s)$, according to **Statement 2**, given that $C(s,e)$ is satisfied, we have that $C(s',e)$ is also satisfied. Then, according to **Statement 4**, we have $e_{i+1}(e(s')) \sqsupseteq e(e_{i+1}(s'))$. Then, given **Statement 1**, we can prove **Lemma 1**.

Therefore, we can further prove this by inductively applying **Lemma 1**:

$$e_k \circ ... \circ e_2 \circ e_1 \circ e(s) \sqsupseteq e \circ e_k \circ ... e_2 \circ e_1(s)$$

Then, let $s_1 = e_k \circ ... \circ e_2 \circ e_1 \circ e(s)$ and $s_2 = e \circ e_k \circ ... e_2 \circ e_1(s)$, given **Statement 1**, we have $\Sigma''(s_1) \sqsupseteq \Sigma''(s_2)$, where $\Sigma''(s_1) = apply(apply(s,e),\Sigma - e), \Sigma''(s_2) = apply(s,\Sigma)$. ■

B Mechanically Checking `apply_asap?`

We used Z3 to check the `apply_asap?` functions in our specification satisfy the four conditions A.2–A.4. To do so, we used the following Z3 specifications:

1) $apply(s,e)$ of each event.
2) $dominates(s_1,s_2)$. It returns whether $s_1 \sqsupseteq s_2$
3) $C(s,\ e)$. It specifies the constraints between e and s. When $C(s,e) = True$, it means we can apply e on s asap.

Among these, we specify $apply(s,\ e)$ using each protocol's TLA+ specification. We show the *dominates* and C specification for Raft below:

```
fn dominates(s1: State, s2: State) {
    //all other states are the same, except
          commit_index, match_index and vote_granted
    //s1's commit_index is not smaller than s2
    //each slot
          of s1's match_index is not smaller than s2
    //if s1'state is candidate, then s1's
          vote_granted is the same as s2. Otherwise,
        s1's vote_granted set is the superset of s2.
    return s1.term == s2.term && ... &&
           s1.commit_index >= s2.commit_index &&
           [for i in range(SERVERS):
            s1.match_index[i]>=s2.match_index[i]] &&
           (s1.state == CANDIDATE ?
                equal(s1.
                     vote_granted, s2.vote_granted) :
                subset(s2.vote_granted
                     , s1.vote_granted))
}
```

```
fn C(s: State, e: Event) {
    //we can apply e asap if this is an event from previous term
    //or it is an AppendEntriesResponse in current state's term
    //or this is a RequestVoteResponse in current state's term, and the state now is not CANDIDATE
    //or this is a AppendEntryRequest in current state's term, but it broadcasts the entires that are a subsequence of the local log.
    //or this is a RequestVoteRequest in current state's term, but server has already voted for someone.
    return e.term < s.term ||
           (e.term == s.term &&
               (e.type == AppendEntriesResponse ||
               e.type == RequestVoteResponse &&
                   s.state != CANDIDATE) ||
               e.type == AppendEntryRequest
                   && s.state == FOLLOWER
                   && subseq(e.entries, s.log) ||
               e.type == RequestVoteRequest
                   && s.votedFor != None)
}
```

The *dominates* and *C* specification for ZAB are below:

```
fn dominates(s1: State, s2: State) {
    //all other states are the same, except last_committed, last_processed, ackid of each log entry, and ack_ld_recv
    //s1's last_committed and last_processed are not smaller than s2
    //each entry's ack set of s2 is the subset of s1
    //if in BROADCAST state, s2's ack leader set is the subset of s2. Otherwise, s2's ack leader set is the same as s1.
    return s1.accepted_epoch
          == s2.accepted_epoch && ... &&
           s1.last_committed >= s2.last_committed &&
           s1.last_processed >= s2.last_processed &&
           [for i in range(ENTRIES):
            subset(s2.Log[i].ack, s1.Log[i].ack)] &&
           (s1.zab_state == BROADCAST?
            subset(s2.ack_ld_recv, s1.ack_ld_recv):
            equal(s1.ack_ld_recv, s2.ack_ld_recv))
}

fn C(s: State, e: Event) {
    //we can apply e asap if this is an ACK
    //or it is an ACKLD in outdated epoch compared with current state's accepted_epoch
    //or ACKLD in current accepted_epoch but current state already has received quorum ACKLD
    return e.Type == ACK ||
           (e.Type == ACKLD &&
                (Epoch(e.Zxid) < s.accepted_epoch
                    || has_quorum(s.ack_ld_recv)))
}
```

C Protocol Specifications

Below, we provide additional information about the protocol specification used in our evaluation, including detailed breakdown of their lengths and additional information about how they were derived.

C.1 Specification Length

Table 3 shows lines of code for each portion of the Ellsberg specifications used in our evaluation. Similar to §4 we also report the length of specification used by Microsoft's CCF project for validating Raft [35] and an implementation derived ZooKeeper specification [38].

Component	Etcd	Redis Raft	ZooKeeper
State	45	47	102
apply	345	312	433
equal	64	64	73
infer_inducing	298	271	236
reachable	37	37	63
apply_asap?	39	39	36
lookahead_type	9	9	6
Other code	115	115	40
Total (Ellsberg)	952	894	989
Baseline	1503	1503	1615

Table 3: *Lines of code in Ellsberg specification when compared to those used by existing tools. For baselines, we use the Raft specification from Microsoft CCF [35] Etcd and RedisRaft, and a recent implementation derived TLA+ specification [38] for ZooKeeper.*

C.2 Raft

We evaluated our approach and Ellsberg using two Raft implementations: Etcd [21], a distributed key-value store that is widely deployed, often as a core component of Kubernetes [39] and other orchestrators; and Redis Raft [32, 75], a module that adds consistent replication to the Redis key-value store. In this section, we describe the specification we used.

Protocol Specification. Our initial approach for producing a specification was to derive one from Ongaro's TLA+ Raft specification [63], since this closely mirrored the protocol described in the Raft paper [64], and we assumed was the distributed protocol implemented in practice. However, we found this was not the case, both implementations we evaluated implement a protocol which makes several changes to the original protocol, and the specification we use includes these changes. The most significant changes from the protocol in the paper affect read-only operations and reconfiguration, and we describe both below.

Read-only operations: Both of the implementations that we evaluated provide linearizable reads, but neither logs read-only operations. Both require the leader to process all read-only requests, and the leader computes the return value using its current state. To ensure linearizability, a node that is processing a read-only request must first assert that it is still the leader, *i.e.*, it needs to check that its term is the same as a quorum's term [95]. Both implementations use the same approach to assert leadership: when a node (that believes it is currently the leader) receives a read-only request, it associates a *request ID* with it and includes this request ID in a heartbeat request (as a reminder, Raft uses empty append entry messages for heartbeats) to other nodes in the cluster. The node then waits until a quorum (majority) of nodes have positively acknowledged the heartbeat message (showing that a quorum agrees on the term, and that the node was the leader when the request

was received), before responding to the read-only request.[2]

Reconfiguration protocol: The other area where both implementations differ is in how they implement cluster reconfiguration: Etcd uses different protocols to handle reconfiguration that adds or removes a single node (the Etcd documentation [26] seems to prefer this protocol) and ones that add or remove multiple nodes (where Etcd largely uses the original joint quorum protocol with minor changes); while Redis Raft only allows the addition or removal of a single node.

Both implementations use a similar (but not identical) single node reconfiguration protocol that avoids the original protocol's use of joint quorums, because in this case quorums are guaranteed to have intersection before and after configuration changes are applied. Therefore, the reconfiguration protocol in this case merely requires that the leader replicate a reconfiguration command to all followers. The two implementation differ on when they apply the reconfiguration command: Redis Raft nodes apply the command as soon as it is added to their log, while Etcd treats reconfiguration commands like any other command and applies them asynchronously after they have been committed. Etcd's approach can pose a safety problem, and recent versions of Etcd change the request vote protocol to ensure that nodes with replicated but unapplied reconfiguration entries do not send request vote messages (and thus do not try to become leaders). This restriction also applies to multi-node reconfiguration in Etcd, and one of the bugs we reproduce and detect with Ellsberg (§6) is due to errors implementing this change.

Beyond differences due to changes in Raft, our specification differs in another crucial way: we model the behavior of messages sent by clients (*e.g.,* `get` and `put`) while Ongaro's specification doesn't.

Our evaluation uses similar protocol specifications for both implementations, and they only differ in when reconfiguration commands take effect.

C.3 Zookeeper Atomic Broadcast

Obtaining a specification for Zookeeper atomic broadcast (ZAB) [38] as implement by ZooKeeper [100] was a bit easier. While the TLA+ specification included in the Junqueira et al's [38] paper no longer reflects the implementation, a recent pre-print [65] has produced an updated TLA+ specification from the implementation, and we used this specification with minor changes.

ZooKeeper is also easier to model, since it does not support linearizable reads, and we did not need to consider modifications for this.

D Bug Descriptions

Below, we describe the bugs used in our evaluation (§6.2), their root-cause and how we detected them.

etcd-741 [25] is a bug that predates Etcd incorporating the linearizable read protocol we described in §C.2. In older versions, an Etcd leader responded to read-only requests with its current value, without asserting that it was the leader, thus violating linearizability. We reproduced this bug by removing the linearizable read logic from Etcd. Ellsberg detected the bug when it observed the leader responding to a client's request before receiving positive heartbeat acknowledgments from a quorum.

etcd-7280 [23] and **etcd-12133** [22] are bugs in Etcd's reconfiguration protocol. As we noted previously (Appendix C.2), Etcd supports two different reconfiguration protocols (a single-node reconfiguration that does not require joint-quorums, and a more general version that uses joint-quorums). Etcd also has two different command types for reconfiguration, one which only allows single node reconfiguration (V1), and one that can be used to trigger both single-node reconfiguration and joint-quorum based reconfiguration (V2). Regardless of command or protocol, reconfiguration takes effect when the command is *applied* by a node. Furthermore, Etcd does not allow a node with an unapplied reconfiguration command in its log to send a request vote message (*i.e.,* to transition to being a candidate).

Both of these bugs are caused by cases where Etcd incorrectly allowed a node with an unapplied reconfiguration command to become a candidate. In the case of etcd-7280 this was caused because Etcd switched to applying reconfiguration requests asynchronously, but assumed that a single reconfiguration request could be in progress at a given time. If more than one reconfiguration request was in progress, then Etcd would allow a node to become candidate after applying the first change, resulting in two leaders being elected simultaneously. Similarly, etcd-12133 was caused by a bug in how Etcd counted the number of pending reconfiguration messages: the bug meant that Etcd did not consider unapplied V2 reconfiguration commands, resulting in a split-brain problem where two leaders are elected for the same term.

In both cases, Etcd's safety invariants say that any inducing states for a request vote message has no unapplied reconfiguration commands. Thus, one might expect that we detect these bugs when a node first sends out a request vote message. However, this is not the case: Ellsberg does not know whether a node has applied a reconfiguration command, since commands are applied asynchronously by a background thread. Instead, when Ellsberg see the request vote message, it merely infers that the reconfiguration command has been applied. However, Ellsberg reports a bug when it later observes that the node has transitioned to being a leader (*e.g.,* by observing outgoing append entry messages) without having received votes from a quorum active in the latest configuration.

etcd-7331 [24] is a bug caused by interaction between Etcd's linearizable read protocol and Raft's leader election protocol that can result in a newly elected leader returning a stale value in response to a read. When an Etcd leader receives a read-only request, it records the current commit index, and checks that

[2] What we have described corresponds to Etcd's `ReadIndex` method. Etcd also implements a lease-based mechanism for linearizable reads, but we neither modeled this mechanism nor use it in our evaluation.

it is still the leader (by sending a heartbeat or append entry message, and waiting for responses from a quorum). However, a newly elected leader's commit index might be smaller than the previous leader's commit index. This is because while Raft's leader election protocol requires that the new leader's log contain all committed entries (and in fact be the most up-to-date log in a quorum), it cannot guarantee that the commit index (which leader's update asynchronously) is up-to-date.

Furthermore, for safety, Raft does not allow a leader to update its commit index until it has replicated an entry in the current term [64, §5.4.2] to a quorum, *i.e.,* it can only update the commit index to point to entries in the current term. Consequently, a read-request handled by a newly-elected leader might return a previous value, violating linearizability.

Etcd fixed this protocol bug in its linearizable read protocol by requiring that leaders commit an entry in the current term before initiating a quorum check for a read-only request. Ellsberg uses the same approach to detect this bug: it raises an alarm if it sees the leader is performing a quorum check for a read-only request (which it can detect because heartbeat or append entry messages used for quorum checks include a request ID for the read request) before committing an entry.

Zookeeper-1154 [101] is data inconsistency bug, where after leader election, nodes disagree on the log. In the leader election protocol implemented by ZooKeeper, a node needs to synchronize its log with all other nodes. To do so, the leader collects the largest entry ID (*ZXID*s) from each follower, and compares the follower's latest ZXID to the latest ZXID in its own log. If the follower's latest ZXID differs, then the leader needs to first have the follower truncate its own log (to the last ZXID on which both agree) and then send missing entries to the follower. Similar to other protocols, ZXIDs in ZooKeeper are monotonically increasing. When implementing the leader election protocol, the ZooKeeper developers introduced a bug and assumed that if the ZXID sent by the follower was *smaller* than the latest ZXID in the leader's log, then the follower must contain a subset of the leader's log entries. The leader would then avoid truncating the follower's log, and immediately send new entries. However, this is unsafe since the follower's log might contain an entry that has a lower ZXID but is not present in the leader's log, resulting in an inconsistent log. Our simulation detects this when the leader sends out the DIFF message since the follower's latest ZXID is not contained in the leader's log, and thus no inducing state for the DIFF message can be reached.

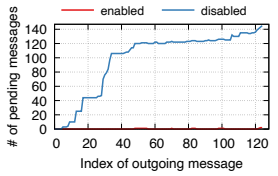
RedisRaft-17 [71] is also a reconfiguration bug: Redis Raft does not allow concurrent reconfiguration requests, and will accept at most one reconfiguration request at a time (rejecting the other). It considers reconfiguration to be complete once the command has been committed to quorum. Due to a bug, it was not checking this correctly. Detecting this bug with Ellsberg was relatively simple: our inference function says that the inducing state for a successful reconfiguration request is one where the reconfiguration command was added to the log. However, our transition function does not allow transitioning into a state with two uncommitted reconfiguration requests, and thus Ellsberg cannot find a reachable inducing state for the second successful reconfiguration response.

RedisRaft-19 [72] is a bug in Redis Raft's linearizable read implementation. Unlike Etcd-741, this bug occurs despite nodes asserting leadership before responding to clients, and is caused by the node being unaware of the current commit index. In Raft, a leader may not know what log entries were committed by the previous leader (in the previous term). Furthermore, Raft requires that leader's only commit entries from previous terms after they have added an entry from the current term to the log [64, §5.4.2]. Therefore, linearizability might be violated if a Raft leader respond to a read-only request before committing an entry in the current term. When writing our protocol specification, we modeled the linearizable read-only protocol by associating a linearization index with each read-only request: a request's linearization index is the log's commit index if the last entry committed in the current term, and is the last log index otherwise. Furthermore, our inference function requires that the *m*-inducing state for a read-only response have committed entries up to the request's linearization index. Therefore, Ellsberg flags a bug in this case, because no valid *m*-inducing states exists for a non-linearizable return value.

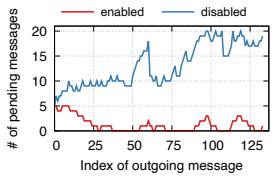
RedisRaft Unreported. This previously unreported bug was discovered by Ellsberg when we ran fuzzing loops to find bugs. The bug is due to an additional complication in Redis Raft's reconfiguration protocol: when adding a node, Redis Raft first issues a reconfiguration command to add a *non-voting* node that cannot participate in quorums. After this command has been committed, Redis Raft next issues a second reconfiguration command to switch the non-voting node to a voting node, allowing it to participate in quorums. However, due to a bug, Redis Raft acknowledges the client reconfiguration request once the non-voting reconfiguration command has been successfully replicated. Consequently, Redis Raft reports that reconfiguration is complete before the quorum size has grown, and thus before the cluster can tolerate additional failures. In this case, an ill-timed failure that occurs between the two commands can render the cluster unavailable, despite the administrator believing that this should not occur. Our specification models both steps of the reconfiguration process, and flagged the bug because it observed the leader acknowledging the client's reconfiguration request before the second step had completed.

RedisRaft-256 [73] is a reconfiguration bug related to the unreported bug described above. This bug resulted in leader acknowledging a client reconfiguration command that adds a node before replicating it to a quorum, and can result in situation where the client reconfiguration command is never updated. Ellsberg alerts on this bug when it observes the leader send a client response without having received append entry responses from a quorum, *i.e.,* before it has been committed.

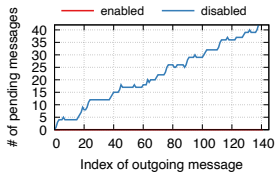
RedisRaft-52 [74] is a bug that leads to data-loss, and is due



(a) *Etcd*

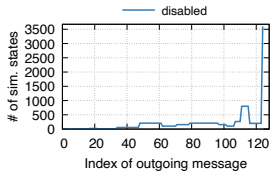


(b) *Zookeeper.*

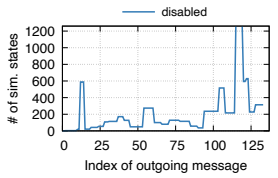


(c) *Redis Raft*

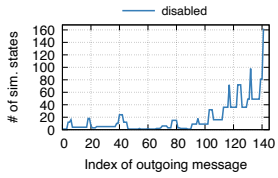
Figure 5: `apply_asap?` *avoids redundant exploration: graphs show pending messages in a 5-node cluster after checking the* n^{th} *outgoing message* `apply_asap?` enabled *and* disabled.



(a) *Etcd.*



(b) *Zookeeper.*



(c) *Redis Raft.*

Figure 6: `apply_asap?` *reduces memory requirements: graphs show number of simulation states* $|S|$ *in a 5-node cluster with* `apply_asap?` *disabled.* ($|S| = 1$ *when* `apply_asap?` *is enabled.)*

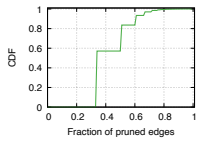
to a performance optimization implemented by Redis Raft where the last few log entries are stored in a cache. When a node appends an entry it adds it to both the cache and the underlying log. When reading an entry, the node first checks the cache, and only refers to the underlying log if it is not in the cache. A bug in Redis Raft meant that it did not update the cache when deleting entries from the log. However, Raft requires to delete log entries in some scenarios, *e.g.,* when a new leader has to overwrite uncommitted entries. This bug can therefore result in an inconsistent state machine replica, which can lead to unexpected node behavior, *e.g.,* rejecting append entry or vote requests that it should have accepted. We reproduced a case where an append entry request was incorrectly rejected due to an old conflicting entry in the cache. The Ellsberg specification detected a divergence when checking the append entry response and raised an alert.

E Effect of Ellsberg's Design Choices

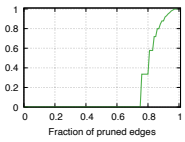
Our evaluation (§6) presented end-to-end performance results for Ellsberg. In this appendix, we evaluate the impact of `apply_asap?` and the `reachable` function on Ellsberg's performance.

E.1 Benefits from `apply_asap?`

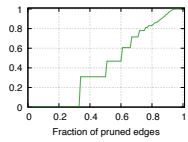
Next we evaluate the effect of using `apply_asap?` to avoid redundant exploration. As we discussed in §3.4.2, `apply_asap?` is used when processing incoming messages (Listing 3 line 24) and during breadth first search (Listing 4 line 25) to reduce the number of pending messages, and the size of the simulation state set S. This improves `find_reachable`'s runtime and reduces Ellsberg's memory requirement.



(a) *Etcd.*



(b) *Zookeeper.*



(c) *Redis.*

Figure 7: *CDF of fraction of edges pruned by the* `reachable` *check (Listing 4 line 22). All experiments are run on a 5-node cluster using the mixed workload.*

We evaluate improvements in `find_reachable`'s runtime by comparing the number of pending messages when `apply_asap?` is enabled to when it is disabled. In Figure 5, we show the number of pending messages as a function of the number of outgoing messages processed. We report results from a 5-node cluster when running the balanced workload, the results were similar for other workloads. We observe that when `apply_asap?` is disabled, the number of pending messages grows as Ellsberg processes more of the trace. By contrast, enabling `apply_asap?` limits the number of pending messages: in our workloads we observe between 0 and 5 pending messages.

Using `apply_asap?` also reduces the number of simulation states in S, improving both performance (results in fewer calls to `find_reachable`) and memory requirements. We evaluated this improvement by comparing the size of S as a function of number of output messages processed with `apply_asap?` enabled and disabled. Results from running a balanced workload on a 5-node cluster, when `apply_asap?` is disabled, are shown in Figure 5. When `apply_asap?` is enabled, we found that S contained a single simulation state ($|S|$=1). By contrast, when `apply_asap?` is disabled the number of states can grow large, with an average of ∼444 states for Etcd, ∼478 states for Zookeeper, and ∼43 for Redis Raft.

In sum, these results show the importance of `apply_asap?` for limiting Ellsberg's memory usage and in achieving our performance goals.

E.2 The `reachable` function's impact

Ellsberg uses the specification's `reachable` function to to avoid exploring schedules that cannot lead to m-inducing simulation states (§3.4.1, Listing 4 line 22). As we explained in §3.4.1, we need to use this function to ensure that checking an outgoing message terminates, and we cannot turn it off.

Therefore, we evaluate the effectiveness of reachability checks by drawing a CDF of the fraction of `reachable` calls that return false (and thus prune exploration). In Figure 7, we show results for the three DPIs running on a 5-node cluster evaluated using the balanced workload. As one would expect, the results depend on the protocol and implementation: pruning is most efficient for Zookeeper because it assumes in-order deliver. The in-order delivery assumption ensures that the number of pending messages is bound by the number of nodes and the protocol structure allows the `reachable` function to prune more than 75% of possible paths. While pruning is less effective for Etcd and Redis Raft, we observe that even in

this case, between 40–80% of calls to the `reachable` function return false. These results show that while Ellsberg can work with specification that over-approximate reachability, this approximation can come at a performance cost.

CEGS: Configuration Example Generalizing Synthesizer

Jianmin Liu[1], Li Chen[2], Dan Li[1], Yukai Miao[2]
[1]*Tsinghua University* [2]*Zhongguancun Laboratory*

Abstract

Network configuration synthesis promises to increase the efficiency of network management by reducing human involvement. However, despite significant advances in this field, existing synthesizers still require much human effort in drafting configuration templates or coding in a domain-specific language. We argue that the main reason for this is that a core capability is missing for current synthesizers: identifying and following configuration examples in configuration manuals and generalizing them to arbitrary topologies.

In this work, we fill this capability gap with two recent advancements in artificial intelligence: graph neural networks (GNNs) and large language models (LLMs). We build CEGS, which can automatically identify appropriate configuration examples, follow and generalize them to fit target network scenarios. CEGS features a GNN-based Querier to identify relevant examples from device documentations, a GNN-based Classifier to generalize the example to arbitrary topology, and an efficient LLM-driven synthesis method to quickly and correctly synthesize configurations that comply with the intents. Evaluations of real-world networks and complex intents show that CEGS can automatically synthesize correct configurations for a network of 1094 devices without human involvement. In contrast, the state-of-the-art LLM-based synthesizer are more than 30 times slower than CEGS on average, even when human experts are in the loop.

1 Introduction

Network configuration management plays a central role in network operations (NetOps). Manually managing configurations is costly and susceptible to human errors that can cause severe network downtime [13, 14, 35]. This is especially important for modern large-scale networks with more than tens of thousands of devices. To increase NetOps efficiency and correctness, researchers have made continuous efforts in configuration synthesis [1, 6, 7, 16, 17, 22, 39, 43, 44, 46]. With the descriptions of intents and network topology from network operators as input, configuration synthesis systems are expected to generate correct configurations for network devices to achieve the intents.

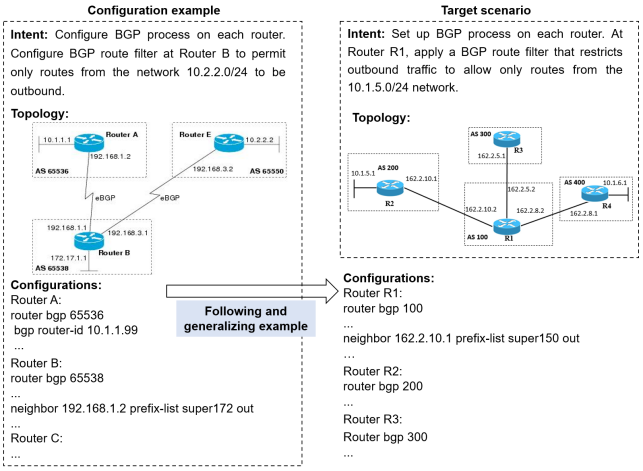


Figure 1: An instance of EFG in configuration synthesis.

Despite considerable advancements in configuration synthesis, existing proposals still require significant human effort. For example, NetComplete [17] requires operators to manually provide configuration templates where some parameters are left symbolic for the synthesizer. Robotron [44] also requires expert effort to construct suitable templates for configuration generation. For Aura [39], operators must learn a new domain-specific language, and code in this language to express their high-level intents.

The main reason for human involvement, we believe, is that a core capability is missing from all existing synthesizers: example following and generalization (EFG). In other words, they cannot identify, follow, and generalize the examples provided by device documentation, and thus must rely on human experts to perform the task. We show an example of EFG in Figure 1, where the configuration example[1] from the Cisco device documentation has a topology with three nodes for an intent, and the target scenario has a topology with four nodes. To make use of an existing synthesizer, such as NetComplete, experts must first look through the device

[1] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/configuration/xe-16/irg-xe-16-book/connecting-to-a-service-provider-using-external-bgp.html

documentation [2] for this example, identify the appropriate configuration snippets, assign the snippets to different devices, and mark symbolic parameters for synthesizers to fill, so that configuration templates can be generated for each device.

EFG is currently a human process, because it requires three sub-capabilities: understanding of topologies in both the example and target scenario, comprehension of natural language description of intents, and code generation of device configuration templates. However, with the recent advances of graph neural networks (GNNs) and large language models (LLMs), we believe that EFG can be automated. This is because 1) GNNs have shown remarkable capability in modeling node characteristics and similarities in many related applications, such as knowledge graph alignment [56,62] and social recommendation [18,55]; and 2) LLMs have demonstrated unprecedented capabilities in comprehending human intents, producing contextually relevant content, and even writing sequential programs in systems such as CoPilot [23] and Jigsaw [28].

Inspired by these achievements, this paper investigates the following research question: *Can we automate EFG for configuration synthesis?* We answer this question affirmatively with the design and implementation of a configuration synthesis system that we call CEGS (Configuration Example Generalizing Synthesizer). This involves solving three main challenges:

C1 Accurately identifying relevant configuration examples from the manual. Given a target network scenario consisting of user intents and target topology, we must find the most relevant configuration examples from the manual. Thus, we need to design a Querier to accurately identify the configuration example that closely aligns with the user intent and target topology from the manual.

C2 Associating the devices in the target topology with those in the example. Establishing accurate associations between devices in the target and example topologies is crucial for EFG. To automate this, we need to design a Classifier to accurately map the nodes in the example to the devices in the target topology, so that the configuration snippets for each node in the example can be assigned to the devices in the target topology.

C3 Accelerating synthesis without compromising correctness. As COSYNTH [36], a pioneering work that applies LLMs to configuration synthesis, indicates, an LLM usually cannot produce the correct configurations for all devices with its first attempt, and it requires multiple iterations (loops) for error correction. Each loop involves call to the LLM inference, and in some scenarios (§4), we show that, without human intervention, COSYNTH is unable to generate the correct results with as many as 300 loops. Since LLM inference is costly, we intend to design an efficient synthesis method that minimizes the number of loops without compromising correctness.

[2] Documentation includes configuration manuals, guides, *etc.*.

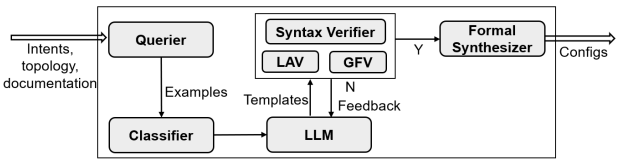


Figure 2: High-level workflow of CEGS.

CEGS comprehensively addresses the above challenges. Taking natural language intents, a target network topology, and the device documentation as input, CEGS attempts to synthesize the correct configuration for network devices in the target topology without human involvement. CEGS utilizes GNN and LLM to achieve EFG automation, and has six core components, namely Querier, Classifier, Syntax Verifier, Local Attribute Verifier (LAV), Global Formal Verifier (GFV), and Formal Synthesizer. The workflow (Figure 2) is as follows: firstly, the Querier identifies the most relevant configuration examples from the device documentation that align closely with the user intents and target topology. Secondly, the Classifier maps the nodes in configuration examples to appropriate devices in the target topology. Thirdly, CEGS leverages an LLM to generate configuration templates for all devices based on configuration examples, leaving the specific parameters of network policies symbolic. Then, CEGS utilizes the Syntax Verifier, LAV, and GFV to verify the templates. Finally, the Formal Synthesizer takes the intents, the topology, and the templates, and attempts to fill the symbolic parameters with values to achieve the intents. In this process, CEGS automatically feeds any errors caught by the Syntax verifier, LAV, GFV, and the synthesizer back to the LLM for error correction.

We enable this workflow with the following contributions:

- We design the Querier to identify relevant examples from the device manual, solving challenge C1. The Querier adopts a two-stage recommendation strategy. The first stage uses an LLM to process the natural language user intent, and then measures the similarity of the user intent to the descriptions of examples. In the second stage, the Querier further incorporates topological similarity information with a GNN, and determines the most appropriate example. In our evaluation with 90 configuration synthesis scenarios, the Querier achieves 100% accuracy in recommending examples.
- We design the Classifier to establish accurate associations between devices in the target and the example topologies, solving challenge C2. Specifically, the Classifier considers both textual and topological similarities to determine the relationships. In our evaluation with 90 configuration synthesis scenarios, CEGS also achieves 100% accuracy in terms of device classification.
- We propose an efficient LLM-driven synthesis method to accelerate synthesis without compromising correctness, solving challenge C3. Compared to prior work, our key innovations are: 1) we guide the LLM configuration generation with appropriate examples from device documen-

tations, instead of relying solely on the "memory" of the LLM; 2) we stipulate the LLM to only generate templates, because they are bad at logical reasoning and analysis [33], thus they struggle to generate specific values to achieve user policy intents; and 3) we instead use Formal Synthesizer to fill in templates, which is significantly faster than calling LLM multiple times with expert in the loop [36].

- We evaluate CEGS using Static, OSPF, and BGP routing intents and a variety of real-world topologies ranging from 20 to 1094 routers. Our results show that CEGS can automatically synthesize correct configurations based on examples, without any human involvement. Compared to state-of-the-art LLM-based synthesis systems COSYNTH, CEGS automatically synthesizes correct configurations for BGP no-transit intents within 2 minutes, while COSYNTH fails to generate correct configurations within one hour. Even with human expert in the loop, COSYNTH is more than 30 times slower than CEGS.

Limitations. This paper introduces the EFG capability to network configuration synthesizers. Enabling EFG does not solve all the challenges in configuration synthesis, and our limitations are as follows. Firstly, we are constrained by the coverage of device documentations: if there is no example of a user's intent, CEGS cannot produce a satisfactory result. When this happens, users of CEGS should ask the vendor for related examples. Secondly, CEGS is also subject to the capability of the Formal Synthesizer. In other words, to guarantee correctness, we can only support intents that are also supported by the Formal Synthesizer we use (NetComplete [17]). We consider the extension of the capabilities of the Formal Synthesizer to be orthogonal to CEGS, and do not claim any contribution in this aspect. Thirdly, the generated configuration is correct only when an LLM can correctly convert a natural language intent to a formal intent. While our evaluation demonstrates 100% accuracy in intent formalization, this result is specific to our evaluation dataset. The generalizability of this accuracy to other datasets or scenarios is not guaranteed and remains an area for further investigation.

2 BACKGROUND AND MOTIVATION

2.1 Current synthesizers require expert effort

Generating and editing configurations of devices based on ever-changing operational intents are daily tasks of NetOps engineers, and configuration synthesizers promise to reduce the load on human operators by generating correct configurations automatically. Recently, many efforts have been made in configuration synthesis [1, 6, 7, 16, 17, 22, 39, 43, 44, 46]. Despite significant advancements, existing synthesizers still require human involvement in the synthesis process.

Existing configuration synthesizers fall into two types: Domain Specific Language (DSL)-based synthesis, and template-based synthesis. Both types require significant expert effort to work. DSL-based synthesizers require experts to express high-level intents using DSLs, such as Datalog (SyNet [16]), RPL (Aura [39]), and regular expressions (Propane [6]). The learning curve and the amount of code needed for large-scale networks are both intimidating for NetOps engineers.

In this paper, we focus on reducing the human efforts for the template-based synthesizers. These systems take a network topology, high-level intents, and configuration templates as input and automatically fill the blanks in the templates. For example, NetComplete [17] autocompletes the configuration templates to correct network-wide configurations. NetComplete uses SMT constraints to model the semantics of network configurations at the level of individual route advertisements and routing policies. Specific configuration values (e.g., BGP policy parameters) are made symbolic, allowing the SMT solver to find a set of values that satisfy the desired policy.

For template-based synthesizers, we observe that the process which requires the most human effort is the process of converting a set of high-level intents into a set of templates for all devices in the topology. For NetOps experts, this process typically involves identifying appropriate configuration examples in the device documentation, assigning the snippets in the examples to devices, and marking symbolic parameters based on intents. We name this process as *example following and generalization* (EFG).

We believe that the automation of EFG can save considerable expert effort. However, automating EFG is challenging, because it requires three sub-capabilities: understanding of topology in both the example and target scenario, comprehension of natural language description of intents, and code generation of device configuration templates. Fortunately, the recent achievements of GNNs and LLMs in AI give us hope in automating these human capabilities. Next, we briefly introduce these two technologies and their prior applications in the area of configuration synthesis.

2.2 LLMs & GNNs for configuration synthesis

GNNs and LLMs have been explored separately in the literature on configuration synthesis, but they have never been combined, to the best of our knowledge.

Graph Neural Networks (GNNs) are tailored to process graph-like data, thus can be useful for topology understanding for EFG. GNNs have been shown to be capable of learning node-level and graph-level representations by aggregating information from neighboring nodes and edges. They are widely applied in various reasoning tasks on graphs [58, 60]. In relation to configuration synthesis, ConfigReco [24] is a configuration recommendation tool that uses GNN. It features a knowledge graph to model network configurations and adopts GNN to recommend configuration templates for manual configuration. However, ConfigReco can only recommend high-likelihood configuration snippets to NetOps engineers, and it cannot generalize the examples to target topologies.

LLMs [3,4,9,47] have demonstrated impressive capabilities in answering questions [41, 42], understanding user intents [34,37,52], comprehending manuals [11,32], generating commands [20, 53, 59], and even composing sequential programs

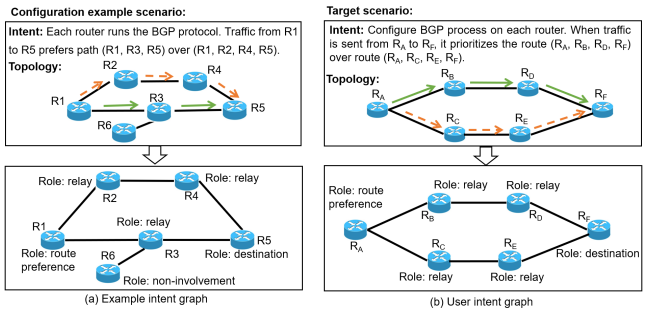


Figure 3: Configuration example and target scenarios, along with the corresponding intent graphs.

in systems like CoPilot [23] and Jigsaw [28]. All of these capabilities show immense potential for automating EFG in configuration synthesis. Using LLMs for configuration synthesis has already been explored in previous work. Rajdeep et al. [36] designed COSYNTH, a configuration synthesis system that uses LLMs to generate network configurations that are aligned with the high-level intents. NETBUDDY [49] [51], a LLM-based synthesis system, leverages LLM to simplify configuration synthesis by splitting configuration generation into multiple fine-grained steps. However, both COSYNTH and NETBUDDY directly employs LLMs to generate concrete configurations based on the specified intents without providing any configuration examples. Thus, COSYNTH and NETBUDDY still require human involvement in the loop to guide the synthesis. For example, as demonstrated in §4.3, without human involvement, COSYNTH and NETBUDDY cannot successfully compile intents into correct network configurations with as many as 300 loops. PreConfig [31] is a pretrained LLM designed for automating network configuration. However, PreConfig only can generate configuration snippets, and it still unable to synthesize network-wide configuration for target network scenarios to achieve specific high-level intents.

2.3 Motivating CEGS's Design Decisions

With the above investigation, we design CEGS which leverages GNN and LLM to automate EFG. To this end, we address the three main challenges. In the following, we list each challenge as **C#** and its related design choice(s) *D#*.

C1. Accurately identifying relevant configuration examples from the device documentation. Given a target network scenario consisting of a network topology and a set of high-level intents, we need to first identify pertinent configuration examples from the device documentation that closely align with the user intent and target topology.

D1. Transforming the recommendation problem into a graph similarity problem. We observe that the most appropriate configuration example should satisfy two criteria simultaneously: 1) the natural language descriptions of the example's intent and the user's intent must be similar; and 2) they share topological similarities. We design a data structure called *intent graph* to take both factors into account. In each intent graph, the nodes represent devices and edges represent links between the devices, thus they correspond directly to network topologies. We embed the high-level intent into the intent graph by adding an attribute called "*Role*" to each node. We use an LLM (GPT-4o) prompted by few-shot examples to process the intent and generate roles for each node (details in §3.1). In this way, we embed the high-level intent into the network topology. Figure 3 illustrates a configuration example scenario and a target scenario, along with the corresponding example intent graph and user intent graph.

With the intent graphs, we transform the problem of finding the most appropriate example to the problem of finding the most similar example intent graph to the user intent graph, which is suitable for GNN algorithms to solve. To accelerate the search, we also propose a two-stage recommendation strategy, and the first stage is to narrow the search space by filtering out irrelevant examples with the help of an LLM (GPT-4o, details in §3.1). The Querier then adopts a GNN algorithm that solves the above graph similarity problem to recommend the most relevant example.

C2. Associating the devices in the target topology with those in the example. To successfully generalize, we must obtain the correspondence between the nodes in the example intent graph and those in the user intent graph. With this correspondence, we can then assign the configuration snippets from the example to the devices in the target topology.

D2. Exact matching of role description of nodes. We observe that, in most cases, nodes can be matched using only the "Role" attribute in the intent graph. Therefore, the Classifier first associates nodes in the user intent graph to nodes in the example intent graph, if their "Role" attribute matches exactly.

D3. GNN-based mapping based on neighborhood similarity. However, in some cases, a node in the user intent graph can be matched with multiple nodes in the example intent graph. To solve this problem, we observe that the configuration of a device is also highly affected by its neighbors. Taking Figure 3 as an example, router $R2$ and $R3$ in the example topology have the same role: acting as a relay, but they have different neighborhood: router $R2$ links a source and a relay, while router $R3$ links a source, a destination, and a router which does not participate in annoucement forwarding. Thus, router $R2$ and $R3$ have different configurations of routing policies affected by their neighborhood topologies. From Figure 3, we observe that router R_B in the target topology has the same role as router $R2$ and $R3$ in the example topology, but router R_B has a neighborhood more similar to that of router R_2 than that of router R_3, so the optimal association of R_B' in the example topology should be R_2.

Therefore, we propose a GNN-based node classification algorithm in the Classifier, because GNNs can learn node-level representations by aggregating the neighborhood information and are well-suited to this problem. If a device is matched to multiple nodes in the example intent graph, the GNN algorithm measures the neighborhood similarity between these nodes and recommends the node in the example intent graph

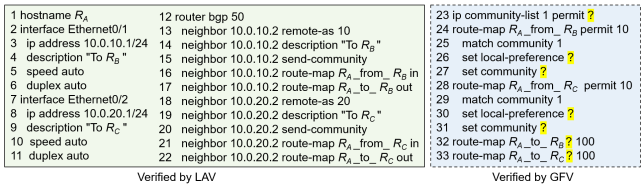


Figure 4: Configuration template for router R_A in Figure 3.

with the highest similarity.

C3. Accelerating synthesis without compromising correctness. Although recent research has demonstrated that LLMs are capable of writing network configurations [36], it is still challenging for LLM to produce concrete network-wide configurations, because it involves logical reasoning, which is not a strong suit for LLMs [36]. As a result, COSYNTH requires many human-in-the-loop iterations to generate the correct configurations. Thus, we seek a new synthesis mechanism to accelerate synthesis and reduce cost (*i.e.*the number of calling LLMs). In the meantime, it is imperative to guarantee the correctness of synthesis results.

D4. Guiding the LLM generation with appropriate examples. LLM can better complete reasoning tasks by introducing a few-shot prompt [12] or combining retrieval-augmented generation (RAG) [21]. Thus, to efficiently guide the LLM in generating the correct configuration, we provide appropriate examples to the LLM as relevant context.

D5. Restricting the LLM to generate only templates. In our configuration generation tasks, determining specific values of network policy parameters for all devices in a network is a highly complex reasoning problem that need to take the entire network into account. To solve this problem, we restrict the LLM to generate only configuration templates, leaving the specific values of network policy parameters as symbols.

D6. Using the Syntax Verifier, LAV, GFV and Formal Synthesizer to guarantee correctness. To guarantee the configuration correctness, we first use the Syntax Verifier to check whether the templates contain invalid syntax. We then utilize the LAV and GFV to jointly verify the semantic correctness of the templates. The LAV performs local verification on individual device templates to check whether the template follows the target topology and user intent descriptions. The GFV conducts global verification across all device templates to verify whether they can collectively achieve the global network policy intent. Any errors checked by the Syntax Verifier, LAV and GFV are fed to LLM for error correction. We finally use the Formal Synthesizer to take the target topology, user intents, and templates for all devices to fill the network policy parameters with concrete values to achieve the user intent.

Take the template (Figure 4) of router R_A in Figure 3 as an example, this template leaves some parameters of BGP import or export policies symbolic, such as *community* value and *preference* value, denoted by a question mark (?). We use the LAV to locally verify the interface configurations and BGP basic configurations from lines 2-22 based on the topology and intent descriptions, and then use GFV to globally verify

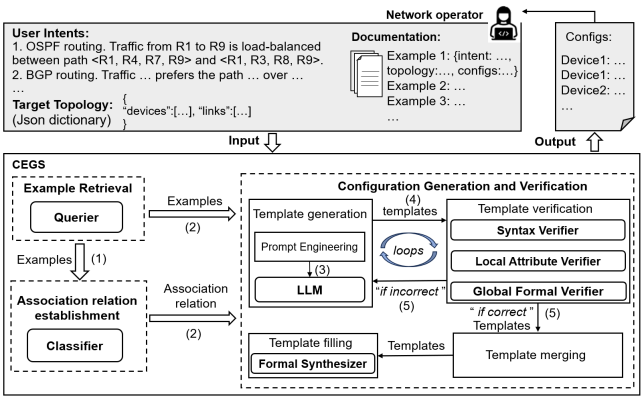


Figure 5: CEGS overview.

the policy configuration templates (lines 23-33) across all devices. Finally, the Formal Synthesizer fills the templates.

3 CEGS DESIGN

Given device documentation and a target network scenario including a target topology T_{tgt} specified in a JSON dictionary and a set of high-level intents $F=\{f^1_{user}, f^2_{user}, \ldots, f^n_{user}\}$ in natural language, CEGS automatically synthesizes network configurations via three phases: retrieval phase, association phase, and generation phase, as shown in Figure 5.

In the retrieval phase, CEGS uses the Querier to retrial relevant configuration examples from device documentation. In the association phase, CEGS utilizes the Classifier to establish accurate associations between devices in the target and example topologies. In the generation phase, given the user intents, target topology, as well as configuration examples and association relations, CEGS aims to generate correct network configuration for all devices in target topology to achieve all user intents. Specifically, we propose an iterative LLM-driven synthesis method to fulfill this task. In each iteration, we first generate configuration templates using an LLM, and validate these templates using the Syntax Veifier, LAV, and GFV. If any errors are checked, they are added to the prompt of the LLM for the next iteration. This phase stops when a set of templates are successfully generated, or when a predefined upper limit (300 in the evaluation) of loops is reached. Finally, we merge templates and fill in them using Formal synthesizer.

3.1 Querier

Given a set of high-level intents F and target topology T_{tgt}, the Querier aims to recommend the most relevant configuration example z_i for each user intent f^i_{user} from the device documentation. To achieve this, we first parse configuration examples from the device documentations. Since the format of device documentations provided by different operators varies, we develop a parsing framework similar to NAssim [11] to convert them into a uniform format. The formats are a uniform container for standardizing the diversity of manual formats. To parse the topology represented in a figure, we utilize GPT-4o to convert the topology figure into the predefined JSON format, followed by manual proofreading of the topology in-

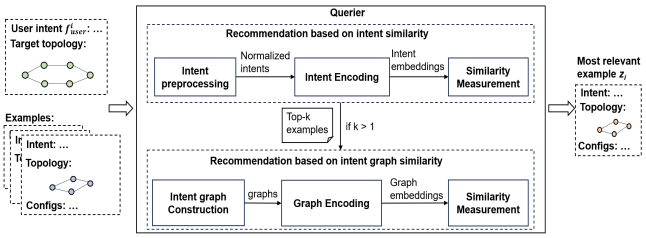


Figure 6: Detailed workflow of Querier.

formation.[3] We use a JSON dictionary to store the parsed configuration examples. Each example consists of a high-level intent in natural language, a network topology specified in JSON dictionary and the configurations of all devices in that topology. We process all the examples using the Querier to obtain the corresponding example intent graphs.

To improve query efficiency and accuracy, we adopt a two-stage recommendation strategy. Figure 6 shows the detailed workflow. For each user intent f^i_{user}, the Querier first adopts text encoding and similarity measurement methods to recommend top-k configuration examples that have the same type of intent as the user intent f^i_{user}, thus filtering out irrelevant examples. The Querier then adopts a GNN algorithm to recommend top-1 example from the top-k examples based on intent graph similarity. The output of the Querier is a set of configuration examples $Z=\{z_1,z_2,..,z_n\}$, where each z_i is the example for each user intent f^i_{user}.

3.1.1 *Recommendation based on intent similarity*

Problem Formulation. The intent is specified in natural language, so recommending configuration examples with the same type of intent can be regarded as a semantic alignment problem. Given a user intent f^i_{user} and example intent f^j_{exp}, this problem is solved by evaluating whether the semantics of intent f^i_{user} and intent f^j_{exp} are consistent.

Intent preprocessing. Due to the flexibility of natural language expression, the same intent can be expressed in various ways. For example, a path-preference intent may be described as "*For traffic from R1 to R4, the route (R1, R3, R4) is preferred over route (R1, R2, R4).*" or "*Traffic from R1 to R4 prioritizing taking (R1, R3, R4) over (R1, R2, R4).*". In addition, there may be some proper nouns in the intent description, such as specific device names and IP address, which do not impact the semantic meaning of the intent. To improve the accuracy of example recommendation, we propose an intent preprocessing approach that normalizes original intents into specific unified expression formats without specific proper nouns. Specifically, we first define a unified expression format for each intent type. We then leverage GPT-4o to convert the original intent into a unified expression format using a few-shot prompt (described in detail in Appendix A.1). For example, for the example intent and user intent described in Figure 3, we use GPT-4o to both convert them into the unified expression format "BGP routing. Traffic from *source* to *destination* prefers *path1* over *path2*". In this unified format, generic terms "*source*" and "*destination*" replace specific device names, and "*path1*" and "*path2*" replace detailed path information. We denote a normalized intent with respect to an original intent f as $norm(f)$.

Intent Encoding. To evaluate the semantic similarity between two intents, Sentence-BERT (SBERT) [40] is the most popular NLP model suitable for this task, as it can efficiently map each sentence to a vector space where semantically similar sentences are close to each other. For each pair of normalized intents of user intent $norm(f^i_{user})$ and example intent $norm(f^j_{exp})$, denoted as $(norm(f^i_{user}),norm(f^j_{exp}))$, we first use the encoder $e(\cdot)$ of SBERT to encode them into a pair of embedding vectors $(e(norm(f^i_{user})),e(norm(f^j_{exp})))$. We then use cosine similarity to measure the similarity between these two embedding vectors, which is calculated by

$$Sim(e(norm(f^i_{user})),e(norm(f^j_{exp})))= \quad (1)$$

$$\frac{e(norm(f^i_{user}))\cdot e(norm(f^j_{exp}))}{\|e(norm(f^i_{user}))\|\times\|e(norm(f^j_{exp}))\|} \quad (2)$$

The cosine similarity calculates the cosine of the angle between two vectors, which is often used to measure the semantic similarity, typically involving text data. If this similarity exceeds a threshold τ (0.9 in the evaluation), we consider intent f^i_{user} and f^j_{exp} to be semantically aligned, i.e., they belong to the same type of intent. Therefore, the configuration example corresponding to intent f^j_{exp} is regarded as a potential reference example c^i_j. We define a set $C^i=\{c^i_1,c^i_2,...,c^i_k\}$ to denote these potential reference examples, where k is the number of these examples. If k=1, we directly consider this example as the most relevant example for the user intent f^i_{user}.

3.1.2 *Recommendation based on intent graph similarity*

The Querier recommends the most relevant example from the top-k examples based on the intent graph similarity.

Problem Formulation. The most relevant example should have the similar intent to the user intent and simultaneously share topological similarity. Thus, we construct an intent graph for a pair of the intent and topology to consider both the two factors at the same. In this way, we transform the problem of recommending the most relevant example into a graph similarity problem.

Intent Graph Construction. Given an intent and a network topology, we first formulate the topology as an undirected graph, where each node v and each edge denote a device and a physical link in this topology, respectively. We then embed the intent into the graph by adding a specific attribute called "Role" to each node based on the intent. We refer to the topology graph embedding the intent as the intent graph g. We use GPT-4o to achieve this more effectively, due to their superior natural language processing capability. We first define a role set for each type intent. The role set should cover all possible roles that a device may play in fulfilling that type of intent. For example, for the path preference intent, as shown

[3] Note: Manual proofreading is required only once for accurate parsing. The parsed examples can be applied to numerous relevant target scenarios.

in Figure 3, we define the role set as *{route preference, relay, non-involvement, destination}*, where *route preference* refers to a device that prefers a neighbor for traffic transmission, such as $R1$ and R_A in Figure 3. *non-involvement* denotes a device that does not participate in traffic transmission, such as $R6$ in Figure 3. We then utilize GPT-4o to extract "Role" r_v for each node v from the intent using few-shot prompt. We describe attribute extraction with GPT-4o in detail in Appendix A.2. Under complex network scenarios, we can use a feature combination to define a role.

Graph Encoding. Give an intent graph g, we first leverage FastText [8], a NLP model, to encode the attribute values of nodes into vectors. FastText is efficient to learn word embedding through learning n-grams at the character level. We construct a corpus including the attribute values of all intent types, and adopt unsupervised learning to train a FastText model on this corpus. For each node $v \in V$, we utilize the trained FastText model to encode its textual attribute r_v into a vector $x_v = e(r_v)$, where $e(\cdot)$ is the encoder.

We then adopt GraphSAGE [25], the most popular GNN algorithm, to calculate the embedding of each node v in the graph g. Before describing how a GraphSAGE model is trained in this work, we show how GraphSAGE is used to perform node embedding. GraphSAGE uses L aggregators (denoted $AGGREGATE_l, \forall l \in \{1,2,...,L\}$) and a set of weight matrices $W^l, \forall l \in \{1,...,L\}$ to generate node embedding through l loops. At each loop l, node v's embedding is calculated by

$$h_v^l \leftarrow \sigma(W^l \cdot CONCAT(h_v^{l-1}, h_{\mathcal{N}(v)}^l)) \quad (3)$$

where, σ is a nonlinear activation function, CONCAT is a concatenation operator. $\mathcal{N}(v)$ is the set of neighbor nodes of the node v. The initial value of node embedding is its feature vector, i.e., $h_v^0 = x_v$. $h_{\mathcal{N}(v)}^l$ is the neighborhood representation of node v at the l-th loop, which is calculated by

$$h_{\mathcal{N}(v)}^l \leftarrow AGGREGATE_l(\{h_u^{l-1}, \forall u \in N(v)\}) \quad (4)$$

where h_u^{l-1} is neighbor node u's embedding at the $l-1$-th loop.

We use a global pooling method to aggregate the embeddings of all nodes in a graph g into a single vector as the representation of the graph through a MeanPool operator. The embedding of a graph g is represented as:

$$E_g = MeanPool(\{h_v, \forall v \in V\}) \quad (5)$$

where h_v denotes the embedding of node v, which is this node embedding obtained from the last loop L.

For each pair of user intent graph g_{user}^i and example intent graph g_{exp}^i, we use euclidean distance to measure the similarity between their graph embedding vectors $E_{g_{user}^i}$ and $E_{g_{exp}^i}$. A similarity score is calculated by

$$Score(E_{g_{user}^i}, E_{g_{exp}^i}) = \frac{1}{d(E_{g_{user}^i} - E_{g_{exp}^i})} \quad (6)$$

where $d(\cdot)$ is the L1 distance, which is often used in graph or node evaluation to capture the absolute differences between corresponding features. For a user intent graph g_{user}^i associated with the user intent f_{user}^i and target topology T_{tgt}, we use the above graph similarity measurement to find the top-1 similar example intent graph, and consider the configuration example associated with the top-1 example intent graph as the most relevant example z_i for the user intent f_{user}^i.

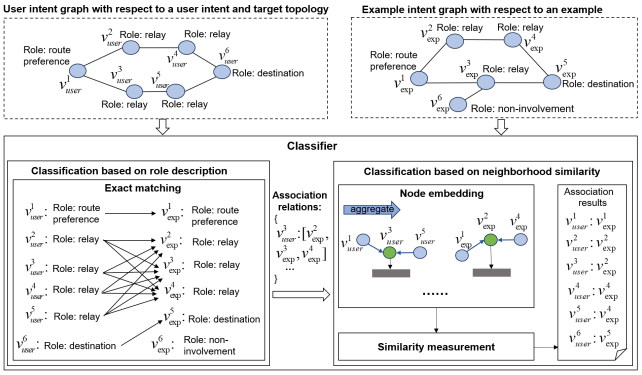


Figure 7: An instance of establishing the association between device in target and example topologies.

3.2 Classifier

For each user intent f_{user}^i and its corresponding configuration example z_i, the Classifier aims to establish the association between devices in target topology T_{tgt} and those in topology T_{exp}^i of the example z_i. The Classifier adopts a two-stage classification strategy to establish accurate association relationships on intent graphs constructed in §3.1.2. The Classifier first establishes the association relation based on the device's roles. If a device in target topology T_{tgt} is associated with multiple devices in example topology T_{exp}^i that have the same roles, the Classifier then adopts GraphSAGE to map this device to a specific one among these multiple devices in example topology T_{exp}^i based on neighborhood similarity. Figure 7 shows an instance of the Classifier establishing the association between devices in the target and example topologies.

Problem Formulation. We formalize this problem as a node classification problem on an intent graph pair of g_{user}^i and g_{exp}^i, where g_{user}^i is the user intent graph with respect to user intent f_{user}^i and target topology T_{tgt}, as well as g_{exp}^i is the example intent graph corresponding to the example z_i of the user intent f_{user}^i. In an intent graph g_{user}^i (or g_{exp}^i), each node v_{user} (or v_{exp}) is characterized by its role $r_{v_{user}}$ (or $r_{v_{exp}}$).

3.2.1 *Classification based on role description*

Given a pair of user intent graph g_{exp}^i and example intent graph g_{user}^i, the Classifier maps the nodes in the example intent graph g_{user}^i to these nodes in the intent graph g_{user}^i based on their roles. For each node v_{exp} in the graph g_{exp}^i, the Classifier sequentially evaluates whether the role $r_{v_{user}}$ of each node v_{user} in the user graph g_{user}^i exactly match those $r_{v_{exp}}$ of node v_{exp}, and if so, establishes an association between node v_{exp} and node v_{user}. In other words, node v_{user} is associated with v_{exp}, if and only if $r_{v_{user}} = r_{v_{exp}}$.

3.2.2 *Classification based on neighborhood similarity*

If a node v_{user} in user intent graph g_{user}^i is associated with multiple nodes in example intent graph g_{exp}^i, the Classifier further

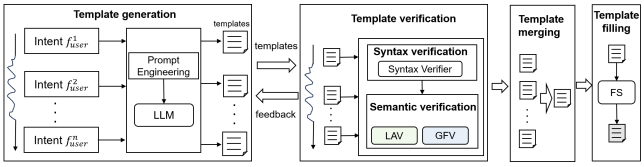


Figure 8: The workflow of LLM-driven synthesis method.

maps this node to a specific one among these multiple nodes. The Classifier first uses FastText to encode the node's textual attribute $r_{v_{user}}$ into a vector $x_{v_{user}}$, as described in §3.1. The Classifier then uses GraphSAGE to calculate the embedding $h_{v_{user}}$ of node v_{user} by aggregating its neighborhood embeddings in the graph g^i_{user}, as described in Equation 3 and 4. Simultaneously, for each node v_{exp} in graph g^i_{exp} that has the same roles with v_{user}, the Classifier uses FastText and GraphSAGE to calculate its embedding $h_{v_{exp}}$. Finally, the Classifier uses the Euclidean distance to measure the similarity between two node embedding vectors. For each node embedding pair of $h_{v_{user}}$ and $h_{v_{exp}}$, a similarity score is calculated by

$$Score(h_{v_{user}}, h_{v_{exp}}) = \frac{1}{d(h_{v_{user}} - h_{v_{exp}})} \tag{7}$$

where, $d(\cdot)$ is the L1 paradigm. The Classifier uses the above similarity measurement to map the node v_{user} to the specific node v_{exp} that has the highest similarity score.

3.2.3 *Training GraphSAGE model*

Given some intent graph pairs and node association relations in these graphs, we can generate a training corpus for training a GraphSAGE model. We construct positive pairs from nodes with associated relationships and negative pairs from nodes that are not associated. We train GraphSAGE model by minimizing the following margin-based ranking loss functions:

$$L = \sum_{(u,v)\in S} \sum_{(u,v')\in S'} [d(h_u, h_v) + \gamma - d(h_u, h_{v'})]_+ \tag{8}$$

where S is the set of positive pairs, S' denotes the set of negative pairs of the node u. h_u, h_v and $h_{v'}$ are the embedding of node u, v and v', which are calculated by GraphSAGE using Equation 3 and 4. γ (set to 3 in the evaluation) is a margin hyperparameter that separates positive and negative pairs.

3.3 An efficient LLM-driven synthesis method

To accelerate synthesis without compromising correctness, we propose an efficient LLM-driven synthesis method. Figure 8 shows the detailed workflow, including four stages. In the template generation, CEGS aims to generate templates with respect to each user intent. To guarantee the configuration correctness, CEGS first verifies template correctness. Once LLM correctly generates templates for all intents, CEGS merges these templates and then employs the Formal synthesizer to fill in the templates so that all intents are achieved.

3.3.1 *Template generation*

In this stage, CEGS leverages an LLM to generate templates for all devices in target topology T_{tgt} with respect to each user intent f^i_{user}. Due to the limited context window of the LLM, it cannot generate templates for all devices under large-scale

```
1 interface Ethernet1/0/0                    9 router ospf 50
2  ip address 47.86.50.1 255.255.255.0       10  router-id 10.10.1.1
3  description "To R2"                       11  network 47.86.50.0 0.0.0.255 area 0
4  ip ospf cost ?                            12  network 99.24.63.0 0.0.0.255 area 0
5 interface Ethernet1/0/1                    13  network 127.2.10.0  0.0.0.255 area 0
6  ip address 99.24.63.1 255.255.255.0
7  description "To R3"
8  ip ospf cost ?
```

Verified by LAV

Figure 9: Configuration template for router R1 with respect to an OSPF routing intent described in Figure 5.

typologies at once. To address this problem, we divide the devices in the topology into multiple subsets and generate the templates of devices in each subset in batches. We verify the templates for all devices collectively, so generating configurations for each batch separately do not compromise the correctness of the final configuration [4]. To speed up synthesis, we parallelize the template generation for devices in each subset, rather than processing them sequentially.

For each user intent f^i_{user}, CEGS first uses LangChain [10] to construct a prompt for each batch. We show an instance of the prompt construction in Figure 14 in Appendix A.4. We then call the LLM multiple times in parallel, feeding each batch's prompt to generate templates for all devices simultaneously. Each template is the configuration where some of network policy parameters (e.g., OSPF cost, route-map import or export parameters) are left symbolic, as illustrated in Figure 4 and Figure 9.

3.3.2 *Template verification*

In this stage, CEGS verifies configuration templates with respect to each user intent. For the templates corresponding to the intent f^i_{user}, CEGS first uses the Syntax Verifier to check the template syntax correctness, and then utilizes the LAV and GFV to jointly verify their semantic correctness.

Syntax verification. To verify template syntax correctness, we develop a Syntax Verifier by extending the Batfish [19] parser to treat symbols defined in the templates as valid syntax. If Syntax Verifier checks any invalid syntax, it reports relevant lines with invalid syntax to LLM for error correction.

Semantic verification. To speed up synthesis, it is crucial to provide localization feedback for LLMs to rapidly correct errors. To achieve this, we develop the LAV and GFV to jointly verify template semantic correctness by performing local verification and global verification. If LAV or GFV check any errors, it provides localized feedback in natural language.

- **LAV** The LLM may produce incorrect templates that do not follow the target topology and user intent descriptions, such as misconfiguring interfaces or incorrectly declaring or missing networks or neighbors. For this, we develop the LAV (that we wrote in Python) to perform local verification on each device template, checking whether it complies with the specified descriptions. Specifically, for each device template, the LAV first parses device attributes such as interface attributes (e.g., interface names, IP addresses), protocol attributes (e.g., AS number, pro-

[4] In our experiments, the size of subset is 40.

cess id) from the topology and intent descriptions. It then checks if the template correctly configures each attribute following these descriptions. If the LAV checks any errors, it provides localized feedback which indicates which configuration lines on which devices may be incorrect.

- **GFV** The LLM may produce templates that do not follow the intended global network policy. For this, we develop the GFV based on the NetComplete to perform global verification on the templates across all devices. The GFV ensures that configurations with interdependencies and cross-device impacts are correct. It reduces this task to a constraint satisfaction problem that it solves with SMT solvers [15]. The GFV uses SMT to encode the templates and uses variables to represent the symbolic in templates. To encode the user intents, the GFV first employs GPT-4o to convert the user intent from natural language to formal specification defined by the NetComplete using a few-shot prompt. We provide an instance how to use GPT-4o to convert natural language intents into formal specifications in Appendix A.3 and discuss the accuracy in section.4.2. The GFV then encodes the formal specification as SMT constraints. The GFV attempts to find a set of values to fill in the policy templates so that the formal specifications are achieved. If the GFV cannot find an appropriate set of values, it provides localized feedback indicating which devices have incorrect network policy configuration templates. Even when the GFV finds feasible values, we do not immediately fill in the templates. Instead, we retain the templates and fill in them collectively during the template filling phase based on all the intents together. [5]

Taking the configuration verification for an BGP routing intent as an example, we illustrate how our LAV and GFV work. Figure 4 shows the configuration template of a device with respect to a BGP routing intent (the user intent shown in Figure 3). We use the LAV to locally verify each device's interface configuration (lines 2-11 in Figure 4) and BGP basic configuration (lines 12-22 in Figure 4) including AS number and neighbor settings. The LAV first extracts the interface description, AS number, and IP addresses of all neighbors from the target topology description. It then checks whether the interface configuration and BGP basic configuration follow these descriptions. Since the routing policy configurations of individual devices affect each other and work together to achieve the intended global routing policy, we use the GFV to globally perform a formal verification on policy configuration templates (lines 23-33 in Figure 4) for all devices.

In some cases, we can complete the template verification using only the LAV, such as for an OSPF routing intent. Figure 9 illustrates a configuration template for a device with respect to an OSPF routing intent. Under the OSPF protocol, achieving a specific global routing policy involves setting an appropriate OSPF cost for each link. The templates represents the OSPF costs as symbols, which are comprehensively filled using the Formal Synthesizer in template filling phase. Thus, we can use the LAV to locally verify each device's template by checking whether the interface configuration (lines 1-8) correctly sets interface attributes following the target topology description and contains the OSPF cost settings (line 4 and line 8). At the same, we use the LAV to locally check whether OSPF basic settings and network statement settings (lines 9-13) comply with the topology description.

[5] While it is theoretically possible for both the formal specification and the template to be incorrectly generated yet still satisfy the SMT constraints, such cases are rare in practice. We plan to address this in future work.

3.3.3 Template merging

Once the LLM correctly generates configuration templates with respect to all intents. We merge the templates of each device with respect to all intents into a whole. In general, the configuration consists of multiple independent configuration segments, each of which implements a specific function. Thus, we merge the templates based on configuration segments. For each device, we merges the configuration segments associated with each specific function across all intents, and then merges the segments corresponding to different functions.

3.3.4 Template filling

Since NetComplete [17] can support configuration synthesis under large-scale realistic networks and cover more type of intents, we select NetComplete as our Formal Synthesizer (FS). We first employ GPT-4o to transform each user intent in natural language into the formal specification defined by NetComplete using a few-shot prompt. We then feed the merged templates for all devices, along with the formal specifications associated with all intents and the target topology to NetComplete to generate concrete values for policy parameters, thereby filling in the templates. The configurations generated by the Formal Synthesizer are correct and can satisfy the specified intents, because the Formal Synthesizer uses SMT-based formal methods to fill templates according to intent constraints, which has been validated in [17].

4 EVALUATION

We implemented CEGS in around 8K lines of Python code. We evaluate CEGS from three aspects: 1) How does CEGS compare to the state-of-the-art LLM-based configuration synthesis system COSYNTH [36] and NETBUDDY [49] [51]? 2) How does CEGS perform in achieving various set of intents across different network scales? 3) How do the core components of CEGS perform?

4.1 Experiment setup

We run all our experiments on a server with 80 core 2.1 GHz processors and 256GB RAM. We evaluate CEGS, COSYNTH, and NETBUDDY in generating Cisco router configurations that comply with the specific routing intents.

Topologies. We sampled 20 real-world ISP PoP-level network topologies from Topology Zoo dataset [29], with scales ranging from 20 to 754 routers. These topologies include 12 commercial networks and 8 educational networks, covering

various areas such as metro, region, country, and continent. Moreover, we constructed a larger topology with 1094 routers by merging three of sampled topologies from the region area.

Intents. We consider six types of prevailing routing intents, covering Static, OSPF, and BGP protocols. For Static protocol, we examine static route intents. For OSPF protocol, we evaluate ECMP and Any-path intents. For BGP protocol, we investigate Simple, Ordered, and No-transit intents. Each intent is defined between a randomly selected source *src* and destination *dst* pair and feasible paths. For a static route or a BGP Simple intent, we randomly select one feasible path p from *src* to *dst*, specifying that traffic from *src* to *dst* must follow the path p. For ECMP, Any-path, or Ordered intents, we first randomly select two available paths p_1 and p_2 from *src* to *dst*. For an ECMP intent, traffic from *src* to *dst* is load-balanced among paths p_1 and p_2. For a Any-path intent, traffic from *src* to *dst* can be forwarded along either path p_1 or p_2. For an Ordered (i.e., path preference) intent, traffic from *src* to *dst* prefers path p_1 over path p_2. We store the source-destination pair and feasible paths for each intent to facilitate configuration validation using Batfish in Sec.4.2. We utilize GPT-4o to generate intent description using few-show examples provided by NetOps experts. We describe these few examples and prompt construction in detail in Appendix.A.5.

Configuration examples corpus. We parsed 300 configuration examples from Cisco documentation and a cloud service provider, covering 40 types of intents, such as BGP routing, OSPF routing, and more. The examples for each type intent cover various networks with different topological features.

LLM selection. In the following experiments, we utilize GPT-4o to generate configurations, owing to its superior capabilities in configuration generation, compared to other models, as shown in Table 9 in Appendix B.2.

GraphSAGE model. We ask NetOps experts to help construct 100 intent graphs. For each pair of matching graphs, we ask the experts to label the associations between corresponding nodes in the two graphs. We then treat the node and its associated counterpart in the other graph as positive node pairs, while pairing the node with other non-associated nodes as negative pairs. These labeled pairs are used to create a training corpus, which is then used to train a GraphSAGE model using the loss function described in equation 8. This model consists of two hidden layers, each with 128 neurons, and is trained with a learning rate of 0.001.

Metrics. We present the synthesis system performance using three metrics: loops, synthesis time, and SMT solving time. We measure intent diversity using a semantic richness metric, defined as the average similarity between pairs of intents of the same type using SBERT embeddings. In our evaluation, the semantic richness across intent types is 0.52.

4.2 Correctness evaluation

Intent Formalization Accuracy. CEGS uses formal methods to verify and fill in templates created by LLM, which depends on accurately converting natural language intents into formal specifications. CEGS handles this with GPT-4o using few-shot prompts. To evaluate GPT-4o's accuracy, we wrote a Python script that converts each intent from our evaluations into the expected formal specification. We then asked GPT-4o to translate these intents and compared its output with the expected results. Our results show that GPT-4o achieved 100% accuracy in converting 1,350 intents from our evaluations.

Table 1: Comparison results between CEGS and COSYNTH. U stands for unsuccessful generation. ✓ (or ×) indicates that the synthesized configurations satisfy (or do not satisfy) the intents as verified by Batfish.

# No-transit intents	3		21		55	
Synthesis system	CEGS	COSYNTH	CEGS	COSYNTH	CEGS	COSYNTH
Automatic loops	**2**	300(U)	**4**	300(U)	**5**	300(U)
Synthesis time	**24s**	1h(U)	**58s**	1h(U)	**1m32s**	1h(U)
(SMT time)	**(5s)**	(0)	**(13.2s)**	0	**(21s)**	(0)
Verified	✓	×	✓	×	✓	×
Manual loops	**0**	2	**0**	5	**0**	22

Table 2: Comparisons between CEGS and NETBUDDY.

# ECMP intents	2		6		10	
Synthesis system	CEGS	NETBUDDY	CEGS	NETBUDDY	CEGS	NETBUDDY
Automatic loops	**1**	300(U)	**3**	300(U)	**4**	300(U)
Synthesis time	**42s**	3h(U)	**2m2s**	3h(U)	**3m26s**	3h(U)
(SMT time)	**(8s)**	(0)	**(20s)**	(0)	**(32s)**	(0)
Verified	✓	×	✓	×	✓	×
Manual loops	**0**	3	**0**	8	**0**	12

Validation. We validate the configurations synthesized by a synthesis system using Batfish. We wrote a Python script to automate this verification process based on the information collected in Section 4.1 about the source-destination pair and feasible paths corresponding to the intent. Specifically, we utilize Batfish "traceroute" to analyze the packet transmission path from a source to a destination specified by the intent. For a Static route intent or Simple intent, we check whether the path matches the expected path. For an Any-path intent, we first check whether the path is either path $p1$ or $p2$ as specified by the intent. We then break a random link on the path $p1$ and a random link on $p2$, and use Batfish "traceroute" to verify whether there is no path for packet transmission. For an Ordered intent, we first check whether the path matches the preferred path specified by the intent. We then randomly break a link on the preferred path, and use Batfish "traceroute" to verify whether the transmission path is changed to the secondary path specified by the intent. For an ECMP intent, we use Batfish "multipathConsistency" to verify whether there are equal paths as specified by the intent for traffic transmission from its source to destination. For a No-transit intent, we first use Batfish's "traceroute" to verify that there is no transmission path between two routers that should not communicate, as specified in the intent. We then use Batfish "traceroute" to verify whether a feasible path exists between two routers that are expected to communicate.

4.3 Comparison to COSYNTH and NETBUDDY

Since COSYNTH and NETBUDDY limit their scopes to different configuration scenarios, we compare them individually. COSYNTH does not have publicly available code, so we reproduced it based on the provided method descriptions in its

paper. We have verified that our reproduction aligns with the experimental results reported in [36]. For NETBUDDY, we use its released source code [50] for evaluation.

COSYNTH focuses on star networks and BGP no-transit intents, so we compare it in this context. In a star network, a central router connects to a CUSTOMER IP, while other routers connect to various ISPs. A no-transit intent specifies that two routers from different ISPs cannot communicate with each other, but can communicate with the CUSTOMER. We evaluate CEGS and COSYNTH on 3 star topologies from a service provider, each with 4, 8, and 12 routers, and containing 3, 21, and 55 no-transit routing intents, respectively. Table 1 presents the performance of CEGS and COSYNTH. We observe that CEGS can synthesize correct configurations that satisfy different set of no-transit intents within 2 minutes without human involvement. However, COSYNTH fails to synthesize correct configuration within 300 loops and requires extra manual loops to correct errors, taking over 3 hours. Despite extra SMT time due to the Formal Synthesizer, CEGS's total synthesis time is much shorter than COSYNTH's. The main reason is that COSYNTH does not fully leverage the EFG capabilities of LLMs. COSYNTH asks LLM to complete configuration synthesis which involves complex logical reasoning based solely on user intents, without providing any examples or leveraging formal methods for logical reasoning. Thus, it requires human experts for correction. As a result, COSYNTH is more than 30 times slower than CEGS, even with expert involvement.

We compare CEGS with NETBUDDY in achieving ECMP (i.e., load-balancing) intents under a topology with 36 routers. To simplify configuration synthesis, NETBUDDY divides configuration synthesis into three subtasks: translating natural language intents into formal specifications, generating high-level configurations (e.g., routing information), and generating network low-level configurations. However, NETBUDDY still fails to produce correct configurations within 3 hours and needs manual intervention, as shown in Table 2. The main reason is that it lacks appropriate configuration examples for LLM reasoning. In contrast, CEGS utilizes the Querier to provide relevant configuration examples, along with formal methods for logical reasoning. CEGS can quickly synthesize correct configurations within 4 minutes without human involvement, over 45 times faster than NETBUDDY.

4.4 Performance under different set of intents and different network scales

In this subsection, we further evaluate the performance of our CEGS for achieving various set of intents under different network scales. First, we evaluate CEGS performance in synthesizing configurations for varying numbers of intents across different network sizes. Table 3 and Table 4 show the results of our CEGS for synthesizing configurations for OSPF intents and BGP intents, respectively. We observe that the synthesis time and SMT time are proportional to the number of intents and the topology size. This is because more times of LLM inference are required for generating configuration templates and more symbolic policy parameters need be calculated by the Formal Synthesizer. The configurations synthesized by CEGS satisfy the corresponding intents, as verified by Batfish. On average, configurations for each device in OSPF intent scenarios consist of about 20 lines, and those in BGP intent scenarios consist of about 35 lines.

Table 3: CEGS performance in synthesizing configurations for various numbers of OSPF intents.

Network size	Intent type	#Intents	Loops	Synthesis time (SMT time)	Verified
20~50	ECMP	2	2	42.4s (4.2s)	✓
		4	2	1m18s (8.1s)	✓
		6	3	2m10s (18.9s)	✓
	Any-path	2	2	43.5 (4.5s)	✓
		4	2	1m20s (9.2s)	✓
		6	3	2m15s (20.5s)	✓
150~200	ECMP	2	4	1m30s (8.2s)	✓
		4	5	2m5s (17.3s)	✓
		6	5	3m10s (21.4s)	✓
	Any-path	2	4	1m40s (15s)	✓
		4	5	2m15s (24s)	✓
		6	5	3m20s (29s)	✓
1094	ECMP	2	9	5m35s (32s)	✓
		4	9	6m30s (51s)	✓
		6	10	7m50s (1m3s)	✓
	Any-path	2	9	6m44s (2m10s)	✓
		4	10	10m33s (4m23s)	✓
		6	10	14m30s (7m35s)	✓

Table 4: CEGS performance in synthesizing configurations for various numbers of BGP intents.

Network size	Intent type	#Intents	Loops	Synthesis time (SMT time)	Verified
20~50	Simple	2	3	1m20s (10s)	✓
		4	3	2m10s (12.5s)	✓
		6	4	3m5s (19s)	✓
	Ordered	2	4	1m40s (13s)	✓
		4	6	2m28s (16s)	✓
		6	6	3m25s (24s)	✓
150~200	Simple	2	6	3m30s (45s)	✓
		4	7	4m35s (58s)	✓
		6	8	6m8s (1m2s)	✓
	Ordered	2	8	4m15s (40s)	✓
		4	11	5m40s (1m5s)	✓
		6	13	6m55s (1m15s)	✓
1094	Simple	2	18	11m35s (2m50s)	✓
		4	19	13m15s (3m50s)	✓
		6	22	14m40s (4m40s)	✓
	Ordered	2	30	18m40s (8m45s)	✓
		4	34	20m35s (9m50s)	✓
		6	37	24m50s (10m30s)	✓

We then evaluate the performance of CEGS under more complex configuration scenarios, where multiple types of intents must be achieved simultaneously. We construct three scenarios, each involving different combinations of intents across the same or different protocols within a network of 197 routers. In Scenario Scen1, 5 BGP No-Transit and 5 Ordered intents need to be satisfied. Scenarios Scen2 and Scen3 involve different types of intents across different protocols. As shown in Table 5, CEGS can synthesizes configurations that satisfy multiple types of intents simultaneously. Even in the third scenario, including 15 intents across Static, OSPF, and BGP protocols, CEGS can synthesize correct configurations for 197 routers within 11 minutes. Configurations of each

Table 5: CEGS performance in synthesizing configurations for various combinations of multiple types of intents.

Scenarios	Protocols	Intent type	#intents	Loops	Synthesis time (SMT time)	Verified
Scen1	BGP	No-transit Ordered	5 5	13	7m30s (45s)	✓
Scen2	Static OSPF	static route ECMP	5 5	3	4m36s (30s)	✓
Scen3	Static OSPF BGP	static route ECMP Ordered	5 5 5	11	10m30s (1m42s)	✓

Table 6: Ablation study results on six core components.

Querier	Classifier	Syntax Verifier	LAV	GFV	Formal Synthesizer	Loops
×	×	✓	✓	✓	✓	300(U)
✓	×	✓	✓	✓	✓	35
✓	✓	×	✓	✓	✓	300(U)
✓	✓	✓	×	✓	✓	300(U)
✓	✓	✓	✓	×	✓	300(U)
✓	✓	✓	✓	✓	×	50
✓	✓	✓	✓	✓	✓	10

device in scenario Scen1, Scen2, and Scen3 contain about an average of 40 lines, 23 lines, and 52 lines, respectively.

4.5 Ablation Study

To evaluate the contribution of six core components in CEGS, we conduct ablation experiments on the sampled 20 realistic topologies, each with one BGP Ordered intent. Since the Classifier depends on the Querier, when evaluating the Querier, we both remove Querier and Classifier, and instruct GPT-4o to generate configuration templates without examples. For evaluating other components, we disable the specific component's functionality in CEGS. From Table 6, we observe that CEGS achieves the best performance when all components are employed simultaneously. Without the Querier and Classifier, CEGS fails to synthesize correct configurations within 300 loops, indicating the importance of configuration examples for automating EFG. Compared to not using the Classifier, CEGS can more rapidly synthesizes correct configurations by matching examples to appropriate devices using Classifier. Without a Formal Synthesizer, CEGS requires more loops to synthesize correct configurations, as GPT-4o struggles to correctly specify policy parameters. Without the Syntax Verifier, LAV or GFV, CEGS fails to synthesize correct configurations within 300 loops due to the lack of error feedback. We discuss the performance of Querier's recommendation and Classifier's classification strategies in detail in Appendix B. *We discuss how to generalize CEGS in Appendix D.*

5 RELATED WORK

Intent-based Networking. Intent-based networking [26, 30] is a networking paradigm that automates network optimization to achieve desired goals by defining high-level intents, which has been well-adapted in Software-Defined Networks (SDN) [38, 57, 61]. For example, LUMI [27] allows operators to express intents in natural language, converts them into formal intents and automatically generates configurations. LUMI takes operator feedback to verify the intents are correct.

Configuration Verification. Configuration verification tools [2, 5, 19, 45, 48, 54] aim to check whether the configurations comply with the network policies in control plane using formal methods. For example, Minesweeper [5], FSR [48] and Bagpipe [54] uses SMT solvers for verification.

Configuration Synthesis. Configuration synthesis tools [1, 6, 7, 16, 17, 22, 39, 43, 44, 46] aim at synthesizing network configurations out of high-level intents. These tools can be categorized into two types: DSL-based synthesizers and template-based synthesizers. DSL-based synthesizers require operators to express intents using the customized DSL, after which they compile these intents into correct configurations. Template-based synthesizers require network operators to provide suitable configuration templates, which then automatically fill in the templates. However, these synthesizers still require human involvement (using DSL to express intents or designing appropriate configuration templates) in the synthesis process, because they lack a core capability of EFG.

LLMs and GNNs for configuration synthesis. ConfigReco [24] is a GNN-based configuration recommendation tool, which recommends high likelihood configuration snippets to operators. Researchers have explored the ability of LLM to write network configuration. NETBUDDY [49] [51] uses a multi-stage pipeline to leverage LLMs for configuration synthesis. COSYNTH [36] adopts a verified prompt programming strategy that combines the LLM with verifiers to translate intents into router configurations. However, NETBUDDY and COSYNTH still require expert involvement in the synthesis process, as they do not fully utilize the potential EFG capability of LLM. PreConfig [31], a pretrained model, is designed to automate network configuration. Although it has the potential ability to generate configuration snippets, it cannot generate complete network-wide configurations.

6 CONCLUSIONS

We presented CEGS that leverages LLMs to automate EFG in configuration synthesis. CEGS takes a set of high-level intents in natural language, a network topology and the device documentation as input, and automatically synthesize correct network configurations. Our results show that CEGS can synthesize correct configurations for complex intents on various realistic networks. While this paper focuses on configuration synthesis, we believe that the automation of EFG can be applied to other configuration management tasks.
This work raises no ethical concerns.

7 Acknowledgments

We thank our shepherd Sanjay Rao and the anonymous NSDI reviewers for their constructive comments. Dan Li and Yukai Miao are the corresponding authors. This work was supported by National Key R&D Program of China (2022YFB3105000), the Beijing Outstanding Young Scientist Program (No. JWZQ20240101008), the National Natural Science Foundation of China under Grant U23B2001, and was supported by Zhongguancun Laboratory.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 482–495, 2020.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, 2020.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[6] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 437–451, 2017.

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Harrison Chase. Langchain.(2022). *https://github.com/langchain-ai/langchain*, 2022.

[11] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. Software-defined network assimilation: bridging the last mile towards centralized network configuration management with nassim. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 281–297, 2022.

[12] Daixuan Cheng, Shaohan Huang, Junyu Bi, Yuefeng Zhan, Jianfeng Liu, Yujing Wang, Hao Sun, Furu Wei, Denvy Deng, and Qi Zhang. Uprise: Universal prompt retrieval for improving zero-shot evaluation. *arXiv preprint arXiv:2303.08518*, 2023.

[13] Richard Chirgwin. Google routing blunder sent japan's internet dark on friday. *The Register*, 2017.

[14] G. Corfield. British airways' latest total inability to support upwardness of planes* caused by amadeus system outage, July 2018.

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[16] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, pages 261–281. Springer, 2017.

[17] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[18] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426, 2019.

[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, 2015.

[20] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

[21] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

[22] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 359–373, 2017.

[23] GitHub. Github copilot: Your ai pair programmer, 2023.

[24] Zhenbei Guo, Fuliang Li, Jiaxing Shen, Tangzheng Xie, Shan Jiang, and Xingwei Wang. Configreco: Network configuration recommendation with graph neural networks. *IEEE Network*, 2023.

[25] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[26] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 86–97, 2018.

[27] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, Walter Willinger, and Sanjay G Rao. Hey, lumi! using natural language for {intent-based} network management. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 625–639, 2021.

[28] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[29] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[30] Aris Leivadeas and Matthias Falkner. A survey on intent-based networking. *IEEE Communications Surveys & Tutorials*, 25(1):625–655, 2022.

[31] Fuliang Li, Haozhi Lang, Jiajie Zhang, Jiaxing Shen, and Xingwei Wang. Preconfig: A pretrained model for automating network configuration. *arXiv preprint arXiv:2403.09369*, 2024.

[32] Yuan Li, Yixuan Zhang, and Lichao Sun. Metaagents: Simulating interactions of human behaviors for llm-based task-oriented coordination via collaborative generative agents. *arXiv preprint arXiv:2310.06500*, 2023.

[33] Zhiming Li, Yushi Cao, Xiufeng Xu, Junzhe Jiang, Xu Liu, Yon Shin Teo, Shang-wei Lin, and Yang Liu. Llms for relational reasoning: How far are we? *arXiv preprint arXiv:2401.09042*, 2024.

[34] Dimitrios Michael Manias, Ali Chouman, and Abdallah Shami. Towards intent-based network management: Large language models for intent extraction in 5g core networks. *arXiv preprint arXiv:2403.02238*, 2024.

[35] Meta. Update about the 4 october outage, October 2021.

[36] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. What do llms need to synthesize correct router configurations? In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 189–195, 2023.

[37] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. Clarifygpt: Empowering llm-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996*, 2023.

[38] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[39] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. Practical intent-driven routing configuration synthesis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 629–644, 2023.

[40] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[41] Jaromir Savelka, Arav Agarwal, Christopher Bogart, and Majd Sakr. Large language models (gpt) struggle to answer multiple-choice questions about code. *arXiv preprint arXiv:2303.08033*, 2023.

[42] Zhenwei Shao, Zhou Yu, Meng Wang, and Jun Yu. Prompting large language models with answer heuristics for knowledge-based visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14974–14983, 2023.

[43] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of fault-tolerant distributed router configurations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–26, 2018.

[44] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439, 2016.

[45] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 94–107, 2023.

[46] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[48] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking*, 20(6):1814–1827, 2012.

[49] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. Netconfeval: Can llms facilitate network configuration? *Proceedings of the ACM on Networking*, 2(CoNEXT2):1–25, 2024.

[50] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. Netconfeval github repository. https://github.com/NetConfEval/NetConfEval, 2024. Accessed: 2024-04-23.

[51] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. Making network configuration human friendly. *arXiv preprint arXiv:2309.06342*, 2023.

[52] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting static resource leak detection via llm-based resource-oriented intention inference. *arXiv preprint arXiv:2311.04448*, 2023.

[53] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[54] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications*, pages 765–780, 2016.

[55] Le Wu, Peijie Sun, Richang Hong, Yanjie Fu, Xiting Wang, and Meng Wang. Socialgcn: An efficient graph convolutional network based model for social recommendation. *arXiv preprint arXiv:1811.02815*, 2018.

[56] Kun Xu, Liwei Wang, Mo Yu, Yansong Feng, Yan Song, Zhiguo Wang, and Dong Yu. Cross-lingual knowledge graph alignment via graph matching neural network. *arXiv preprint arXiv:1905.11605*, 2019.

[57] Ze Yang and Kwan L Yeung. Sdn candidate selection in hybrid ip/sdn networks for single link failure protection. *IEEE/ACM Transactions on Networking*, 28(1):312–321, 2020.

[58] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. Qa-gnn: Reasoning with language models and knowledge graphs for question answering. *arXiv preprint arXiv:2104.06378*, 2021.

[59] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.

[60] Yongqi Zhang and Quanming Yao. Knowledge graph reasoning with relational digraph. In *Proceedings of the ACM web conference 2022*, pages 912–924, 2022.

[61] Ziyao Zhang, Liang Ma, Kin K Leung, and Franck Le. More is not always better: An analytical study of controller synchronizations in distributed sdn. *IEEE/ACM Transactions on Networking*, 29(4):1580–1590, 2021.

[62] Qi Zhu, Hao Wei, Bunyamin Sisman, Da Zheng, Christos Faloutsos, Xin Luna Dong, and Jiawei Han. Collective multi-type entity alignment between knowledge graphs. In *Proceedings of The Web Conference 2020*, pages 2241–2252, 2020.

Given an intent, please convert it into one of the expression format defined as follows:
Format1: <protocol> routing. Traffic from source to destination prefers path1 over path2.
Format2: <protocol> routing. Traffic from source to destination must follow the path.
Format3: <protocol> routing. Traffic from source to destination is load-balance between path1 and path2.
…

For example:
==========
Intent: Each router runs the BGP protocol. Traffic from from R1 to R4 prioritizing taking (R1, R3, R4) over (R1, R2, R4).
Normalization: BGP routing. Traffic from source to destination prefers path1 over path2.

…
==========

Intent: All routers are configurated with BGP protocol. When traffic is sent from R_A to R_F, it prioritizes the route (R_A, R_B, R_D, R_F) over route (R_A, R_C, R_E, R_F).

Figure 10: Example prompt for normalizing intents.

Given a configuration intent and a network topology, please extract a role attribute for each node in the topology from the intent.

For example:
==========
Intent: Each router runs the BGP protocol . Traffic from R1 to R5 prefers the path (R1, R3, R5) over (R1, R2, R4, R5).
Topology: {"routers": [R1, R2, R3, R4, R5], "links": [{"node1": {"name": "R1"}, "node2": {"name": "R2"}},...]}
Role attributes: {"R1":route preference, "R2": relay, "R3": relay, "R4": relay, "R5": destination, "R6": non-involvement}

Intent: ...
Topology: ...
Role attributes: ...

Intent: ...
Topology: ...
Role attributes: ...
…
==========

Intent: ...
Topology: ...
Role attributes:

Figure 11: Example prompt for extracting attributes for each device in the topology from the intent.

Given a routing requirement type, the source and destination of traffic flows, as well as the possible transmission paths of traffic flows, generate an intent description to express this requirement.

Requirement definition:
Static route or Simple requirement: given a path *p*, this requirement specifies that traffic from a source to a destination must pass the path *p*.
ECMP requirement: given two path *p1* and *p2*, this requirement specifies that traffic from a source to a destination is load-balanced between path *p1* and *p2* .
….

Examples:
==========
The source and destination of traffic flows: R1, R9
Possible transmission paths: (R1, R3, R5, R7, R9), (R1, R4, R8, R9)
Requirement: OSPF ECMP

Intent description: Configure OSPF routing process on each device and implement an OSPF routing policy to load-balance traffic from R1 to R9 across the paths (R1, R3, R5, R7, R9) and (R1, R4, R8, R9).

…
==========

The source and destination of traffic flows: routerA, routerF
Possible transmission paths: (routerA, routerC, routerD, routerF), (routerA, routerB, routerE, routerF)
Requirement: OSPF ECMP

Intent description:

Figure 12: Example prompt for generating intent description.

A Prompt Construction

In this section, we provide detailed examples of prompt construction involved in configuration synthesis with our CEGS.

A.1 Intent preprocessing prompt

Give a user intent, we normalize it using unified expression formats. We utilize GPT-4o to convert it into this format using a few-shot prompt. The prompt includes three parts: the definition of unified expression formats, few examples, and the user intent. Figure 10 illustrates an example of such a prompt.

A.2 Role extraction prompt

Given a pair of an intent and a topology, we leverage GPT-4 to extract a role attribute for each device in the topology from the intent using prompts with few-shot examples to embed the intent to the topology graph. Figure 11 shows a detailed example prompt.

A.3 Intent conversion prompt

CEGS utilizes the formal tool to verify and fill in templates. For this, CEGS first uses GPT-4o to convert the user intent from natural language to formal specification defined by the formal tool using a few-shot prompt. A example prompt is shown in Figure 13.

A.4 Template generation prompt

Figure 14 shows an example of prompt for LLM to generate configuration templates. The prompt includes six parts: background description, configuration example, target scenario, association relation between devices in the target and example topologies, format definition of the configuration template, and instructions. The configuration example consists of a high-level intent, a topology specified in a JSON dictionary and the configurations of all nodes in the topology. The target network scenario includes the user intent f^i_{user} and target topology T_{tgt}. The instruct indicates which devices in the topology the LLM should generate templates for.

A.5 Intent construction prompt

In the evaluation in Sec.4, we generate intent description for each target scenario using GPT-4o. We first ask NetOps experts draft few examples for each type intent. We then utilize GPT-4o to generate intent description using a few-shot prompt, as shown in Figure 12.

B Detailed Experimental Results

In this section, we provide detailed experiment results of ablation study and comparison between different LLMs.

B.1 Ablation study

We conduct detailed ablation studies on the Querier's recommendation strategies and Classifier's classification strategies to further evaluate the performance of Querier and Classifier.

Ablation study of recommendation strategies. Table 7 shows the ablation study results on the recommendation strategies of the Querier. The results convey four important insights. *First*, without adopting intent preprocessing, the Querier achieves 85% accuracy in example recommendations. Owing to inaccurate examples, CEGS is unable to successfully synthesize correct configurations for all networks. This is due to the flexibility of natural language expression, which leads to diverse expressions of the same type intent, posing a challenge for accurate identification. In constract, by adopting an intent standardization preprocessing method, the Querier shows significant improvement in recommending configuration examples with 100% accuracy, enabling our CEGS to successfully synthesize correct configurations for all networks within an average of 28 loops. In addition,

Given that intents in natural language, your task is to transform the intents into the responding formal specifications. There are three types of formal specifications: PathReq(), ECMPPathsReq(), and PathOrderReq().

1. PathReq() is used to represent a specific transmitting path requirement, which is defined as: PathReq(<protocol>, <destination>, <path>, False). It indicates that traffic destined for <destination> transmits following the path <path>.

2. ECMPPathsReq() is used to represent a load-balance requirement, which is defined as: ECMPPathsReq(<protocol>, <destination>, [PathReq(<protocol>, <destination>, <path1>, False), PathReq(<protocol>, <destination>, <path2>, False)], False). It indicates that traffic destined for <destination> is load-balanced along the path <path1> and the path <path2>.

3. PathOrderReq() is used to represent a path-preference requirement, which is defined as: PathOrderReq(<protocol>, <destination>, [PathReq(<protocol>, <destination>, <path1>, False), PathReq(<protocol>, <destination>, <path1>, False)], False). It indicates that traffic destined for <destination> prefers the path <path1> over the path <path2>.

In these formal specifications, <protocol> represents the routing protocol used by the router, with a value range of [Protocols.BGP, Protocols.OSPF, Protocols.Static].

==========

For example:

Intent: BGP routing. The traffic from RouterA to RouterD prefers the path of <RouterA, RouterB, RouterD> over the path of <RouterA, RouterC, RouterD>.

Specification: [PathOrderReq(Protocols.BGP, 'RouterD', [PathReq(Protocols.BGP, 'RouterA', [RouterA, RouterB, RouterD], False), PathReq(Protocols.BGP, 'RouterD', [RouterA, RouterC, RouterD], False)], False)

…

==========

Intent: All routers are configurated with BGP protocol. When traffic is sent from R_A to R_F, it prioritizes the route (R_A, R_B, R_D, R_F) over route (R_A, R_C, R_E, R_F).

Specification:

Figure 13: Example prompt for converting intents into formal specifications.

compared with not using the second-stage recommendation, the Querier uses graph similarity measurement method to recommend the most relevant examples, allowing CEGS to synthesize the correct configuration faster. This is because the graph similarity method can find examples that share similar topological features to the target topology, thus guiding the Classifier better match the example to devices.

Ablation study on classification strategies. Table 8 shows the ablation study results on the classification strategies of the Classifier. We observe that the complete Classifier achieves the best performance. The Classifier achieves 90% accuracy by relaying on exact matching to establish association between devices in the target and example topologies based device's roles. This is because, in some cases, the device configuration is also highly affected by its neighborhood while exact matching cannot measure the neighborhood similarity between devices. By combining exact matching with neighborhood similarity measurement method, the Classifier enhances association establishment accuracy and synthesis speed, allowing CEGS to synthesize correct configurations with an average reduction of 8 loops.

B.2 Comparison of different LLMs in configuration generation

We evaluate the performance of different LLMs for configuration generation, including LLaMA-2-7B, LLaMA-2-13B, Mixtral-8x7B-Instruct, Gemma-1.1-7B-IT, LLama-3-70B-Instruct, and GPT-4o. Table 9 shows the comparison results. We leverage these LLMs to generate configurations under two target scenarios. One scenario consists of a small-scale star network with 5 nodes, and 6 simple no-transit intents. Another scenario consists of a larger network with 22 nodes, and one complex path-preference intent. We observed that the performance of GPT-4o is the best, followed by LLama-3-70B-Instruct. Therefore, in our evaluation, we leverage GPT-4o to generate configuration templates, due to its outstanding configuration generation ability.

Table 7: Ablation study results on the recommendation strategies of the Querier. U stands for unsuccessful generation.

Intent similarity	Intent preprocessing	Intent graph similarity	Accuracy	Loops
✓	×	×	85%	200(U)
✓	✓	×	100%	28
✓	✓	✓	100%	10

Table 8: Ablation study results on the classification strategies of the Classifier.

Exact matching	Neighborhood similarity	Accuracy	Loops
✓	×	90%	18
✓	✓	100%	10

C A case study

We provide a complete case to show our CEGS how to synthesize correct configurations, given a target network scenario. Figure 15 shows a target network scenario consisting two user intents and a target network topology.

C.1 Identifying configuration examples

We utilize the Querier to recommend the most relevant example for each user intent. Take intent f_1 as an example, we elaborate on how we use the Querier to recommend a configuration example. First, we normalize the intent f_1 into an unified expression format of *"BGP routing. Traffic from source to destination prefers path1 over path2"* using GPT-4o

You are an expert in writing network configurations. We provide you with a configuration example. Given a target network scenario including a target topology specified in a JSON dictionary and the target intent, your task is to generate the configuration template for each device in the target topology, following the configuration example, so that the target intent is satisfied.

Configuration example:

Topology T1: {" devices": [" U_A " : {AS: 100}, ...], " links": [{"node1": {"name": "U_A", "interface": "Ethernet2/0/0", "ip address": "10.17.43.1/24"}, "node2": {{"name": "U_B", "interface": "Ethernet2/0/0", "ip address": "10.17.43.2/24}}, ...]}

Intent I1: All routers are configured with the BGP protocol. When traffic originates from router U_A and is destined for U_D, it prioritizes the path (U_A, U_B, U_D) over (U_A, U_C, U_D).

Configurations:

Configuration of U_A : hostname U_A \n interface Ethernet2/0/0\n ip address 10.17.43.1/24 \n …

……

Target network scenario:

Target topology T2: {" devices": [" R_A " : {AS: 10}, ...], " links": [{"node1": {"name": "R_A", "interface": "Ethernet1/0/0", "ip address": "10.25.32.5/24"}, "node2": {{"name": "R_B", "interface": "Ethernet1/0/0", "ip address": "10.25.32.10/24}}, ...]}

Target Intent I2: Each router runs the BGP protocol. Traffic form router R_G to R_D selects the path (R_G, R_F, R_D) as preferred over (R_G, R_E, R_D).

Association relations between routers in the target topology T2 and routers in example topology T1:

$R_A : U_E$, $R_B : U_E$, $R_C : U_F$, $R_D : U_F$, $R_E : U_C$, $R_F : U_B$, $R_G : U_A$

Format requirements for configuration template:

This configuration template considers OSPF cost settings and BGP route-map configuration as templates, while the rest remains as specific configurations. The format of OSPF cost configuration template and BGP route-map configuration template are:

1. OSPF cost configuration template: this template leaves the ospf cost as symbol, represented by [value].

2. BGP route-map policy configuration template:

1) Follow the import route-map name and export route-map name, which are defined as 'RMap_{router}_from_{neighbor}' and 'RMap_{router}_to_{neighbor}', respectively, (where router and neighbor should be replaced with the router's name and its neighbor's name.).

2) The route-map configuration template should leave policy parameters (e.g., local preference value, community value) as symbolic which is represented by [value]. For instance:

```
ip community list 1 [value]
route-map RMap_RouterF_from_RouterM permit 10
 match community 1
 set local-preference [value]
 set community [value] additive
```

3) If the route-map configuration uses the community or prefix, you must first provide the corresponding definition of "ip community list" or "ip prefix-list". The exclamation mark (!) should be used to separate the route-map configuration of different neighbors.

Please directly generate the configurations for routers {R_B, R_F, …} in the target topology T2, referring to the configurations of their association nodes in the example topology T1, so that the target intent I2 is satisfied. Note: You cannot provide any comments in the response, nor provide any comments in the generated configuration.

Figure 14: Example prompt for generating configuration templates.

Table 9: Comparison results of different LLMs for configuration generation. U stands for unsuccessful generation.

Target network scenario	Models					
	LLaMA-2-7B	LLaMA-2-13B	Mixtral-8x7B-Instruct	Gemma-1.1-7B-IT	LLaMA-3-70B-Instruct	GPT-4o
5-node network, 6 no-transit intents	50(U)	50(U)	50(U)	50(U)	5	2
22-node network, 1 path-preference intent	50(U)	50(U)	50(U)	50(U)	30	4

with the few-shot prompt shown in Figure 10. The intent of each configuration example is also normalized into a specific expression format during the configuration example parsing process. For example, the intents of configuration examples shown in 16 and 17, also are normalized as *"BGP routing. Traffic from source to destination prefers path1 over path2"*. These two examples, whose intent semantically match intent f_1 with a similarity score of 1, are considered potential reference examples for intent f_1. We then construct the corresponding user intent graph and two example intent graphs, and using GraphSAGE to calculate the similarity between them. The similarity between the example intent graph of configuration example in Figure 16 and the user intent graph is 0.86, while the similarity between the example intent graph of configuration example in Figure 17 with the user intent graph is 0.75. Thus, we select the configuration example z_1 in Figure 16 as the example associated with intent f_1. Similarly, we recommend the best configuration example for intent f_2. The configuration example in Figure 17 is chosen due to its higher similarity between its example intent graph and the user intent graph corresponding to intent f_2. From Figures 15, 16, and 17, we can observe that when considering intent f_1, the example intent graph corresponding to example z_1 is more similar to the intent graph for f_1, particularly since the nodes R_A, R_B, and R_D in Figure 15 closely match the node attributes and neighborhood relations of nodes V_A, V_B, and V_D in Figure 2. However, only node R_A exhibits high similarity to node U_A in 17. Therefore, the user intent graph for f_1 is closer to the example intent graph of configuration example z_1.

Intents:

Intent f_1: Each router runs the BGP protocol. Traffic form router R_A to R_E selects the path (R_A, R_C, R_E) as preferred over (R_A, R_B, R_D, R_E).

Intent f_2: Each router runs the BGP protocol. Traffic form router R_G to R_D selects the path (R_G, R_F, R_D) as preferred over (R_G, R_E, R_D).

Topology:

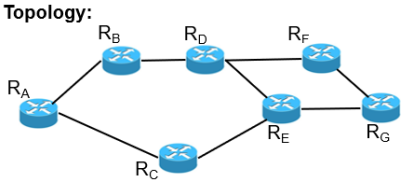


Figure 15: A target network scenario.

Intent: All routers are configured with the BGP protocol. When traffic originates from router V_A and is destined for V_F, it prioritizes the path (V_A, V_B, V_D, V_F) over (V_A, V_C, V_E, V_F).

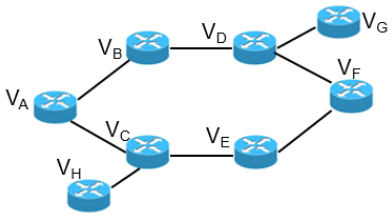


Figure 16: Configuration example z_1 for the user intent f_1.

C.2 Establishing the association between devices in the target and example topologies

For each user intent f_1, we utilize the Classifier to establish the association between the device in the target topology and those in the topology of example z_i corresponding the intent f_i. Figure 18 shows the association results with respect to each example.

C.3 Generating configurations

We first leverages GPT-4o to generate configuration templates for each device in the target topology with respect to each user intent in turn. Figure 14 shows the example prompt for generating configuration templates. Take router R_D in the target topology as an example, Figure 19 and Figure 20 show the configuration template for router R_D with respect to intent f_1 and intent f_2, respectively. We use the Syntax Verifier, LAV, and GFV to verify these templates. We then merge the configuration templates for each device with respect to the two user intents. Figure 21 shows the merged templates for router R_D. Finally, we use the Formal Synthesizer to specify the specific parameters in the templates. Figure 22 shows the full configuration of router R_D.

D Discussion on user guide and generalization for CEGS

In this section, we first provide a user guide detailing how network operators can use our CEGS to synthesize configurations in the real network setting. We then provide a comprehensive guide detailing the steps a network operator needs to follow to generalize CEGS with new intents.

Intent: All routers are configured with the BGP protocol. When traffic originates from router U_A and is destined for U_D, it prioritizes the path (U_A, U_B, U_D) over (U_A, U_C, U_D).

Topology:

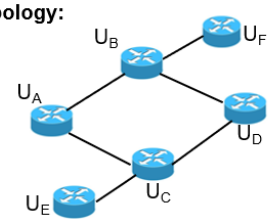


Figure 17: Configuration example z_2 for the user intent f_2.

Association relations between devices in the target topology and those in the topology of the example z_1:

$R_A : V_A$, $R_B : V_B$, $R_C : V_C$, $R_D : V_D$, $R_E : V_F$, $R_F : V_G$, $R_G : V_G$

Association relations between devices in the target topology and those in the topology of the example z_2:

$R_A : U_E$, $R_B : U_E$, $R_C : U_F$, $R_D : U_F$, $R_E : U_C$, $R_F : U_B$, $R_G : U_A$

Figure 18: Association between devices in the target and example topologies.

D.1 A user guide

To utilize CEGS to synthesize configurations for devices in a real network, the network operator needs to prepare the operator intents, network topology in JSON format, and the relevant device documentation. For example, consider a cloud network consisting of three Autonomous Systems (ASes), as shown in Figure 23(b). The operator's network, AS1000, includes four routers: RouterA, RouterB, RouterC, and RouterD. This network is connected to one customer peer AS2000, and two external peers, AS3000 and AS4000. The operator aims to manage traffic flow transmission from the customer peer to two external peers. For this, the operator first needs to draft routing intents in natural language for these traffic flows. Figure 23(a) provides the intent example. The operator then should describe the network topology in JSON format, as illustrated in Figure 23(b). Finally, the operator needs to prepare the device documentation, which should include a configuration guide that contain many configuration examples. At the same time, the operator must ensure that at least one configuration example related to each specified intent is included in the documentation. Figure 23(c) provide a sample documentation example from the cisco[6].

D.2 How to generalize CEGS?

Our CEGS supports static route, OSPF, and BGP configurations across variety of routing intents supported by the Formal Synthesizer (NetComplete). To generalize CEGS to new in-

[6] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_ospf/configuration/15-mt/iro-15-mt-book/iro-cfg.html#GUID-EF90EB2B-A283-4F7A-B044-C44BAE9F4E0A

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
ip community-list 1 permit 'EMPTY?Value'
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 'EMPTY?Value'
 set community 'EMPTY?Value' additive
!
route-map RMap_RD_to_RB 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
route-map RMap_RD_to_RF 'EMPTY?Value' 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 19: Configuration template of router R_D for intent f_1.

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
interface Ethernet4/0/0
 ip address 10.2.10.12 255.255.255.254
 description "To PeerRD"
 speed auto
 duplex auto
!
ip community-list 1 permit 'EMPTY?Value'
!
route-map RMap_External_RD_from_PeerRD permit 10
 set community 'EMPTY?Value' additive
 set local-preference 'EMPTY?Value'
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 'EMPTY?Value'
 set community 'EMPTY?Value' additive
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_to_RB 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.10.12 remote-as 10010
 neighbor 10.2.10.12 description "To PeerRD"
 neighbor 10.2.10.12 advertisement-interval 0
 neighbor 10.2.10.12 soft-reconfiguration inbound
 neighbor 10.2.10.12 send-community
 neighbor 10.2.10.12 route-map RMap_External_RD_from_PeerRD in
 neighbor 10.2.10.12 route-map RMap_RD_to_PeerRD out
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 21: Configuration template of router R_D for intent f_1 and intent f_2.

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
interface Ethernet4/0/0
 ip address 10.2.10.12 255.255.255.254
 description "To PeerRD"
 speed auto
 duplex auto
!
route-map RMap_External_RD_from_PeerRD permit 10
 set community 'EMPTY?Value' additive
 set local-preference 'EMPTY?Value'
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 'EMPTY?Value'
 set community 'EMPTY?Value' additive
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_to_RB 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.10.12 remote-as 10010
 neighbor 10.2.10.12 description "To PeerRD"
 neighbor 10.2.10.12 advertisement-interval 0
 neighbor 10.2.10.12 soft-reconfiguration inbound
 neighbor 10.2.10.12 send-community
 neighbor 10.2.10.12 route-map RMap_External_RD_from_PeerRD in
 neighbor 10.2.10.12 route-map RMap_RD_to_PeerRD out
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 20: Configuration template of router R_D for intent f_2.

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
interface Ethernet4/0/0
 ip address 10.2.10.12 255.255.255.254
 description "To PeerRD"
 speed auto
 duplex auto
!
ip community-list 1 permit 100:0
!
route-map RMap_External_RD_from_PeerRD permit 10
 set community 100:0 additive
 set local-preference 100
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 100
 set community 100:0 additive
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_to_RB permit 100
!
route-map RMap_RD_to_RE permit 100
!
route-map RMap_RD_to_RE permit 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.10.12 remote-as 10010
 neighbor 10.2.10.12 description "To PeerRD"
 neighbor 10.2.10.12 advertisement-interval 0
 neighbor 10.2.10.12 soft-reconfiguration inbound
 neighbor 10.2.10.12 send-community
 neighbor 10.2.10.12 route-map RMap_External_RD_from_PeerRD in
 neighbor 10.2.10.12 route-map RMap_RD_to_PeerRD out
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 22: Full configuration of router R_D complying with intent f_1 and intent f_2.

tents for other protocols or network techniques (e.g., VLAN) on specific vendor devices, the user extends CEGS by following six steps:

Configuration example preparation. The user first needs to collect relevant device documentation which contain configuration guide and ensures the configuration guide cover some configuration examples related to the target scenarios.

GraphSAGE training. The Querier and Classifier depend on a GraphSAGE model to recommend examples and build association between devices in the target and example topologies. To achieve this, the user needs to train a GraphSAGE model. First, the user should construct a intent graph dataset that contains numerous matching pairs of intent graphs. These intent graphs should cover the relevant target scenarios, including similar intents and topology structures. Each data in the dataset consists of a pair of intent graphs and the labels indicating the associations between nodes in the two graphs. The user then trains the GraphSAGE model on this dataset using the training method described in Sec.3.2.3. The user may need to tune the GraphSAGE model by adjusting hyperparameters such as the number of layers, number of neurons, and learning rate.

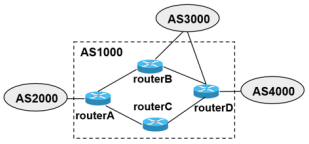
Prompt crafting. The user needs to craft the prompt for the LLM to generate configuration templates. The prompt, as shown in Figure 14, consists of six parts: background description, configuration example, target network scenario, association relation between devices in the target and example topologies, format definition of the configuration template, and instructions. The users should tailor the template format definition according to the configuration templates they need to generate. The user can design customized instructions as needed. In addition, the user also needs to craft other prompts, such as those for intent normalization and role extraction, as described in A.

LAV extension. CEGS develops the LAV to locally verify whether each device template complies with the the target topology and user intent descriptions, such as interface configurations and protocol basic configurations. The user can easily

Intent 1: Configure BGP process on each routers to implement the policy: traffic from AS2000 to AS3000 prefers the path <AS2000, routerA, routerB, AS300> over the path < AS2000, routerA, routerC, routerD, AS3000 >.

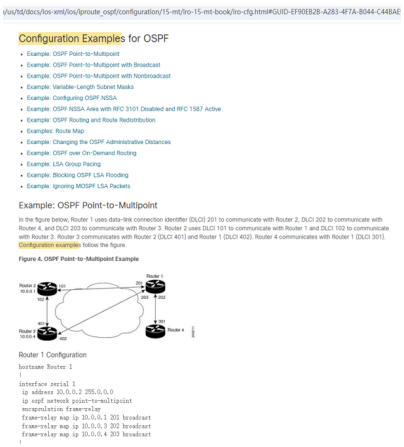
Intent 2: Configure OSPF process on each routers to implement the policy: traffic from AS2000 to AS400 is load-balanced between the path <AS2000, routerA, routerB, routerD, AS4000> and the path <AS2000, routerA, routerC, routerD, AS3000>.

(*a*) Intent description




```
{
    "routers": ["routerA":{"AS": 1000,...}, ...],
    "links": [
      {
          "node1": {
              "name": "routerA",
              "interface": "GigabitEthernet1/0/0",
              "ip address": "10.90.17.1/24"
          },
          "node2": {
              "name": "routerB",
              "interface": "GigabitEthernet1/0/0",
              "ip address": "10.90.17.2/24"
          }
      },
      ...
    ]
}
```

(*b*) Topology description


Configuration Examples for OSPF

- Example: OSPF Point-to-Multipoint
- Example: OSPF Point-to-Multipoint with Broadcast
- Example: OSPF Point-to-Multipoint with Nonbroadcast
- Example: Variable-Length Subnet Masks
- Example: Configuring OSPF NSSA
- Example: OSPF NSSA Area with RFC 3101 Disabled and RFC 1587 Active
- Example: OSPF Routing and Route Redistribution
- Examples: Route Map
- Example: Changing the OSPF Administrative Distances
- Example: OSPF over On-Demand Routing
- Example: LSA Group Pacing
- Example: Blocking OSPF LSA Flooding
- Example: Ignoring MOSPF LSA Packets

Example: OSPF Point-to-Multipoint

In the figure below, Router 1 uses data-link connection identifier (DLCI) 201 to communicate with Router 2, DLCI 202 to communicate with Router 4, and DLCI 203 to communicate with Router 3. Router 2 uses DLCI 101 to communicate with Router 1 and DLCI 102 to communicate with Router 3. Router 3 communicates with Router 2 (DLCI 401) and Router 1 (DLCI 402). Router 4 communicates with Router 1 (DLCI 301). Configuration examples follow the figure.

Figure 4. OSPF Point-to-Multipoint Example

Router 1 Configuration

```
hostname Router 1
!
interface serial 1
 ip address 10.0.0.2 255.0.0.0
 ip ospf network point-to-multipoint
 encapsulation frame-relay
 frame-relay map ip 10.0.0.1 201 broadcast
 frame-relay map ip 10.0.0.3 202 broadcast
 frame-relay map ip 10.0.0.4 203 broadcast
!
```

(*c*) Device documentation

Figure 23: Example input for CEGS.

adapt the LAV for other protocols or network techniques by implementing checks for the protocol-specific basic settings or network technique settings, as well as interface attribute settings when involved.

GFV extension. CEGS develops the GFV to perform global verification on templates across all devices, ensuring that configurations with interdependencies and cross-device impacts are properly validated. For example, BGP route-map policy configurations on different devices influence one another and work together to achieve the intended global routing policy. The user should extend the GFV to globally validate configurations that affect each other across all devices involved in fulfilling the intended network behavior. The GFV provides localized feed about configuration errors for LLM correction. The user can replace the LAV or GFV using other verification tools that can provide localization feedback, indicating which specific configuration on which devices are incorrect.

Formal Synthesizer selection. The user needs to select or develop appropriate Formal Synthesizer that is capable of filling correct parameters in the templates to achieve the intent.

D.3 Why does CEGS need to use formal methods?

In configuration synthesis tasks, it is a significant challenge for LLMs to accurately calculate network policy parameters based on the entire network information. For example, achieving a load-balancing intent under the OSPF protocol requires accurately calculating the cost of each link in the network, which is difficult for LLM reasoning. Therefore, it is essential to use formal method to calculate network policy parameters to guarantee the configuration correctness.

Suppressing BGP Zombies with Route Status Transparency

Yosef Edery Anahory[1], Jie Kong[2], Nicholas Scaglione[2], Justin Furuness[2], Hemi Leibowitz[3], Amir Herzberg[2], Bing Wang[2], Yossi Gilad[1]

[1]*School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem, Israel*
[2]*School of Computing, University of Connecticut, Storrs, CT*
[3]*Faculty of Computer Science, The College of Management Academic Studies, Rishon LeZion, Israel*

Abstract

Withdrawal suppression has been a known weakness of BGP for over a decade. It has a significant detrimental impact on both the reliability and security of inter-domain routing on the Internet. This paper presents *Route Status Transparency* (RoST), the first design that efficiently and securely thwarts withdrawal suppression misconfigurations and attacks. RoST allows ASes to efficiently verify whether a route has been withdrawn; it is compatible with BGP as well as with BGP security enhancements. We use simulations on the Internet's AS-level topology to evaluate the benefits from adopting RoST. We use an extensive real-world BGP announcements dataset to show that it is efficient in terms of storage, bandwidth, and computational requirements.

1 Introduction

The Border Gateway Protocol (BGP) determines how packets route between Autonomous Systems (ASes) on the Internet. Despite its crucial role in facilitating Internet communication, BGP is fragile, and when failures occur, they can cause significant disruptions. Some failures are due to benign errors and misconfigurations, while others are attributed to malicious actions [29, 41]. Many of these issues are addressed by the increasing adoption of the Resource Public Key Infrastructure (RPKI) [11, 25, 33], which allows an IP prefix owner to associate it with the ASes allowed to announce the prefix through BGP. Furthermore, the networking community has also devoted tremendous efforts to design and standardize protocols for route authentication (i.e., beyond the origin), most notably, BGPsec [6]. With route validation in place, announcing routes with invalid sources or bogus links becomes less of a problem for global Internet routing.

However, the authentication mechanisms do not solve the problem of *freshness*. That is, how can an AS know whether a BGP announcement has been withdrawn or not since it was announced? This key problem is the root cause of stuck routes, also known as "zombie" routes, where ASes choose to route traffic through withdrawn routes because these ASes are unaware that these routes have been withdrawn [15, 35]. This happens when some AS along the route withdraws an announcement, but a downstream AS does not propagate the withdrawal (intentionally or due to an error). The once valid route keeps appearing "alive" to the following ASes, who may opt to use it and export it further, accidentally misleading other ASes. Since the zombie route *was* legitimate, it continues to pass the route authentication checks even with RPKI and full-fledged BGPsec. Zombie routes can cause disconnections when an AS mistakenly sends traffic to a neighbor who no longer has a valid route to the destination IP address (since that route was withdrawn). Moreover, zombie routes can lead to inefficient routing where an AS would think it uses a better (e.g., cheaper or shorter) route to reach some destination, but a downstream AS will route the traffic differently than expected. In some instances, this misrouting has even led to the creation of routing loops [30]. In many cases, withdrawal suppression is attributed to benign errors rather than deliberate attacks [15], illustrating the fragility of Internet routing.

Currently, the only proposed approach to solve this problem is *key rollover* [39, 42], which requires route validation (i.e., BGPsec) to already be deployed. In key rollover, keys are rotated periodically; as a result, signed routes have a limited validity period and cannot be reused 'forever'. Thus, after a key rollover, the routes signed by the old key cannot be authenticated by the new key. This means that a withdrawal suppression can only be effective until the current keys are rotated. However, key rollover requires a hefty operational effort, since the AS's BGPsec key pair needs to be updated, a new certificate must be issued, and the old one revoked (if the rollover occurs when the previous key is still valid).

In this work, we present *Route Status Transparency (RoST)*, a novel mechanism for mitigating withdrawal suppression. RoST-adopting ASes publish cryptographically-signed information about the routes they export to protect their announcements against withdrawal suppression. Other-RoST adopting ASes download and use this information to verify whether any of the routes they receive were actually

withdrawn. Specifically, RoST uses the transitive attribute capability to identify and monitor routes. Namely, every AS in the path will generate a sequence of route IDs as the announcement is distributed along the path. By combining the transparent route status information with these route IDs, each AS that performs validation based on RoST can check if the announcements match the transparent information, and if not, identify withdrawn routes.

RoST detects and limits the impact of BGP withdrawal suppression to minutes, whether due to intentional attacks or misconfigurations, and adds only a negligible overhead to the BGP protocol. ASes that adopt RoST deploy a local machine (an "agent") that synchronizes with the route status transparency reports and identifies withdrawn routes. The agent machine performs the heavy lifting of validating these reports and then configures the local BGP routers to withdraw invalid routes. In contrast to key rollover, RoST can be adapted with vanilla BGP to mitigate the zombie routes phenomenon, thus circumventing the challenges involved in the deployment of BGPsec and the costs of key rollover. We evaluate RoST in terms of overhead using routing data from multiple vantage points on the Internet and compare it with key rollover.

In summary, this work addresses withdrawal suppression, a fundamental problem in today's Internet routing that will persist even when route authentication mechanisms are fully deployed. We design and evaluate the RoST protocol, the first protocol that efficiently detects and mitigates withdrawal suppression. RoST is compatible with the currently deployed BGP, and we show how it can be deployed without requiring changes to today's routers.

2 Background

Route withdrawals and announcement suppression. A BGP router can withdraw a route either explicitly by sending a withdrawal announcement or implicitly by sending an updated announcement. In Figure 1, we illustrate both explicit and implicit withdrawals. The figure illustrates a scenario in which AS 100 initially chooses AS 666 as its provider, and thus AS 100 sends an announcement with prefix 1.2/16 and AS-path 100 to its provider, AS 666. Then, AS 100 decides to switch its provider to AS 300, so it sends an *explicit* withdrawal announcement to AS 666. In an *implicit* withdrawal, AS 100 sends a new announcement to AS 666 with an artificially inflated AS-path, 100-100-100, known as 'path prepending'. This makes the path from AS 100 to AS 666 three-hop long, making the path from AS 100 to its current provider, AS 300, more favorable (e.g., AS 200 will route to prefix 1.2/16 through AS 300, instead of AS 666). A withdrawal suppression happens when some AS on the route fails to propagate a withdrawal message or an alternative route (explicit/implicit withdrawal, respectively). This can be unintentional due to misconfigurations or software bugs in

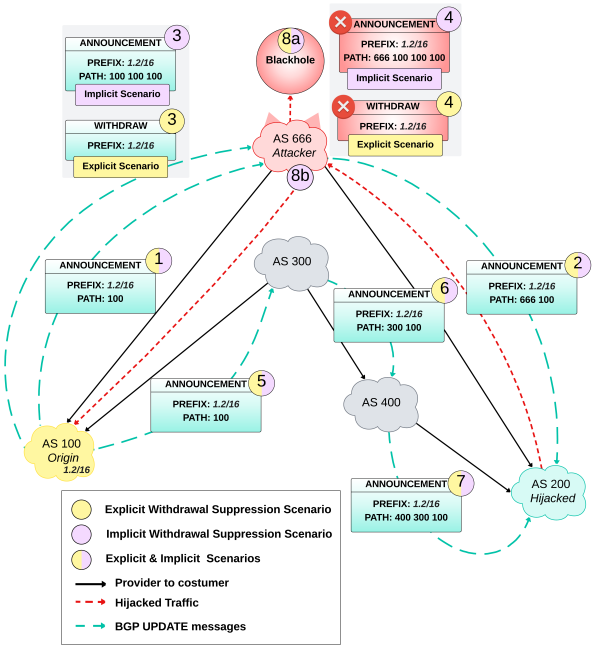


Figure 1: Illustration of explicit and implicit withdrawal suppression attacks. The diagram highlights the sequence of steps involved in each attack and the actions taken by AS 666. Steps unique to explicit withdrawal suppression are marked in yellow with a "b" label, and steps unique to implicit withdrawal suppression are marked in purple with an "a" label. Steps that are common to both scenarios are shown with a combination of yellow and purple.

BGP routers, or intentional, for malicious purposes such as to keep traffic flowing through a route with the attacker's AS (AS 666 in Figure 1).

Stuck or zombie routes. Measurement studies [15, 35] using route beacons confirm that zombie outbreak happens daily, which can cause traffic to follow undesirable and unavailable paths, leading to network congestion, degraded service quality, or even complete loss of connectivity. Specifically, the analysis in [15] reveals that the impact of zombie routes is influenced by the tier of the network where the zombie route was detected, which has significant implications. The authors demonstrate that zombie routes detected within a Tier-1 AS can affect the majority of ASes. A real-world example of this occurred on August 30, 2020, when CenturyLink, which had acquired Level 3 Communications (a Tier-1 AS), experienced a significant incident. According to a report by Cloudflare [36], CenturyLink inadvertently applied a firewall rule that blocked all internal BGP traffic, leading to extensive outages across their network. What made this incident particularly severe was the failure of affected routers to issue BGP withdrawal messages. Without these messages, external networks (including Cloudflare) continued to route

traffic through CenturyLink, unaware of the internal issues. The outage lasted several hours, and many networks were forced to manually reconfigure their systems to reroute traffic away from CenturyLink/Level 3.

3 RoST Design

In this section, we detail the design of RoST. We start with its goals (§3.1) and then present a high-level system overview (§3.2). We then detail how RoST agents maintain routes' status (§3.3), how the routes' status is made transparent (§3.4), how agents track the status of these routes (§3.5) and how routes advertisements are validated (§3.6). A detailed pseudocode for the main RoST operations is presented in Appendix A.

For simplicity, in this section we assume a single trusted repository and refer to it in a singular form ("the repository"). Later, in §7.2, we discuss how to extend it to support multiple repositories and to overcome repository faults. Finally, when we refer to AS-Paths, we write the path from left to right, where the rightmost AS is the origin of the IP prefix.

3.1 Design Goals

RoST aims to detect and mitigate withdrawal suppression in inter-domain routing on the Internet. It does not deal with path manipulation attackers that may alter or spoof route advertisements. Such attackers should be handled by path authentication mechanisms [6, 32]. We design it to achieve the following three desiderata:

- Efficiency, i.e., low computation, communication, and storage overhead.
- Compatibility with BGP and future security enhancement of BGP (e.g., BGPsec [6], BGP-iSec [32]). Namely, it should be possible to deploy RoST without extending or standardizing new routing protocols.
- Compatibility with BGP routers. That is, it should be possible to mitigate withdrawal suppression by having local RoST agent automatically issue commands to the router through existing command line or application interfaces rather than requiring specialized router software or hardware support.

3.2 High-Level Operation

RoST detects withdrawal suppression by providing route status transparency. We outline its operation following Algorithm 1. Each adopting AS deploys a RoST *agent*, which performs three main functions: (1) report the current routes of the AS to a public repository (Algorithm 1, lines 6-7), (2) retrieve the current routes of other ASes from the repository (Algorithm 1, lines 8–13), and (3) validate advertised BGP routes without relying on propagation of withdrawal announcements (Algorithm 1, lines 14–19).

Algorithm 1 High-Level Operations of RoST Agent

```
1:  Input:
2:      AgentAS: AS number for which the agent is responsible
3:      BatchInterval: Time interval for periodic reporting
4:      R: Trusted repository
5:  procedure RoSTAgent(AgentAS, BatchInterval, R)
6:      // 1. Initialize asynchronous tasks
7:      Event 1 - Reports local routes to R every BatchInterval
8:      Event 2 - Subscribe to route updates from R
9:      Event 3 - Listen for BGP update messages from the router
10:     // 2. Handle incoming events
11:     while true do
12:         event ← Wait for the next available event
13:         if event is "Periodic Report Trigger" (Event 1) then
14:             Report local routes to R
15:         else if event is "Route Update from R" (Event 2) then
16:             Validate impacted routes
17:             if a route is invalid then
18:                 Propagate withdraw or alternative route
19:         else if event is "BGP Update Message" (Event 3) then
20:             Validate the new route using repository data
21:             if the route is invalid then
22:                 Propagate withdraw or alternative route
23:             Update route status before forwarding BGP message
```

Using a standalone agent that is separate from BGP routers promotes simpler deployment because it does not require replacing or updating existing BGP routers. Specifically, when the agent detects a route that was withdrawn but the withdrawal was suppressed, the agent instructs the BGP routers of the AS to remove the withdrawn route using the routers' existing API.

To illustrate how RoST detects withdrawal suppression, consider the scenario depicted in Figure 2. AS 4 has two available routes to a prefix that originates at AS 1, either via path 5-3-2-1 or 6-2-1. Assume that AS 4 chose to route through AS 6 as that AS path is shorter. Suppose that at some point, AS 2 decides to stop using AS 6 as a provider, and as a result, AS 2 sends a withdrawal update message to AS 6; however, AS 6 suppresses this withdrawal. This means that AS 4 would continue to route traffic through AS 6 since it is unaware of the withdrawal.

Using RoST, AS 4 *can* detect and mitigate this suppression. That is, once AS 2 issues the withdrawal, its agent updates the repository about the withdrawal. This allows AS 4's agent to learn about the withdrawal from the repository despite AS 6 suppressing it. To authenticate the information recorded at the repository, RoST assumes a standard RPKI deployment. This allows agents to sign the routes' status information they make transparent, and other agents to verify its authenticity. Given the increasing adoption of RPKI, with most unique prefix-

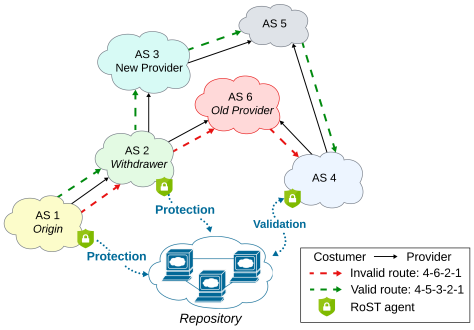


Figure 2: High-level overview of RoST operations. AS1, the origin AS, and AS2, the withdrawer, both perform the protection role by ensuring that their routing information is up-to-date and uploaded to the trusted repository when changes occur. When AS2 switches providers from AS6 to AS3, AS6, the old provider, continues to attract traffic using an outdated route. AS4, a customer of AS6, takes on the validation role by retrieving the updated route status from the trusted repository. This allows AS4 to detect any withdrawal suppression by AS6.

origin pairs already having valid RPKI records (54% [2]), we believe that it will not hinder RoST's adoption.

3.3 Tracking Route Status

An AS's agent maintains the status of the AS's routes using vectors called Route Status Vectors (RSVs). Each RSV represents the route updates to a specific IP prefix and whether or not this route is active. That is, an RSV is in the form of: $(Prefix, RouteID, Status)$.

The agent updates the repository with batches of route updates. These updates are identified using a *RouteID* that consists of two counters, $RouteID = (BatchID, PathID)$. The BatchID counter identifies the batch number where that update occurred, and the PathID counter indexes the specific path update for the RSV's prefix within that batch. This design decision is based on our analysis of real-world route announcements collected by public route collectors (see §4). Our analysis shows that while the average number of path changes per prefix remains under 20 per day, some prefixes can experience more than 1 million route changes per day (see Figure 5a). This can be caused by multiple reasons, such as path prepending, provider change, or path finding [15]. By periodically sending batched updates, RoST minimizes its space requirements: a burst of path changes would be 'absorbed' into one BatchID; PathID would reset (to 0) in the next batch, which avoids the rapid counter increments if RoST were to use a single counter. The PathID would increment within a batch only if the route has changed and still active. If the route is withdrawn, only the status field changes to invalid.

Every RoST agent maintains a set of RSVs per neighboring AS, reflecting the routes that were announced to the specific neighbor. We refer to this set as RSV-Out, and denote the set of RSVs that the agent of AS x maintains for neighboring AS y as RSV-Out$_{(x,y)}$. We refer to the pair (x,y) as the *interface* between AS x and AS y. The RSV-Out$_{(x,y)}$ structure also contains a *BatchCounter*, which is incremented periodically at each batch interval. To illustrate, refer to step ① in Figure 3, where we present how the entries of RSV-Out$_{(100,200)}$ are maintained at the end of batch 54 (i.e., BatchCounter is 54) in five different scenarios:

1. *Prefix 1.1/16, no changes during the current batch*: This prefix has an older BatchID (48) and PathID (1), meaning that the route to this prefix has not changed since batch 48. The status of this route remains active (T).
2. *Prefix 1.2/16, one route change*: The BatchID has been updated from 51 to 54, initialized with a new PathID (1). This reflects that the route to this prefix has been updated once in the current batch and remains active.
3. *Prefix 1.3/16, withdrawn route*: This prefix was previously active (T), but its status has been changed to withdrawn (F), i.e., the route has been withdrawn in the current batch (BatchID 54 and PathID 0).
4. *Prefix 1.4/16, multiple route changes*: Similarly to the case of prefix 1.2/16, but in this instance, there were three different route change announcements within the current batch, as indicated by the PathID of 3.
5. *Prefix 1.5/16, withdrawn route after multiple changes*: This prefix resembles the scenario with prefix 1.4/16, but the route has been withdrawn after two active route announcements (PathID 2) within the current batch, changing its status to withdrawn (F).

3.4 Making Routes' Status Transparent

At the end of each batch interval, the agent updates the repository about the routes that changed during that interval. It generates for every RSV-Out a corresponding delta, ΔRSV-Out, and reports it to the repository. To illustrate, refer to step ② in Figure 3, where ΔRSV-Out$_{(100,200)}$ contains all route status updates that were made to RSV-Out$_{(100,200)}$ during batch 54. Specifically, ΔRSV-Out$_{(100,200)}$ contains all the prefixes in RSV-Out$_{(100,200)}$ except prefix 1.1/16, since it is the only prefix that has not changed during batch 54.

In addition, each ΔRSV-Out contains three other values: (1) the batch counter, (2) a Merkle tree root hash committing to all route updates (the leaves are the entries of the (updated) RSV-Out), and (3) a signature over the batch counter, the interface pair, and the Merkle root, signed by the AS's private key (based on the RPKI deployment assumption). The Merkle tree commitment enables agents from other ASes to fetch an authenticated *subset* of the ΔRSV-Out that only contains updates relevant to the fetching agent, as we explain in §3.5.

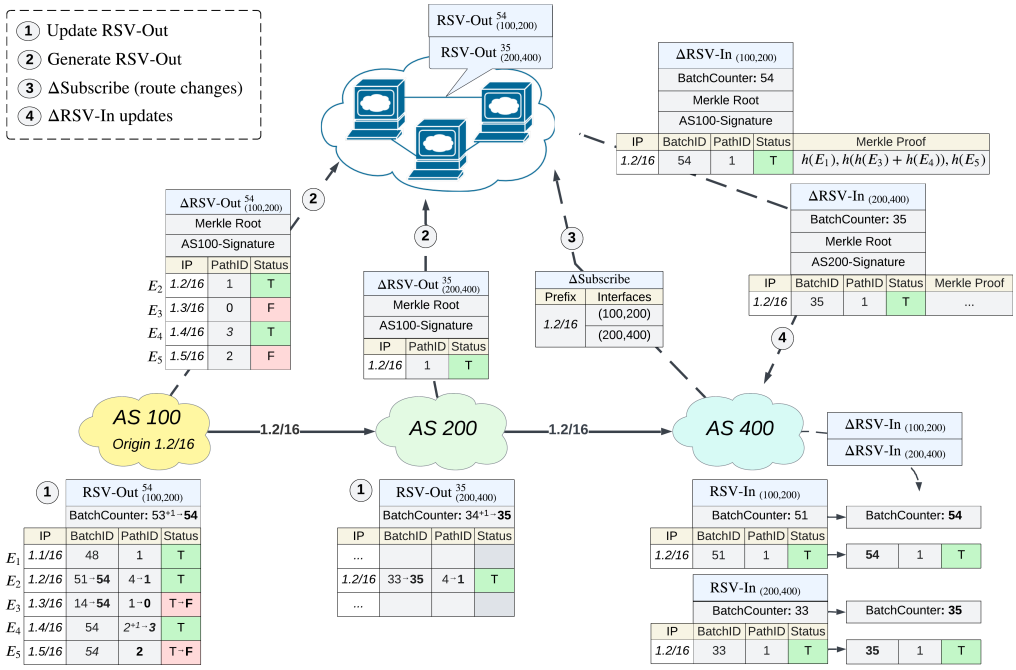


Figure 3: The diagram illustrates the steps involved in RSV processing and distribution between ASes and the trusted repository. AS 100 and AS 200 publish a ΔRSV-Out to the trusted repository, which contains updates for several prefixes. AS 400, which monitors the interfaces (100, 200) and (200, 400), subscribes to 1.2/16 for those interfaces to receive updates. The trusted repository sends ΔRSV-In when there are new updates, which AS 400 then merges with its existing RSV-In data. This ensures that AS 400 has the most current and validated route statuses, protecting the network against the use of outdated or invalid routes.

3.5 Retrieving Route Transparency Data

To learn about route changes, agents receive updates about routes from the repository. However, each agent only cares about some of the RSVs, pertaining to the routes used by their AS's BGP routers. Thus, when a router chooses to use a route to a specific prefix, the agent subscribes to updates about the current RSVs for that prefix from all the ASes on the route's AS-Path. For example, if AS w uses a route with AS-Path $x-y-z$, then the agent of AS w subscribes to RSVs of the routes that z publishes to y, y publishes to x, and x publishes to w, because suppression can occur in each of these hops.

To track these RSVs, the agents store them per interface; we refer to them as RSV-Ins. To illustrate, observe AS 400 in Figure 3, which routes traffic to prefix $1.2/16$ using the AS-Path 400-200-100. As a result, AS 400 tracks the RSVs to this prefix from AS 100 to AS 200 (in RSV-In$_{(100,200)}$) and from AS 200 to AS 400 (in RSV-In$_{(200,400)}$).

To learn about RSVs updates, the agent sends a message called Δ*Subscribe* to the repository, indicating for each prefix which RSVs it is interested in. In response, the repository provides the agent with the latest RSV-Ins according to the subscription; it keeps the agent current using ΔRSV-In messages whenever the relevant RSVs update. For example, in Figure 3, AS 400 notifies the repository that it is interested in the RSV of prefix $1.2/16$ that AS 100 maintains for AS 200 and the RSV of prefix $1.2/16$ that AS 200 maintains for AS 400 (shown in step ③). Before batch number 54, the BatchID of the prefix $1.2/16$ in RSV-In$_{(100,200)}$ at AS 400 was 51. AS 100 and AS 200 changed the route to $1.2/16$ during batch 54 and 35 (respectively), and both ASes sent a ΔRSV-Out reflecting this change (shown as part of step ②). Thus, the repository sends two updates to the agent of AS 400 (based on its subscription): ΔRSV-In$_{(100,200)}$ and ΔRSV-In$_{(200,400)}$, with the updated RSVs (shown as part of step ④).

Using the Δ*Subscribe* request over querying for updates each time allows agents to receive updates sooner without repeatedly polling the repository. It is more efficient since the agent sends Δ*Subscribe* only once after the BGP router selects a new route.

Importantly, the repository attaches proofs to its updates using the information recorded in the latest ΔRSV-Out. The repository attaches the latest batch counter, the Merkle tree root hash and signature from the latest ΔRSV-Out and an inclusion proof for every RSV in the ΔRSV-In (i.e., the neighboring nodes on the tree in the path from the relevant leaf update to the root hash). The repository can compute such proofs since it receives the entire ΔRSV-Out and thus can recreate the entire Merkle tree. The agent, on the other hand, only retrieves a subset of the ΔRSV-Out; it checks the signature to verify the Merkle root and uses these inclusion proofs to verify that the RSVs were indeed issued

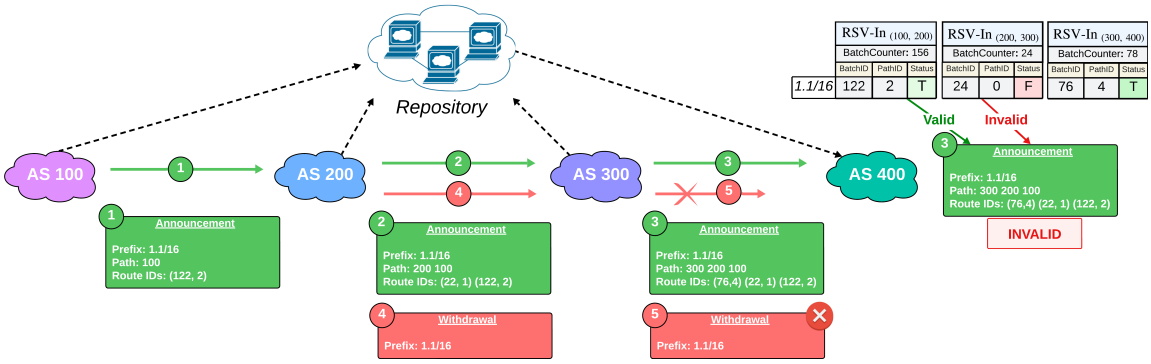


Figure 4: This diagram illustrates Route Status Validation for prefix 1.1/16 across different ASes. AS 100 originates the announcement and propagates it to AS 200, which in turn forwards it to ASes 300, 400. AS 200 later withdraws this prefix, but AS 300 suppresses this withdrawal. The corresponding RSV-In entries for each AS show the status and identifiers for the prefix. AS 400 detects the discrepancy between the announcement and the RSV-In received from AS 200 for the interface with AS 300. It marks the announcement as invalid.

by the claimed AS. After verification, the agent updates the corresponding RSV-In with the updated RSVs. Furthermore, the agent must check if actions need to be taken to withdraw a route that was suppressed; we discuss how an agent can do it by interacting with a standard router interface in §6.

3.6 Route Status Validation

The agent validates routes it receives through BGP against the information recorded in the repository. There are two triggers for running the validation procedure: either when an agent receives a route announced to its AS through BGP or when it receives an update from the repository.

To validate a route for prefix p, the agent must check the route against the relevant RSV-Ins records. To that end, RoST adds a new transitive extended community attribute [5, 26, 40] to BGP announcements. This new attribute contains a sequence of *RouteIDs*, where the BGP routers prepend their (current) *RouteID* to this sequence of *RouteIDs*, before forwarding an announcement. By doing so, agents that receive an announcement can validate whether the announced route is fresh, by comparing the *RouteIDs* in the transitive attribute against the corresponding *RouteIDs* that were retrieved from the repository in the relevant RSV-Ins.

For example, consider an origin AS w that announces to neighboring AS x the AS-Path for prefix p for the first time during batch number n. AS w would add to the announcement the RoST's transitive attribute containing a single *RouteID*, $(n,1)$. Assume that AS x decides to announce this route to its neighbor, AS y, during batch number n', and that AS x has made two other changes to the route to prefix p during batch n'. Thus, AS x would prepend $RouteID = (n',3)$ to the proposed transitive attribute, i.e., the transitive attribute on the announcement sent from AS x to AS y would be: $(n',3),(n,1)$.

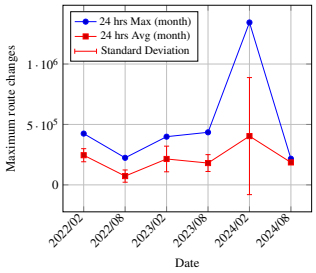
To validate the route for the prefix p against its RSV, the agent checks the repository's data about the status of the *RouteIDs* in p's BGP announcement's attributes. If at least one of the interfaces along the AS-Path of the route is not being tracked, the agent subscribes to updates from the repository to make sure it receives the records for all *RouteIDs*. If any of those routes' status is withdrawn, or if there was a later *RouteID* recorded for the same prefix (i.e., with a higher BatchID or the same BatchID but a higher PathID), then it considers that route withdrawn.

In Figure 4, we present route status validation for one route of AS 400, where the prefix is 1.1/16 and the AS-Path is 300-200-100. In the figure, the route withdrawal issued by AS 200 in batch 24 was dropped by AS 300 due to misconfiguration. AS 400 will be able to learn that AS 200 has withdrawn the routes since the status attribute in the RSV entry in RSV-In$_{(200,300)}$ has been set to false.

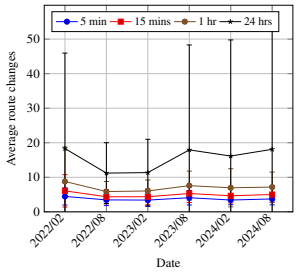
4 Evaluation Overhead

In this section, we evaluate RoST's overhead in terms of storage, communication, and computation. We use the Internet topology and traces from BGP routers to show that it is feasible to deploy RoST in today's Internet, considering the number of ASes, their announcements, and route update rate.

Methodology. To ensure a comprehensive and realistic evaluation of RoST, we use real-world BGP routing data collected across six distinct months: February and August of 2022, 2023, and 2024. We analyze data from Routing Information Base (RIB) snapshots and BGP update messages provided by RIPE NCC's Routing Information Service (RIPE RIS) [34]. RIPE RIS has multiple vantage points on the Internet that have peering links to multiple ASes and provide the full feed of their BGP traces and RIBs. We utilize RIB snapshots (generated every 8 hours) and traces of BGP



(a) Prefixes with the most daily route changes.



(b) Average over different time intervals.

Figure 5: Comparison of max and average route changes aggregated across the RIPE-RIS vantage points.

	Vantage Point	#Prefixes	RSV-In #Entries
IPv4	AS 51185	991,499	3,578,688
IPv6	AS 44085	220,074	1,078,736

Table 1: AS 51185 and AS 44085 are the RIPE VPs with the maximum number of prefixes connectivity for IPV4 and IPv6 respectively. We show the number of RSV-In entries required for each of them.

updates (batched every 5 minutes) from these vantage points over the data collection month. We collected data from 55 vantage points and processed around 10^{10} BGP updates.

4.1 Storage

Each agent has two storage-related functionalities: (1) storing information in RSV-Out vectors and (2) storing the RSV-In entries from other ASes. The repository stores the full RSV-Out vectors from all ASes.

Storage Overhead					
	Interfaces	**#ASes**	**%**	**RSV-Out**	**RSV-In**
Agent	1-10 (avg 2.14)	68,888	89.11%	22 MiB	65 MiB
	11-50 (avg 22.27)	6,707	8.68%	0.23 GiB	
	51-200 (avg 76.23)	1,607	2.08%	0.80 GiB	
	201-639 (avg 257.95)	108	0.14%	2.7 GiB	
Repository	496,910	118,031	100%	5.1 TiB	3 TiB

Table 2: Storage overhead of RoST with respect to the number of interfaces each AS has. RSV-Out/RSV-In are calculated for the average number of interfaces each group.

4.1.1 Agent Storage

In RSV-Out, every entry for an IPv4 IP prefix requires 89 bits[1] and every IPv6 entry requires 209 bits[2]. Based on RIPE RIS's traces, we observed that the ASes that announced

[1]Prefix: 40 bits, BatchID: 32 bits, PathID: 16 bits and status: 1 bit.
[2]Prefix: 136 bits, BatchID: 32 bits, PathID: 16 bits and status: 1 bit.

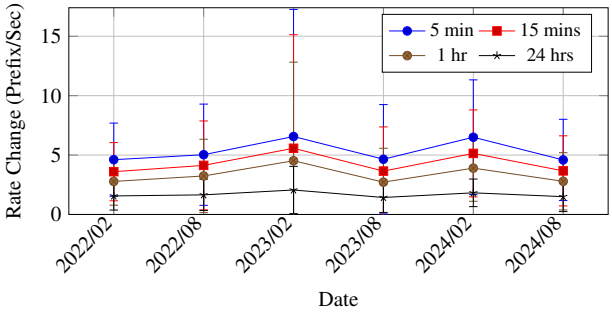


Figure 6: Average route updates per second for different batch sizes (dividing the average $\Delta RSVs$ size by the batch interval length). Across all sample dates, the average rates of changes per prefix were 5.32/s, 4.29/s, 3.33/s, and 1.67/s for the 5min, 15min, 1hr, and 24hrs intervals, respectively.

the maximum number of prefixes have announced 991,499 prefixes for IPv4 and 220,074 prefixes for IPv6, see Table 1. We follow a conservative approach and assume that RIPE RIS's traces do not contain all prefixes, since some prefixes announced to each neighbor might not always be visible to RIPE RIS's vantage points. Hence, we round these numbers to 1M IPv4 prefixes and 250K IPv6 prefixes. Furthermore, we assume a worst-case scenario, in which every RSV-Out has entries for *all* of these prefixes. Thus, by multiplying *entry size* $\times$ *#prefixes*, we estimate that the size of a single RSV-Out is 16.83MiB (10.6MiB for IPv4 plus 6.23 MiB for IPv6).

Each agent creates an RSV-Out for every neighboring AS. Based on the RIPE RIS traces, we found that the average number of interfaces per AS is 6.43 and that 89.11% of the ASes have at most 10 interfaces, see Table 2. Based on the aforementioned size of a single (worst-case scenario) RSV-Out, we get that for the average AS the storage size is 106.76MiB (16.84MiB $\times$ 6.43) and for ASes with 10 interfaces, the storage size is 168.39MiB. For ASes with the maximum number of interfaces observed, i.e., 639 interfaces, the storage size is 10.51GiB.

The agent is also responsible for retrieving and maintaining RSV-In associated with the interfaces of the other ASes appearing in the "best routes" of each of its prefixes. To quantify the expected overhead, we count for each of the 991,499 IPv4 prefixes and 220,074 IPv6 prefixes the number of interfaces based on their AS-paths. We observe that in such a worst-case scenario, any agent would need to process 3,578,688 entries for IPv4 and 1,078,736 entries for IPv6. By multiplying each of them with their corresponding entry size (see RSV-Out entry size for IPv4 and IPv6, 89 and 209 bits respectively, see above), we get 65MiB of storage, which can easily fit in a commodity agent machine's memory; see Table 2 (RSV-In column). Note that this number is identical to all agents because it does not depend on the number of interfaces.

4.1.2 Repository Storage

There are two contributing components to the repository's storage requirement: (1) storing all RSVs from each AS in the network and (2) storing the prefixes that an AS validator is interested in receiving route updates from for each of the interfaces. We aggregate the cost for each AS given the number of interfaces there are on the Internet, and assuming the highest-cost scenario in which every AS announces a route to every prefix announced on the Internet to each of its neighbors. We conclude that the repository would need to store 8.1 TiB, which is feasible even under the demanding assumption above.

4.2 Communication

We quantify the increase in BGP update length due to adding RoST's identifying tags (encoded as transitive BGP attributes) and the bandwidth consumption when ASes exchange ΔRSVs with the repository.

BGP update length overhead. Each BGP announcement in our protocol must include a new transitive attribute, the *RouteIDs*. This attribute slightly increases the size of BGP messages. To quantify this increase, we analyzed the routing table from a RIPE-RIS route collector (rrc00) using the BGPdump tool on August 1st, 2024. The results show that the average AS-path length for IPv4 is 3.86 hops (when excluding AS path prepending). For each hop on the route, RoST encodes its identifier as a 7B attribute (1B for the length and 6B for BatchID and PathID). This 27B increase represents a modest increase compared to the average BGP update length, especially considering that a BGP update message is allowed up to a maximum size of 4096B [37].

Bandwidth overhead. Agents generate ΔRSVs at each batch containing route updates, and agents at other ASes periodically fetch and validate announcements they receive against these vectors. To evaluate RoST's bandwidth overhead, we measure the number of prefixes that experienced route changes at different batch intervals (5 minutes, 15 minutes, 1 hour, and 24 hours). The shorter the batch interval, the faster RoST would be able to react to withdrawal suppression, but the greater the bandwidth overhead. This is because a route update requires a single ΔRSV entry per batch, regardless of the number of updates within that batch and whether it was due to withdrawal or path modification.

We use the route updates observed from RIPE-RIS vantage points through six different months February and August 2022, 2023 and 2024. For each vantage point, we calculate the number of prefixes that experienced route changes within the different batch intervals we consider. Figure 6 illustrates the frequency of route changes; we use this data to calculate the bandwidth cost of sending and receiving updates from the repository.

4.2.1 Agent Communication

Bandwidth overhead for submitting ΔRSVs. Each AS periodically publishes ΔRSV to the repository. As shown in Figure 6, ΔRSV should reflect average changes between 1.67 to 5.32 prefixes per second, depending on the interval length. The bandwidth overhead for a specific agent is also proportional to the number of interfaces its AS has, since any route update requires updating the ΔRSV for each interface. In Table 3 we can see the average bandwidth for each category of ASes and each different batch. Even for ASes in the category with the maximum number of interfaces (201-639), the average bandwidth consumption is very low, at only 122.13 Kbps even when batch interval is 5 minutes.

Bandwidth overhead for fetching ΔRSVs from the repository. A RoST agent synchronizes with the repository to obtain the latest route updates (ΔRSVs). Agents only request updates for active routes within their ASes to minimize bandwidth costs. The average cost per agent is negligible 0.21 Kbps for 5 minute intervals (see Table 3). To derive the repository's cost, we sum across all ASes and get 25.12 Mbps which are more than reasonable costs for a regular server with good Internet connection and negligible compared to the overall response.

The response bandwidth is more significant since it includes a Merkle tree inclusion proof for each updated entry. Each entry contains 89 bits for the RSV regular entry, plus an additional 640B for the Merkle proof, assuming SHA-256 as the hash function, and a tree with depth 20 to support the 1M IP prefixes announced on the Internet. To estimate the number of entries of a response, we refer again to the results in Figure 6, which represent the average number of prefix changes per second.

It is important to note that while these figures represent the prefixes that experienced route changes, the actual number of responses sent by the repository depends on how many interfaces have reported these changes. In the worst-case scenario, all interfaces along the path are impacted and report the changes. As mentioned earlier, on average, non-prepending paths have a length of 3.86 ASes. So the average bandwidth under these assumptions will be around 14KB/s (*route change/sec* $\times$ *path length* $\times$ *entry size*) for 5 minutes of batch interval, which, as explained for the ΔRSV-Out bandwidth, is acceptable for most machines. The overall response cost for the repository, considering a total of 100K ASes across the entire Internet, is between 7.9-12.63 Gbps (see Table 3 for exact results). We expect overhead to be much less than this, however, modern server and network hardware (e.g., Cisco [12]) can support such data transfer rates.

Bandwidth Overhead												
	Interf.	ASes	5min			15min			1hr			
			ΔRSV-Out	ΔRequests	ΔRSV-In	ΔRSV-Out	ΔRequests	ΔRSV-In	ΔRSV-Out	ΔRequests	ΔRSV-In	
Agent	1-10	89.11%	1.01 Kbps	0.21 Kbps	106.97 Kbps	0.82 Kbps	0.17 Kbps	86.26 Kbps	0.63 Kbps	0.13 Kbps	66.96 Kbps	
	11-50	8.68%	10.54 Kbps			8.50 Kbps			6.60 Kbps			
	51-200	2.08%	36.09 Kbps			29.11 Kbps			22.59 Kbps			
	201-639	0.14%	122.13 Kbps			98.49 Kbps			76.45 Kbps			
Repository	496,910	118,031	235.28 Mbps	25.12 Mbps	12.63 Gbps	189.73 Mbps	20.25 Mbps	10.18 Gbps	147.27 Mbps	15.72 Mbps	7.90 Gbps	

Table 3: RoST bandwidth, showing 5-minutes, 15-minutes, and 1-hour intervals.

4.3 Computation

We next analyze the overhead associated with processing ΔRSVs at the agent and the repository. The steps that dominate the computational costs are: agents generating ΔRSV-Outs (§4.3.1), repository handling updates and requests (§4.3.2), and ASes handling ΔRSV-Ins from the repository (§4.3.3).

4.3.1 ASes Uploading Changes to the Repository

Generating a Δ vector involves three main operations:

1. Selecting entries updated in current batch: This can be performed in advance as entries are updated, incurring negligible computational overhead.

2. Calculating the Merkle root: A full RSV contains 1M entries per interface, which results in around 2M hashes per interface.

3. A signature that authenticates the Merkle root with the version and the interface.

The overall average hash rate for agents with the largest number of interfaces sending updates every 5 minutes, results in less than 1M hash/s (859,833-for all numbers, see Table 4; ΔRSV-Out columns). A MacBook Pro with a 3.2 GHz M1 processor can generate *3 million SHA-256 hashes per second* per core (the computation is also parallelizable with multiple cores). Thus, the computational overhead for the root calculation should be reasonable for agent machines. A signature authentication needs to be made for every ΔRSV-Out sent. Assuming the worst-case scenario in which all interfaces were involved in at least one route change at each batch, the number of signatures that need to be made equals the total number of interfaces. Majority of ASes have between 0-10 interfaces (with an average of 2). Assuming that they refresh the repository every 5 minutes, this would result in ≈ 2 signatures every 5 minutes. The same MacBoook used for hash/s benchmarking is capable of performing 100 RSA signatures per second, which would be enough even for AS with higher number of interfaces (639) refreshing every 5 minutes.

4.3.2 Repository Handling Updates and Requests

The repository first validates and stores each agent's updates. Since messages are sent using a TLS session, the validation process is minimal. Updates are stored by the repository which requires targeted reads and writes in the DB. We know from the bandwidth evaluation that if all 118,031 agents are sending updates every 5 minutes, this results in 235Mbps (i.e., 29MBps as noted in Table 3). The majority of common disks can read and write at a speed of 300 - 500 MBps, thus, this adds zero I/O latency. The repository also needs to generate ΔRSV-Ins. In this step, the repository needs to generate an inclusion proof for each of the entries that had a route change. If all agents are being updated every 5 minutes (21 entries will need to be generated per second per agent, *route change/sec* (Figure 6)×*3.86 avg path length*), then a total of ≈2.4M entries per second will need to be processed. Each entry requires a Merkle proof containing ≈20 entry hashes (Figure 6). In total, we get a total of 48M h/s. These costs can be improved by caching hashes of already computed entries, but even without such improvement, the overhead is feasible. (E.g, using 18 cores of CPUs, ≈ 54M h/s, at its full computational power.)

4.3.3 ASes Handling New Updates from the Repository

The repository pushes new entry updates in the form of ΔRSV-Ins to agents, and agents need to process them. The operations that dominate this process are the verification of authenticity and integrity of the data. The agents should first verify the signature (Merkle root+version+interface). On average, as explained above, if the agent gets updated every 5 minutes, then 21 entries are received per second. Assuming the worst case where each entry belongs to a different interface, this requires 22 verifications/s (Mac M1 Pro (1 core) can compute ≈ 25K verifications/s for RSA). The agent also needs to verify all entries (21 for the same scenario) against the set of hashes included in the Merkle proof, which requires ≈20 hashes per entry; this results in 410 h/s, which remains minimal. For detailed numbers for different intervals, see the agents' ΔRSV-In column Table 4.

Computational Overhead								
			5min		15min		1hr	
	Interf.	ASes	ΔRSV-Out	ΔRSV-In	ΔRSV-Out	ΔRSV-In	ΔRSV-Out	ΔRSV-In
Agent	1-10	89.11%	7,133 h/s + 2 sig/5min	410 h/s + 21 vfy/s	2,377 h/s + 2 sig/15min	331 h/s + 17 vfy/s	594 h/s + 2 sig/1hr	257 h/s + 13 vfy/s
	11-50	8.68%	74,233 h/s + 22 sig/5min		24,744 h/s+ 22 sig/15min		6,186 h/s + 22 sig/1hr	
	51-200	2.08%	254,100 h/s + 76 sig/5min		84,700 h/s + 76 sig/15min		21,175 h/s + 76 sig/1hr	
	201-639	0.14%	859,833 h/s + 258 sig/5min		286,611 h/s + 258 sig/15min		71,653 h/s + 258 sig/1hr	
Repository	496,910	118,031	0 h/s + 0 sig/5min	48,475,803 h/s + 0 vfy/s	0 h/s + 0 sig/15min	39,090,450 h/s + 0 vfy/s	0 h/s + 0 sig/1hr	30,342,937 h/s + 0 vfy/s

Table 4: RoST Computational Overhead, showing 5-minute, 15-minute, and 1-hour intervals. MacBook Pro with a 3.2 GHz M1 processor can generate *3 Million SHA-256 h/s, 25K RSA-vfy/s and 100 RSA-sig/s* per core. *Legend: h/s* stands for hashes per second, *sig* for signature, and *vfy* for verification.

4.4 Comparison to Key Rollover

Key Rollover operates in a different fashion than RoST. It relies on BGPsec and rotates its signing key often, making sure zombie routes become invalid. However, this procedure requires refreshing *all* routes an AS advertises since they were all signed by the BGPsec signing key. Thus, if key rollover is ubiquitously adopted, all routes on the Internet would need to be refreshed often . Each provider, for example, that announces routes for all prefixes to its customers, would need to refresh about 1.2M routes per customer, resulting in major computational and communication costs. RoST, in comparison, makes routing status transparent and available to agents in order to detect zombie routes (rather than proactively periodically revoking all routes). Thus, its overhead is much smaller. It only requires 258 signatures (on average) for the ASes with the higher number of interfaces when its agent submits a new RSV-Out (Table 2). Therefore, the agent can refresh its routing data at the repository at a much higher rate than key rollover may be invoked, which results in a shorter withdrawal suppression exposure interval.

5 Benefits Under Partial Adoption

Deploying BGP mechanisms takes effort and usually happens gradually over time. Thus, it is imperative to propose mechanisms that are simple and backward compatible, but also provide benefits even under partial adoption.

RoST is designed with this mindset, providing a direct incentive for adopting ASes. An AS that adopts RoST lets other adopters detect when its own announcements are suppressed, which helps the adopter ensure its routing preferences and business decisions are executed. Though without full adoption, RoST cannot guarantee detection of all withdrawal suppressions, its detection rate increases with the adoption rate.

To illustrate RoST's benefits, consider the scenario depicted in Figure 2, but assume that initially, only AS 4 adopts RoST. First, observe that AS 2 is now incentivized to adopt RoST because it would allow AS 4 to detect that AS 6 continues to attract traffic designated to AS 2 using an outdated route, which AS 2 wants to prevent. Second, once AS 2 adopts

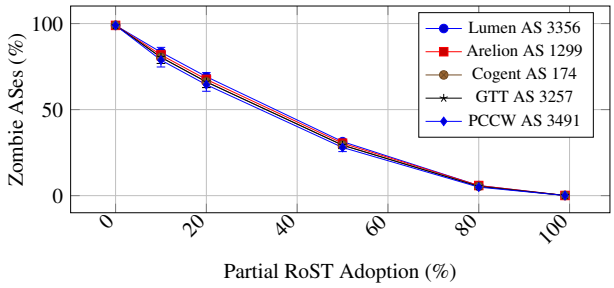


Figure 7: Impact of RoST in partial adoption on all Internet ASes when Tier-1 ASes suppress withdrawals. (Error bars mark standard deviation.)

RoST, AS 4 can detect this withdrawal suppression even though ASes 2 and 4 are the only adopters.

5.1 Evaluation

To evaluate the impact of partial adoption, we used the BGPy simulator [16], which allows simulating routes on the Internet's AS-level topology. We use a snapshot of the Internet topology from January 2025 from CAIDA [9]. We simulated the following scenario: a prefix is announced by a RoST-adopting AS, which then explicitly withdraws that announcement, while some Tier-1 AS suppresses the withdrawal. We then count the number of ASes that kept the withdrawn route active (i.e., use a zombie route). We make this measurement under different RoST-adoption rates across all Internet ASes, and assign different Tier-1 ASes the suppressor role. We use Tier-1s as suppressors in our simulations since they have the most customers, and therefore, intuitively, they should have the highest impact.

Figure 7 illustrate the impact of adopting RoST on other ASes on the Internet. We repeat the simulation for five different Tier 1 ASes as suppressors, where the adopting ASes are selected uniformly at random for each adoption rate and execution. Each data point in the graph is the average of 1,000 executions, each time with a different origin (error bars mark the 95% confidence interval). The graph shows the number of zombie ASes (i.e., ASes choosing the withdrawn route, y-axis)

monotonically decreases with the rate of adopters (x-axis). Moreover, this benefit shows as soon as ASes begin adopting (in contrast to some interdomain routing mechanisms that require near-ubiquitous adoption to provide benefits like BGPsec or key rollover that is built on top of it). Lastly, we notice that the number of non-zombie ASes is greater than the number of adopters, which illustrates the collateral benefit of adopting RoST. Namely, by adopting RoST and filtering zombie routes, the AS may also indirectly protect others since it would not announce the withdrawn route to them.

6 Compatibility with BGP Routers

We now describe how to support RoST on today's routers, without any modification to router software or hardware. Specifically, we describe how the RoST agent of an AS (external to the AS's routers, but with credentials to configure them) can: (1) attach the *RouteIDs* to announcements that the AS exports, and (2) filter withdrawn routes. We illustrate performing these operations on Cisco routers using their standard command interface, but this approach is compatible with devices from other common vendors (e.g., Juniper). Specifically, support for the BGP transitive extended community attribute is widespread [17] and includes the most common routers (such as Cisco [1] and Juniper [22] routers). A library such as Netmiko [8] allows automating interaction with routers for applying these commands. Of course, a complete implementation would need to provide fault tolerance, handling cases where an agent or the repositories fail, e.g., by defaulting to the standard BGP behavior.

6.1 Attaching *RouteIDs*

The agent needs to verify the *RouteIDs* of every BGP announcement, and if valid, the agent adds an additional *RouteID* to the community attribute before the announcements are propagated. To that end, we configure the BGP router to send BGP event logs to the agent:

```
logging host <agent-ip>
```

These logs are parsed by the agent to detect when new BGP routes are announced. To ensure that no routes are automatically advertised until the agent completes the validation and updating of the *RouteIDs*, we configure the router to suppress all routes by default in the outbound direction [14]:

```
# Block all routes from being advertised
ip prefix-list <prefix_list_name> permit 0.0.0.0/0 le 32
route-map <route_map_name> deny
 match ip address prefix-list <prefix_list_name>
neighbor <neighbor_ip> route-map out <route_map_name>
```

The above snippet shows how to suppress all prefixes (0.0.0.0/0 le 32) from being advertised to some specific AS (neighbor_ip). To attach a *RouteID* to an exported route, first we need to allow attaching extended community to advertisements with the neighbor (if not already enabled):

```
# Configure Neighbor to Send Extended Communities
router bgp <agent_AS_number>
    neighbor <neighbor_ip> send-community extended
```

Then, we can use Netmiko to dynamically attach a *RouteID* to an advertisement via the following router command:

```
set extcommunity rt <routeid>
```

6.2 Filtering Withdrawn Routes

The other functionality that RoST requires is to withdraw routes that the agent detects as withdrawn, even though the explicit/implicit withdrawal was suppressed. Thus, the agent needs to instruct the router to remove a specific (withdrawn) route from the Local RIB, which will cause the router to choose a new best route. Cisco supports this with the following command:

```
# Defines a prefix list to allow specific IP prefixes
ip prefix-list REMOVE permit <ip-prefix>
# Remove the routes matching the REMOVE prefix list
clear ip bgp <neighbor-ip> soft in prefix-list REMOVE
```

7 Deployment Considerations

We next discuss privacy and fault tolerance challenges in deploying RoST.

7.1 Privacy

Since RoST is a transparency system, it inventively reveals information about the status of routes ASes advertise. Some ASes might be less willing to share this information, at least for some of their neighbors, although it will help ensure that their announcements don't get suppressed. For example, some ASes might not want to expose (specific) paying customers. However, we argue that RoST's impact on privacy is limited in practice. The information made transparent by RoST is on par with what is available through existing or proposed BGP mechanisms. Moreover, a significant portion of this information is already inferred and made public today, for example, by services like RIPE RIS, which we used as part of our evaluation. Furthermore, many of the relationships between ASes can be deduced (explicitly or implicitly) from other BGP-related protocols. For example, RPKI reveals ASes' prefixes (via ROAs) and customer-provider relations (via resource certificates), and the latter can also be deduced from proposed mechanisms such as Only-To-Customer (OTC) [3] and Autonomous System Provider Authorization (ASPA) [4].

That said, some ASes may still decide to withhold some of their route status information, either by only sharing some of the prefixes they can reach through a neighbor or even

completely the fact that they peer with that neighbor. Luckily, this does not impact RoST beyond protecting the withheld routes. It only means that the withholding ASes deny other ASes the opportunity to detect suppression of those routes.

7.2 Repository Faults

So far, we described RoST with one repository, creating a single point of failure. In practice, however, we envision a deployment with multiple repositories operated by independent entities. This not only improves the scalability and robustness of the system but allows for mitigating the impact of a malicious repository that selectively hides status updates for some routes (note that since the agent updates are signed, the repository cannot generate fake updates). Such a deployment may also piggyback on the existing BGP repositories, such as the RPKI repositories maintained by RIRs, to maintain RoST' information.

A simple approach for running RoST with multiple repositories is to have agents send ΔRSV-Out updates to all repositories. Agents then receive updates from multiple repositories, ensuring that even when some repositories are down (e.g., due to a major outage) or censor the status for specific routes, the agent will retrieve this information from another repository. This approach ensures all (honest) repositories will have their data without the need to synchronize repositories (and it is the sender's interest to make sure their data is stored at the repository, §5).

Facilitating efficient synchronization across the repositories could save the agent communication costs from the previous solution Such synchronization may use Byzantine Fault Tolerance (BFT) protocols to ensure that all repositories hold the same route statuses despite malicious repositories. Further, the repositories can take turns being the leader in the BFT protocol to ensure a malicious repository cannot censor route status updates [10, 23]. Lastly, BFT protocols create a certificate for the data after repositories reach consensus. This allows agents to update from one repository and check the certificate to ensure other repositories have the same updates.

8 Related Work

Significant effort has been devoted to improving the security of interdomain routing [7, 18, 20, 31, 38]. Although not all proposed mechanisms can be used immidiately with currently deployed BGP, the increasing widespread adoption of RPKI [25] is promising since RPKI is a cornerstone for other cryptography-based mechanisms. Specifically, RPKI enables origin validation [11, 19, 24, 27, 33] against prefix hijack attacks, where an attacker advertises a false origin for a prefix to hijack traffic intended for the real origin. BGPsec [28] builds on RPKI to add signatures on route advertisements, providing route authentication that protects against bogus paths advertised with the correct origin. Recently, the IETF published other drafts for mechanisms that aim to increase routing reliability and security. For example, the Only-To-Customer (OTC) [3] transitive BGP path attribute defines a mechanism against route leaks, and the Autonomous System Provider Authorization (ASPA) [4] extends RPKI's origin validation to check ASes beyond the first hop [13].

However, none of the above security mechanisms addresses withdrawal suppression. Approaches that reduce router misconfigurations and software bugs, e.g., through better policy specification, router design, and troubleshooting tools [21, 29], can help reduce the chance of accidental withdrawal suppression, but they require adoption at the suppressing AS rather than the victim. The proposed solution to withdrawal suppression is key rollover [39, 42], which requires route authentication (e.g., via BGPsec). It incurs high operational costs, as well as significant computation and communication overheads from rotating keys often, as discussed earlier in §1. As a result, key rollover cannot operate at short timescales to quickly mitigate withdrawal suppression. In contrast, RoST addresses withdraw suppression by leveraging RPKI to have ASes sign route status data and publish it, requiring no change to RPKI and not relying on a route authentication mechanism. Furthermore, RoST is compatible with BGP and requires no change to BGP routers.

9 Conclusion

In this paper, we presented RoST, an efficient and practical protocol designed to defend against withdrawal suppression. RoST can integrate with current routing hardware by using standard APIs and is thus readily deployable. Further, we showed through simulations that RoST provides benefits even to early adopters. We evaluated RoST's overhead based on real-world routing data and showed RoST incurs reasonable storage, compute, and network costs and can thus mitigate withdrawal suppression at much shorter timescales compared to the key rollover alternative (which also relies on BGPsec).

10 Acknowledgments

We thank our shepherd, Oliver Hohlfeld, and the anonymous reviewers for their thoughtful comments, suggestions, and feedback. This work was supported by grant no. 2022701 from the United States-Israel Binational Science Foundation (BSF), by the Research Authority Fund of the College of Management Academic Studies, Rishon LeZion, Israel, by grants 2247810, 2410268, and 2149765 of the National Science Foundation, and by Dr. Herzberg's endowment from Comcast. The opinions expressed in the paper are those of the researchers and not of their institutions or funding sources.

References

[1] Cisco ios bgp command reference. https://www.cisco.com/c/en/us/td/docs/ios/12_2s/feature/guide/fsnextcl.html.

[2] RPKI Monitor - Route Origin Validation. https://rpki-monitor.antd.nist.gov/ROV. Accessed: date.

[3] Alexander Azimov, Eugene Bogomazov, Randy Bush, Keyur Patel, and Kotikalapudi Sriram. Route Leak Prevention and Detection Using Roles in UPDATE and OPEN Messages. RFC 9234, May 2022.

[4] Alexander Azimov, Eugene Uskov, Randy Bush, Keyur Patel, Job Snijders, and Russ Housley. A Profile for Autonomous System Provider Authorization. Internet-Draft draft-ietf-sidrops-aspa-profile-07, Internet Engineering Task Force, January 2022. Work in Progress.

[5] Tony J. Bates and Enke Chen. An Application of the BGP Community Attribute in Multi-home Routing. RFC 1998, August 1996.

[6] Steven Bellovin, Randy Bush, and David Ward. Security Requirements for BGP Path Validation. RFC 7353, August 2014.

[7] Kevin Butler, Toni R. Farley, Patrick McDaniel, and Jennifer Rexford. A survey of BGP security issues and solutions. *Proceedings of the IEEE*, 98(1):100–122, 2010.

[8] Kirk Byers. Netmiko - multi-vendor library to simplify paramiko ssh connections to network devices, 2024. https://github.com/ktbyers/netmiko.

[9] CAIDA. The CAIDA AS Relationships Dataset, 2025-01-05. https://www.caida.org/catalog/datasets/as-relationships/.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[11] Taejoong Chung, Emile Aben, Tim Bruijnzeels, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Roland van Rijswijk-Deij, John Rula, and Nick Sullivan. RPKI is Coming of Age: A Longitudinal Study of RPKI Deployment and Invalid Route Origins. In *Proc. of IMC*. ACM, 2019.

[12] Cisco Systems. Unified computing system adapters white paper. https://www.cisco.com/c/en/us/products/collateral/interfaces-modules/unified-computing-system-adapters/whitepaper-c11-741550.html, 2024. Accessed: 2024-09-12.

[13] Avichai Cohen, Yossi Gilad, Amir Herzberg, and Michael Schapira. Jumpstarting BGP security with path-end validation. In *Proc. of ACM SIGCOMM*, pages 342–355. ACM, 2016.

[14] Extreme Networks. Extreme networks article detail. https://extreme-networks.my.site.com/ExtrArticleDetail?an=000082826, 2024. Accessed: 2024-09-12.

[15] Romain Fontugne, Esteban Bautista, Colin Petrie, Yutaro Nomura, Patrice Abry, Paulo Gonçalves, Kensuke Fukuda, and Emile Aben. BGP zombies: An analysis of beacons stuck routes. In *Passive and Active Measurement: 20th International Conference, PAM 2019, Puerto Varas, Chile, March 27–29, 2019, Proceedings 20*, pages 197–209. Springer, 2019.

[16] Justin Furuness, Cameron Morris, Reynaldo Morillo, Amir Herzberg, and Bing Wang. Bgpy: The bgp python security simulator. In *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, CSET '23, page 41–56, New York, NY, USA, 2023. Association for Computing Machinery.

[17] Yossi Gilad, Tomas Hlavacek, Amir Herzberg, Michael Schapira, and Haya Shulman. Perfect is the enemy of good: Setting realistic goals for BGP security. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018*, pages 57–63, 2018.

[18] Amir Herzberg, Matthias Hollick, and Adrian Perrig. Secure Routing for Future Communication Networks (Dagstuhl Seminar 15102). *Dagstuhl Reports*, 5(3):28–40, 2015.

[19] Tomas Hlavacek, Haya Shulman, Niklas Vogel, and Michael Waidner. Keep your friends close, but your routeservers closer: Insights into rpki validation in the internet. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[20] G. Huston, M. Rossi, and G. Armitage. Securing BGP: A literature survey. *IEEE Communications Surveys & Tutorials*, 13(2):199–222, 2011.

[21] Vinit Jain and Brad Edgeworth. *Troubleshooting BGP: A Practical Guide to Understanding and Troubleshooting BGP*. Cisco Press, 2016.

[22] Juniper Networks. BGP communities and extended communities match conditions overview. https://www.juniper.net/documentation/us/en/software/junos/routing-policy/bgp/topics/concept/policy-bgp-communities-extended-commu

nities-match-conditions-overview.html, 2024. Accessed: 2024-09-12.

[23] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, pages 45–58, 2007.

[24] APNIC Labs. Rpki, 2023.

[25] Matt Lepinski and Stephen Kent. An Infrastructure to Support Secure Internet Routing. RFC 6480, February 2012.

[26] Tony Li, Ravi Chandra, and Paul S. Traina. BGP Communities Attribute. RFC 1997, August 1996.

[27] Weitong Li, Zhexiao Lin, Mohammad Ishtiaq Ashiq Khan, Emile Aben, Romain Fontugne, Amreesh Phokeer, and Taejoong Chung. RoVista: Measuring and Understanding the Route Origin Validation (ROV) in RPKI. In *Proceedings of the ACM Internet Measurement Conference (IMC'23)*, pages 1–14, Montreal, Canada, October 2023. ACM.

[28] M. Lepinski (Ed.) and K. Sriram (Ed.). BGPsec Protocol Specification. RFC 8205, sep 2017.

[29] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[30] Paweł Małachowski. Bgp zombie routes, 2020. https://es.slideshare.net/slideshow/bgp-zombie-routes/238666533.

[31] Asya Mitseva, Andriy Panchenko, and Thomas Engel. The state of affairs in BGP security: A survey of attacks and defenses. *Computer Communications*, 124:45–60, June 2018.

[32] Cameron Morris, Amir Herzberg, and Samuel Secondo. Bgp-isec: Improved security of internet routing against post-rov attacks. In *Proceedings of the 2024 Network and Distributed System Security Symposium (NDSS 2024)*, 11 2023.

[33] National Institute of Standards and Technology (NIST). NIST RPKI Monitor, version 2.0. https://rpki-monitor.antd.nist.gov/. Accessed November 2023.

[34] RIPE NCC. Routing information service (ris), 2024. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris.

[35] Porapat Ongkanchana, Romain Fontugne, Hiroshi Esaki, Job Snijders, and Emile Aben. Hunting BGP zombies in the wild. In *Proceedings of the 2021 Applied Networking Research Workshop*, ANRW '21, page 1–7, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Matthew Prince et al. August 30th 2020: Analysis of centurylink/level(3) outage, Jul 2022. https://blog.cloudflare.com/analysis-of-todays-centurylink-level-3-outage/.

[37] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[38] Muhammad S. Siddiqui, Diego Montero, Rene Serral-Gracia, Xavi Masip-Bruin, and Marcelo Yannuzzi. A survey on the recent efforts of the Internet Standardization Body for securing inter-domain routing. *Computer Networks*, 80:1–26, April 2015.

[39] Kotikalapudi Sriram and Doug Montgomery. Design Discussion and Comparison of Protection Mechanisms for Replay Attack and Withdrawal Suppression in BGPsec. Internet-Draft draft-sriram-replay-protection-design-discussion-11, Internet Engineering Task Force, October 2018. Work in Progress.

[40] Dan Tappan, Srihari R. Sangli, and Yakov Rekhter. BGP Extended Communities Attribute. RFC 4360, February 2006.

[41] Pierre-Antoine Vervier, Olivier Thonnard, and Marc Dacier. Mind Your Blocks: On the Stealthiness of Malicious BGP Hijacks. In *NDSS*, 2015.

[42] Brian Weis, Roque Gagliano, and Keyur Patel. BGPsec Router Certificate Rollover. RFC 8634, August 2019.

A RoST Pseudocode

In this section, we detail the operation of a RoST agent in pseudocode, given in Appendix A.

Algorithm 2 RoST Agent

```
1: Input:
2:     AgentAS: AS for which this agent is responsible
3:     BatchInterval: Time interval for batching updates in ΔRSV-Out
4:     R: Repository
5: State:
6:     RSV-Out, RSV-In ← ∅                                  ▷ Tracks outgoing & incoming route status per interface
7:     SubscribedInterfaces ← ∅                             ▷ Tracks subscribed interfaces for each prefix
8:     PrivateKey ← LoadPrivateKey(AgentAS)                 ▷ Private key for signing RSV-Out data
9:     BatchInterval ← Input BatchInterval
10:    RepositoryConnection ← ConnectToRepository(R)        ▷ Manages communication with repository
11:    RouterConnection ← ConnectToRouter(AgentAS)          ▷ Connection to the router of 'AgentAS'
12: procedure RoSTAgent(AgentAS, BatchInterval, R)
13:    // Set up event-driven handlers for router and repository interactions
14:    Register ProcessBGPUpdate(U) as a handler for incoming BGP updates from router   ▷ See Line 18
15:    Register ProcessRSVInUpdates(ΔRSV-In) as a handler for RSV-In updates from repository R   ▷ See Line 24
16:    Schedule periodic tasks every BatchInterval to:
17:        Generate and upload ΔRSV-Out to repository R using GenerateAndUploadRSVOut( )   ▷ See Line 30

18: procedure ProcessBGPUpdate(U)
19:    for each Neighbor AS_y of AgentAS do
20:        if IsEligibleToPropagate(AS_y, U.Prefix) then   ▷ See Line 167
21:            Status ← ValidateRouteStatus(U)   ▷ See Line 34
22:            if Status = Valid or Pending then UpdateAndPropagateBGP(U, AS_y)   ▷ Updates RSV-Out state. See Line 108
23:            else if Status = Invalid then HandleInvalidRoute(U, AS_y)   ▷ See Line 90

24: procedure ProcessRSVInUpdates(ΔRSV-In)
25:    ValidatedPrefixes ← MergeRSVIn(ΔRSV-In)   ▷ Returns validated prefixes. See Line 54
26:    for each Prefix P in ValidatedPrefixes do
27:        for each Neighbor AS_y of AgentAS do
28:            if (RIBOutEntry ← QueryRIBOut(Prefix = P, Neighbor = AS_y)) ≠ ∅ then   ▷ See Line 158
29:                if ValidateRouteStatus(RIBOutEntry) = Invalid then HandleInvalidRoute(RIBOutEntry, AS_y) ▷ See
                   Lines 34 and 90

30: procedure GenerateAndUploadRSVOut
31:    for each interface (AS_x, AS_y) in RSVOut do
32:        if (DeltaInterface ← GenerateDeltaRSV(AS_x, AS_y)) ≠ ∅ then   ▷ See Line 130
33:            UploadRSVOut(DeltaInterface)   ▷ See Line 174

34: procedure ValidateRouteStatus(Route, AS_y)
35:    Extract Prefix, AS-Path, and RouteID sequence (BatchID, PathID) from Route
36:    Status ← Valid   ▷ Initial assumption that the route is valid
37:    MissingInterface ← False   ▷ Flag to check if subscription is needed
38:    for each (AS_prev, AS_current) in AS-Path do
39:        Retrieve RSVInInterface ← RSVIn(AS_prev, AS_current)   ▷ Interface-specific RSV-In
40:        Retrieve RSVInEntry ← RSVInInterface[Prefix]   ▷ Access RSV-In entry by prefix
41:        if (AS_prev, AS_current) ∉ SubscribedInterfaces[(Prefix, AS_y)] then
42:            MissingInterface ← True   ▷ Mark that the subscription needs to be updated
43:        if RSVInEntry does not exist or BatchID > RSVInEntry.BatchID then
44:            Status ← Pending   ▷ Mark status as pending due to missing data
45:        else if BatchID < RSVInEntry.BatchID or PathID ≠ RSVInEntry.PathID or RSVInEntry.Status = Inactive then
46:            return Invalid   ▷ Inconsistent or inactive route
47:    if MissingInterface = True then
48:        // Update the subscription state for this AS_y and Prefix
49:        Replace SubscribedInterfaces[(Prefix, AS_y)] with all interfaces in AS-Path
50:        // Prepare the full subscription set for all AS_y for the Prefix
51:        FullSubscriptionSet ← ⋃_{AS_y} SubscribedInterfaces[(Prefix, AS_y)]
52:        SubscribeToRSVInUpdates(Prefix, FullSubscriptionSet)   ▷ Send a single subscription for the prefix across all AS_y
53:    return Status   ▷ Return Valid or Pending status
```

```
54: procedure MERGERSVIN(ΔRSV-In)
55:     // Step 1: Extract batch-level metadata
56:     Extract BatchCounter, MerkleRoot, Signature, (AS_prev, AS_current) from ΔRSV-In
57:     // Step 2: Retrieve Interface-Specific BatchCounter
58:     RSVInInterface ← RSVIn(AS_prev, AS_current)
59:     if RSVInInterface does not exist then
60:         Initialize RSVIn(AS_prev, AS_current) ← ∅
61:         LocalBatchCounter ← 0                                   ▷ Default value for new interface
62:     else
63:         Retrieve LocalBatchCounter ← BatchCounter stored in RSVIn(AS_prev, AS_current)
64:     // Step 3: Check BatchCounter from ΔRSV-In
65:     if BatchCounter ≤ LocalBatchCounter then
66:         Log: "Stale ΔRSV-In batch from (AS_prev, AS_current) with BatchCounter BatchCounter"
67:         Reject and exit
68:     // Step 4: Verify Batch Signature
69:     Reconstruct SignedData ← HASH(BatchCounter, MerkleRoot, (AS_prev, AS_current))
70:     Retrieve PublicKey ← RPKI[AS_prev]
71:     if not VERIFYSIGNATURE(SignedData, Signature, PublicKey) then
72:         Log: "Invalid signature for ΔRSV-In batch from (AS_prev, AS_current)"
73:         Reject and exit
74:     // Step 5: Process Individual Entries with Merkle Proofs
75:     ValidatedPrefixes ← ∅                                       ▷ Initialize list of validated prefixes
76:     for each entry E in ΔRSV-In do
77:         Extract Prefix, BatchID, PathID, Status, MerkleProof
78:         // Validate Merkle Proof for Entry
79:         if not VERIFYINCLUSIONPROOF(Prefix, BatchID, PathID, Status, MerkleProof, MerkleRoot) then
80:             Log: "Invalid Merkle proof for Prefix from (AS_prev, AS_current)"
81:             Continue                                            ▷ Reject and skip this entry
82:         // Merge Valid Entry into Local RSV-In
83:         RSVIn(AS_prev, AS_current)[Prefix] ← (BatchID, PathID, Status)
84:         Add Prefix to ValidatedPrefixes
85:         Log: "Successfully merged Prefix from (AS_prev, AS_current)"
86:     // Step 6: Update BatchCounter for the Interface
87:     Update RSVIn(AS_prev, AS_current).BatchCounter ← BatchCounter
88:     Log: "Updated BatchCounter to BatchCounter for (AS_prev, AS_current)"
89:     return ValidatedPrefixes                                    ▷ Return the list of validated prefixes

90: procedure HANDLEINVALIDROUTE(Route, AS_y)
91:     Extract Prefix and specific route attributes from Route
92:     // Step 1: Remove the route from the router's RIB for the interface with AS_y
93:     REMOVEROUTEFROMRIB(Prefix, Route, AS_y)                     ▷ Remove route for the interface with AS_y. See Line 161
94:     Log: "Removed invalid route Route for Prefix to AS_y"
95:     // Step 2: Check for alternative routes in the router's RIB for the interface with AS_y
96:     AlternativeRoutes ← QUERYALTERNATIVEROUTES(Prefix, AS_y) ▷ Query remaining routes for the interface with AS_y. See Line 164
97:     if AlternativeRoutes is empty then
98:         // Step 3: Propagate withdrawal using UpdateAndPropagateBGP
99:         WithdrawalUpdate ← Generate withdrawal update message for Prefix
100:        UPDATEANDPROPAGATEBGP(WithdrawalUpdate, AS_y)           ▷ Attach RouteID and propagate withdrawal. See Line 108
101:        Log: "Withdrawal propagated for Prefix to AS_y"
102:    else
103:        // Step 4: Select the best alternative route and announce it using UpdateAndPropagateBGP
104:        BestRoute ← SELECTBESTROUTE(AlternativeRoutes)
105:        AnnouncementUpdate ← Generate announcement update message for BestRoute
106:        UPDATEANDPROPAGATEBGP(AnnouncementUpdate, AS_y)         ▷ Attach RouteID and propagate announcement. See Line 108
107:        Log: "Announced alternative route for Prefix to AS_y"
```

```
procedure UpdateAndPropagateBGP(U, AS_y)
    Extract Prefix, AS-Path, Action (e.g., Announce or Withdraw) from U
    // Step 1: Update RSV-Out for interface with AS_y
    Retrieve RSVOutInterface ← RSVOut(AgentAS, AS_y)
    // New Prefix Entry Handling
    if Prefix ∉ RSVOutInterface then Initialize empty entry for Prefix in RSVOutInterface
    // New Batch Handling for Prefix Entry
    if RSVOutInterface.BatchCounter > RSVOutInterface[Prefix].BatchID then
        Reset RSVOutInterface[Prefix].PathID ← 0
        Update RSVOutInterface[Prefix].BatchID ← RSVOutInterface.BatchCounter
    // Update Existing Entry Based on Action
    if Action = Announce then
        Increment RSVOutInterface[Prefix].PathID
        Update RSVOutInterface[Prefix].Status ← Active
    else if Action = Withdraw then
        Update RSVOutInterface[Prefix].Status ← Inactive
    // Step 2: Attach RouteID to the BGP Update Using API
    RouteID ← (RSVOutInterface[Prefix].BatchID, RSVOutInterface[Prefix].PathID)
    AttachRouteIDToUpdate(U, RouteID)                          ▷ Router API: Attach RouteID
    // Step 3: Forward Updated BGP Message to AS_y Using API
    ForwardUpdate(U, AS_y)                                     ▷ Router API: Forward message to neighbor
    Log: "Forwarded BGP update for Prefix to AS_y with RouteID (BatchID, PathID)"

procedure GenerateDeltaRSV(AS_x, AS_y)
    Retrieve RSVOutInterface ← RSVOut(AS_x, AS_y)
    // Step 1: Identify Delta Entries for the Current Batch
    Initialize DeltaEntries ← ∅
    for each Prefix in RSVOutInterface do
        if RSVOutInterface[Prefix].BatchID = RSVOutInterface.BatchCounter then
            Add RSVOutInterface[Prefix] to DeltaEntries
    // Step 2: Build Merkle Tree for All RSV-Out Entries
    Initialize AllEntries ← All entries in RSVOutInterface
    MerkleLeaves ← Hash(AllEntries)
    // Hash each entry in AllEntries
    MerkleRoot ← BuildMerkleTree(MerkleLeaves)
    // Step 3: Sign the Merkle Root, BatchCounter, and Interface
    ToSign ← Hash(MerkleRoot, RSVOutInterface.BatchCounter, (AS_x, AS_y))
    Signature ← SignWithPrivateKey(ToSign, PrivateKey)
    // Step 4: Increment BatchCounter and Update BatchStartTime
    Increment RSVOutInterface.BatchCounter
    RSVOutInterface.BatchStartTime ← CurrentTime()
    // Step 5: Prepare and Return Delta
    if DeltaEntries ≠ ∅ then
        return {(AS_x, AS_y), RSVOutInterface.BatchCounter − 1, DeltaEntries, MerkleRoot, Signature}
    return ∅
```

Router API Module

The following functions interact directly with the router via the `RouterConnection` state. These functions handle operations such as querying or modifying the router's RIB and propagating updates to neighboring ASes. They are invoked by the `RoSTAgent` to enforce protocol logic.

procedure FORWARDUPDATE(U, AS_y)
    Use *RouterConnection* to send U to AS_y
    Log: "Forwarded BGP update for $U.Prefix$ to AS_y"

procedure ATTACHROUTEIDTOUPDATE(U, *RouteID*)
    Use *RouterConnection* to append *RouteID* to U
    Log: "Attached RouteID ($RouteID.BatchID, RouteID.PathID$) to BGP update for $U.Prefix$"

procedure QUERYRIBOUT($Prefix, AS_y$)
    Query *RouterConnection* for RIB-Out entry matching $Prefix$ for neighbor AS_y
    return Matching entry or $\emptyset$ if not found

procedure REMOVEROUTEFROMRIB($Prefix$, $Route$, AS_y)
    Use *RouterConnection* to remove $Route$ for $Prefix$ on the interface with AS_y
    Log: "Route $Route$ for $Prefix$ removed from the interface with AS_y"

procedure QUERYALTERNATIVEROUTES($Prefix$, AS_y)
    Use *RouterConnection* to query all remaining routes for $Prefix$ on the interface with AS_y
    return List of alternative routes for $Prefix$ on the interface with AS_y or $\emptyset$ if none found

procedure ISELIGIBLETOPROPAGATE($AS_y, Prefix$)
    Query *RouterConnection* for policy decision on propagating $Prefix$ to AS_y
    return True if the policy allows propagation, **False** otherwise

Interaction with the repository The following functions interact directly with the repository via the `RepositoryConnection` state. These functions handle operations such as subscribing to updates and uploading data to the repository. They are invoked by the `RoSTAgent` to implement repository-specific logic.

procedure SUBSCRIBETORSVINUPDATES($Prefix$, $InterfaceSet$)
    Use *RepositoryConnection* to send a subscription request for:
        $Prefix$ and all interfaces in $InterfaceSet$
    Log: "Subscribed to RSV-In updates for $Prefix$ with interfaces $InterfaceSet$"

procedure UPLOADRSVOUT($DeltaInterface$)
    Use *RepositoryConnection* to upload $DeltaInterface$

ValidaTor: Domain Validation over Tor

Jens Frieß[§‡], Haya Schulmann[§†], and Michael Waidner[§‡]

[§]National Research Center for Applied Cybersecurity ATHENE
[‡]Technische Universität Darmstadt [†]Goethe-Universität Frankfurt

Abstract

Domain Validation (DV) is the primary method used by Certificate Authorities (CAs) to confirm administrative control over a domain before issuing digital certificates. Despite its widespread use, DV is vulnerable to various attacks, prompting the adoption of multiple vantage points to enhance security, such as the state of the art DV mechanism supported by Let's Encrypt. However, even distributed static vantage points remain susceptible to targeted attacks. In this paper we introduce ValidaTor, an HTTP-based domain validation system that leverages the Tor network to create a distributed and *unpredictable* set of validators. By utilizing Tor's exit nodes, ValidaTor significantly increases the pool of available validators, providing high path diversity and resilience against strong adversaries.

Our empirical evaluations demonstrate that ValidaTor can achieve the validation throughput of a commercial CA and has the potential to scale to a validation volume comparable to Let's Encrypt, while using minimal dedicated infrastructure and only a small fraction (~0.1%) of Tor's remaining available bandwidth. While unpredictable selection of validators makes ValidaTor fully resistant to targeted attacks on validators, we also show the use of Tor nodes improves path diversity and thereby the resilience of DV to subversion by well-positioned ASes, reducing the number of Autonomous Systems (ASes) capable of issuing fraudulent certificates by up to 27% compared to Let's Encrypt. Lastly, we show that the chance of subversion by malicious, colluding exit nodes is negligible ($\leq 1\%$ even with a quarter of existing exit nodes).

We make the code of ValidaTor as well as the datasets and measurements publicly available for use, reproduction, and future research.[1]

1 Introduction

Digital public key certificates associated with Internet domains are fundamental to Internet security. These certificates include crucial information such as the domain name and the public validation key corresponding to the domain owner's private key, ensuring the authenticity of communicating parties and enabling secure connections. The correct issuance of certificates is critical, as any compromise could allow an adversary to impersonate a legitimate service and intercept sensitive communications.

Certificate Authorities (CAs) use Domain Validation (DV) to verify domain ownership before issuing a certificate. This process typically involves generating a random challenge-response pair, and having the domain owner place the response at the challenged location under the target domain, e.g., in a DNS TXT record or an HTTP-hosted `.txt` file. The CA validates the domain by querying for the challenge and checking that the obtained response matches the challenge. If the CA receives the correct response, the domain owner has successfully demonstrated control over the domain, and the CA issues a signed certificate for the domain with the public key provided by the domain owner.

Although the Public Key Infrastructure (PKI) security relies on the correctness of DV for checking control over domains, the DV procedure itself remains vulnerable. The challenge-response process of DV is conducted over unencrypted and unauthenticated channels and heavily relies on the Domain Name System (DNS) RFC1035 and the Border Gateway Protocol (BGP) RFC4271. Both are largely unprotected and vulnerable to hijacking attacks [1, 2], and adversaries have launched such attacks, e.g., to steal cryptocurrency [3–5].

The attacks and the importance of certificate security triggered research efforts to identify vulnerabilities in the DV procedure and to develop a more secure design. Initially, vanilla DV was performed with a single request to the target domain. However, it was shown that an adversary could manipulate the challenge request of a CA, causing it to be executed against servers that the adversary controls [6]. As a result, the largest CA, Let's Encrypt, developed MultiVA, which uses additional vantage points [7]. During the validation process a number of vantage points independently send a challenge to the target domain. MultiVA became an official market standard with Automated Certificate Management Environment (ACME) in

[1] https://github.com/jenfrie/tova

RFC8555. The use of multiple vantage points aims to increase security despite being conducted over unprotected channels since it is assumed that a realistic adversary cannot control multiple vantage points and it is more difficult to attack them concurrently than a single vantage point.

The MultiVA was also extensively validated in the research community [7, 8]. Simulations have shown that the average resilience of a customer domain against issuance of fraudulent certificates is 88.6% [9], reflecting the percentage of adversarial ASes that cannot intercept communication via same-prefix BGP hiijacks to a sufficient number of vantage points to complete DV successfully. However, the analysis of [9] excluded a number of attack vectors, such as sub-prefix hijacks, as well as attacks against the DNS resolvers. Another important aspect that was not considered in these simulations is the dependency between DNS and the inter-domain routing infrastructures, which further increases the attack surface: DNS cache poisoning can be leveraged to facilitate BGP hijacks [10]. Indeed, [11] exploited a property in the software of many popular DNS resolver implementations to force all the vantage points to query one specific nameserver of the adversary's choice. This reduced the distributed defense to just one weak link, the nameserver. By hijacking the prefix of the nameserver, the adversary could intercept communication with all the vantage points. This shows that fixed vantage points, i.e., always at the same IP address or on the same IP prefix, allow the adversary to launch targeted attack against them in advance, before initiating the domain validation process. Both these issues were exploited in [11] to subvert DV from multiple vantage points, indicating that merely increasing the number of vantage points does not suffice to fully prevent targeted attacks.

A resilience of 88.6% offered by MultiVA [9] against interceptions for a critical building block like PKI is arguably not sufficient. Ideally we should achieve a 100% resilient PKI. Furthermore, 88.6% resilience considers only BGP hijacks, but does not account for other common attack vectors targeting the validators and connected infrastructure, e.g., DNS cache poisoning or infiltration of on-path routers.

The main problem, as pointed out by [11], is that current vantage points are static. In addition to being distributed, the vantage points' selection needs to be *unpredictable*. If the adversary does not know the addresses of the vantage points involved in a given validation, it cannot attack them.

Therefore, to improve security, the validators should be selected at random from the set of vantage points. The pool of vantage points should be large enough that it is infeasible to hijack any significant fraction. With a dedicated deployment, such as that of Let's Encrypt, generating such a large pool is not practical since it would require setting up a significant amount of geographically distributed infrastructure, prohibitively increasing the costs. In fact, scalability and cost are a major inherent limitation of any dedicated infrastructure.

Contributions. Motivated by the limitations in the existing domain validation mechanisms, we aim to design a system that resolves them. We develop a novel approach for distributed domain validation using Tor [12], dubbed ValidaTor. With over 2,200 Tor exit nodes, located across 280 unique BGP origins and 1,221 unique IP addresses, our design substantially increases the scale of the distributed infrastructure available to DV. This is a huge improvement compared to the 7 BGP origins currently used by Let's Encrypt. It allows a randomized selection of a configurable number of validators, effectively preventing BGP hijack attacks for subverting validation. We perform extensive evaluations to demonstrate the security and performance of our proposal based on real-world measurements and simulations. Our contributions:

- Conceptually, we offer a novel use of the Tor network to address practical limitations in DV. Our design and implementation of ValidaTor resolves the scalability of massively distributed DV and enhances its resilience.
- We propose a path overlap metric to capture the diversity of paths taken from validators to target domains through the BGP network, allowing for easy comparison of DV deployments with regard to threats from well-positioned ASes.
- Our experimental evaluations show that ValidaTor decreases average path overlap by ~50% compared to Let's Encrypt's MultiVA, translating to a reduction of 21% with 3 validators (up to 27% with 7 validators) in potential fraudulent certificates from well-positioned ASes. Combined with the resistance to BGP hijacks through unpredictable selection of validators, this significantly hardens DV against subversion.
- We show that with our selection method the chance of an attacker controlling validation with malicious Tor nodes is negligible, amounting to less than 1% even when controlling a quarter of all existing Tor nodes.
- We demonstrate a validation rate of up to 11.9 validations per second with 5 validators and estimate that, even at the scale of the entire Web PKI, Tor's remaining bandwidth would only be taxed by 0.11%. The median duration for a single validation in this case is ~2 seconds, with at least 95% of all validations under 6 seconds. This is comparable with the use of the certbot ACME client.
- We design ValidaTor so that it enables easy adoption and requires no changes to the existing infrastructure of the CAs, which is critical to motivate deployment and avoid misconfiguration. We make our prototype and the collected data available for public use.

Ethical considerations. We ensure that the design of ValidaTor is ethical, does not burden the Tor network and does not interfere with the connections of other Tor clients. As detailed in Section 6.4, the expected network load on Tor from a full switch to ValidaTor by the entire Web, is 635 MBit/s, around 0.1% of the remaining available bandwidth. We also take extensive measures to ensure that our experiments follow the ethical guidelines for network measurements [13, 14].

Organization. We review Related Work in Section 2 and provide an overview of Tor network in Section 4. The design

and implementation of ValidaTor are given in Section 5. We provide experimental evaluations in Section 6 and security analysis against different attack vectors is in Section 7. We conclude our paper in Section 9.

2 Related Work

Security of DV mechanisms in CAs have been extensively studied. We review existing literature related to DV:

Vulnerabilities in DV. Domain validation mechanisms are susceptible to various attacks that can undermine the security of digital certificates. In [15], the authors outline general security considerations for CAs, highlighting potential weaknesses in various validation methods. Compromised or maliciously configured DNS servers can be exploited to redirect validation requests to attacker-controlled servers [16]. Hijacking the DV process is easy against a vanilla DV, where a single validator communicates with a single server in the target domain.

Such attacks can also be extended to MultiVA, where the validation nodes are either fixed, predictable or the set of possible validators is small, allowing the adversary to attack them all in advance. [11] showed that in those cases adversaries can force all the DV nodes to run the validation against a server of the attacker's choice.

These methods allow attackers to pass validation checks and obtain certificates for domains they do not own, posing a serious risk to web security. Furthermore, [17] demonstrated specific attacks on DV mechanisms, including DNS hijacking and email-based validation exploits, which can be used to fraudulently obtain certificates for domains not owned by the attacker. [18] described man-in-the-middle (MitM) attacks and cache poisoning as significant threats to the integrity of domain validation and [6] demonstrated vulnerability of popular CAs to cache poisoning, allowing adversaries to issue fraudulent certificates. These studies underline the necessity for rigorous validation protocols to mitigate the risks of domain spoofing and unauthorized certificate issuance.

Measurements of DV. Empirical evaluations of CA practices have been crucial in assessing the real-world effectiveness of DV and impact of attacks on DV mechanisms. [19] performed a large-scale analysis of HTTPS traffic to assess the security of DV implementations across various CAs. Their findings revealed inconsistencies in how CAs implement and enforce DV protocols, highlighting potential security gaps.

[20] measured the adoption of security enhancements such as DNS Security Extensions (DNSSEC) and found that despite increased awareness, adoption rates remain low. This indicates a persistent vulnerability in the DV ecosystem due to the reliance on insecure DNS infrastructure. Similarly, [21] evaluated the adoption of DNSSEC by CAs, finding that while adoption rates were increasing, significant gaps remained in ensuring consistent and secure validation processes across the board. Additionally, [22] evaluated the security of email-based DV methods, showing that attackers can exploit weak email configurations to intercept validation emails, further emphasizing the need for robust security measures.

Improvements to DV. In response to identified vulnerabilities and attacks, various improvements to DV protocols have been proposed. [23] presented enhanced validation methods leveraging DNSSEC to provide cryptographic guarantees of domain ownership, although its adoption and security still remain a challenge [24]. [25] suggested a multi-factor validation approach that combines DNS-based validation with additional verification steps such as HTTP-based challenges and validation token usage.

Recently, [26] proposed ADDVent: the idea is to inflate the infrastructure available to DV by crowdsourcing web clients over an advertisement network. This approach is expensive due to the involved costs of advertisement network. Another issue is the trust required in the advertisement network - a malicious network can subvert the security of DV.

Despite significant research efforts in the area of DV, the limitations to deployment of distributed DV introduced by the need for dedicated infrastructure has so far not been addressed. Nevertheless, the limited number of vantage points performing validation and their fixed location played a major role in attacks against DV, exposing the PKI to fraudulent certificates. In this work we aim to resolve these limitations by designing ValidaTor: DV based on Tor.

3 Tor vs. Other Proxies

Since ValidaTor leverages the Tor network first and foremost as a massively distributed set of proxies, it is worth discussing alternative solutions that offer the same capabilities for randomized, distributed DV, without some of the constraints incurred by Tor.

Dedicated volunteer networks. RIPE Atlas [27] and NLNOG RING [28] are examples of volunteer-run multi-perspective systems. The former is the RIPE NCC's global Internet connectivity measurement network, consisting of volunteer-hosted probes that can be triggered to ping an arbitrary destination. The latter is a collective of machines exclusively hosted by network operators, made available to participants for testing network connectivity.

In a similar manner, a distributed network of nodes used exclusively for DV could be created. However, whereas in the above use cases there is little incentive for malicious behavior, with DV there is a strong incentive specifically for adversaries to volunteer and control the validation nodes, acting benignly until an opportunity for a high-value certificate presents itself. It is thus risky to allow such a volunteer network to be open to public participation.

A dedicated network would likely only be feasible if **(a)** participation is restricted to CAs (similar to NLNOG's focus on network operators) or **(b)** the network implements sufficient monitoring and maintenance to guard against malicious participants; something Tor already does. In any case, ValidaTor could be readily deployed now and incrementally moved to a dedicated network, when such efforts get off the ground.

Proxy providers. Rather than using Tor circuits, requiring a minimum of two hops to reach a proxy node, we could halve the performance costs[2] by using a simple proxy provider. However, this introduces two problems: **(1)** all proxies are under the control of a single entity, thus requiring trust in a third party. This could potentially be offset by diversifying proxy providers. **(2)** We find that the costs are prohibitive and threaten the economic viability of such a system. Based on a cost comparison of popular providers, Brightdata [29] offers the lowest rate at $0.42/GB. This covers only datacenter IPs, but still provides over 770K IPs across 98 countries. Based on our projected traffic volume of 317 MBit/s for exit nodes (see Section 6.4), this would amount to $4.2M/year.

Public VPN providers. The same considerations apply to VPN providers. These typically offer fewer, but still sufficient and globally distributed, endpoints. ExpressVPN [30], for example, offers 160 endpoints at $100/year, covering 5 devices. Connecting to all 160 endpoints at once to avoid switching overhead would thus result in just $3200/year. While this supposedly covers unlimited bandwidth, with our estimated traffic of ~1.1 TB/hour, this would likely violate terms of service. It is questionable whether this traffic load could be spread among a sufficient number of VPN providers to fall within a reasonable bandwidth use per provider.

Other anonymity networks. Other solutions, like Nym, require cryptocurrency payments to route traffic based on volume. It is also a smaller network than Tor with exit nodes on the order of a few hundred. I2P's focus is on services internal to its overlay network, rather than proxying connections; for DV we require proxies into the clearnet. The selection of outproxies is comparatively limited, on the order of 10 to 100 and not intrinsically mapped out like in Tor.

To conclude, the alternatives to DV over Tor are either limited in their vantage points and capacity, expensive to use, or pose higher engineering hurdles, which hinders their adoption in the real world. As each alternative comes with its own unique implementation challenges, we choose Tor as the most promising candidate to explore in-depth. Tor removes the financial cost as a barrier to the adoption of massively distributed domain validation and allows an easy-to-deploy solution with sufficient capacity.

4 Overview of Tor

Tor (The Onion Router) is a privacy-focused network of volunteer-hosted "relays" designed to enable anonymous, private communication over the Internet by routing users' requests through multiple encrypted hops, preventing traffic analysis and censorship [12]. **Onion proxies (OPs)** are the entrypoint for client software using Tor, responsible for managing circuits and connections to websites. **Onion routers (ORs)** are the relay-components which form the circuits. Each OR decrypts one layer of encryption to determine the next OR to forwards the packets to [31]. The **exit node** is the final node in a circuit, decrypting the user's original packet for its final hop to the destination server. From the server's perspective, the traffic appears to originate from the exit node.

Crucially for ValidaTor, all traffic until the exit node is protected. All unprotected traffic originates at the exit node, including DNS resolution. ValidaTor can therefore leverage a large variety of perspectives (~2.2K) using only a single server. Because the exit node sees the original unencrypted packet, it also presents the most critical threat. Node selection is therefore designed to minimize the risk of malicious, colluding exit nodes (see Section 7.5). Lastly, the list of active ORs is maintained by **directory servers**. These servers provide the necessary information for OPs to build circuits, which is periodically updated and signed to ensure integrity [12].

In this work we leverage the Tor nodes as an overlay to gain two critical benefits: first, we obtain a distributed infrastructure for DV with a large number of potential validator nodes which supports unpredictable selection. Second, the DV communication protocol benefits from security on the network layer making it impossible for any adversary before the exit node to observe the validation requests to the domains.

5 ValidaTor: Design & Implementation

We explain the limitations in existing DV mechanisms, and discuss how we resolve them with our proposal, ValidaTor, which uses Tor as an overlay to validate control over domains. We present the design and the implementation of ValidaTor, and provide motivation for different architectural choices.

5.1 Resolving Limitations in DV

The central problem with DV is that it is, by necessity, unauthenticated and thus vulnerable to manipulation by a Man-in-the-Middle (MitM) adversary. Successive improvements to DV, as laid out in Sections 1 and 2, have therefore focused on increasing the difficulty and resources required to mount a successful attack, and limiting an attacker's degree of influence on the process as a whole.

With vanilla DV, an attacker needs to target only a single server or intercept only a single communication path. With MultiVA this is increased to 3 or 4 servers and previous research has worked towards optimizing the distribution of these vantage points. Nonetheless, there are several ways in which the remaining attack surface can be further reduced and the use of Tor can address all of these by routing DV requests through a random set of exit nodes:

Dynamic validators. A static set of validators is vulnerable to targeted attacks. Adversaries can plan and prepare ahead of time, knowing exactly where their targets are. The same applies for a random but *small* set of validators, which could still be exhaustively targeted. Tor provides access to a large number of endpoints, allowing a vast number of possible combinations to be chosen from for each validation at random. This unpredictability means an attacker needs to be prepared to attack a significant fraction of these endpoints to have a

[2] We would still need to use HTTPS proxies to protect data in transit.

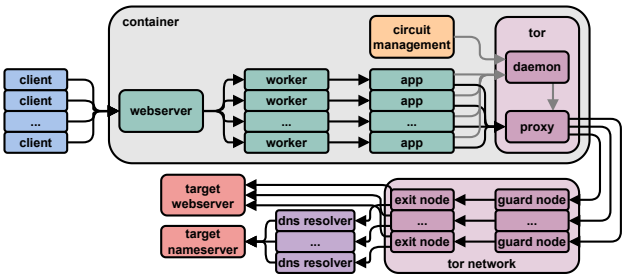


Figure 1: Container design and domain validation request flow.

realistic chance of hijacking DV. The size and distribution of the Tor network makes this prohibitively difficult.

Path diversity. A lack of diversity in the paths from validators to target domains results in more choke points in the network, where an attacker has a larger degree of influence. Whether an adversary is already on-path or uses routing hijacks to gain an on-path position, an attacker's influence in the BGP network is topologically constrained. Thus, reducing the overlap of validation paths reduces the set of remaining potential adversaries. The diverse set of endpoints offered by Tor allows validators to be selected with a significantly higher degree of distribution.

Denial-of-Service. Blocking DV by on-path adversaries can lead to denial-of-service for certificate issuance. As Tor is already set up to circumvent censorship through its distributed nature, Tor-based DV benefits from this as well.

5.2 Design

The core idea behind our design of ValidaTor is to select a random set of Tor exit nodes for each validation to act as validators. The nodes involved in any given validation are selected such that they are highly distributed in the BGP network to limit the degree of path overlap.

Component interactions. To implement a proof-of-concept for the ValidaTor server we construct a containerized environment using Docker, containing the following components: the Tor service, a custom circuit management service, a webserver and Python `Flask` application for handling the validation logic, i.e., triggering requests to target domains and aggregating the results.

As outlined in Figure 1, a validation applicant first sends a validation request to the webserver (see below for further details). A worker process then runs `Flask` application's validation logic, which triggers multiple web requests to the target domain through Tor's SOCKS5 proxy. The application also sends control instructions to the Tor daemon to assign the resulting streams to a random set of circuits, whose exits are ensured to be sufficiently distributed. The circuits are built and managed in parallel by the circuit management service (discussed in detail in Section 5.3), which likewise communicates with the Tor daemon through control instructions. Once streams are assigned to circuits, the Tor proxy forwards each request to its guard node, which then forwards it to the exit node. Each exit node resolves the target domain through its DNS resolver and subsequently queries the challenge resource at the resolved IP address. The responses from the target webserver are then returned via the Tor circuit to the application, which aggregates them and produces the reply that the container's webserver sends back to the client.

Validation. We implement our proof-of-concept ValidaTor as a web service that CAs and other clients can use to validate a domain by specifying the `.txt` resource to be challenged (as used by the ACME standard RFC8555 [32]), the target domain, and the protocol with which to send the request (either HTTP or HTTPS). The service then attempts to retrieve the resource via an initial k exit nodes. If the content of the resource (e.g., in ACME, a 43 character random alphanumeric string) does not match across all k requests, the system attempts to tie-break with additional exit nodes up to a maximum of n. If at least k responses contain the same content, this content is returned, otherwise an error.

This mimics the k-out-of-n validation scheme used by Let's Encrypt, where initially 3 out of 4 validation servers attempt to retrieve a specific `.txt` resource, and only if these do not receive the same result the 4th server is used. Then, if at least 3 out of the 4 results match the expected response, the validation is completed successfully.

5.3 Constructing Circuits

This section details the challenges in managing DV over Tor and how we construct and assign circuits to ensure both randomization and distribution for each validation.

Manual stream assignment. During regular client operation, the Tor service maintains a handful of circuits, constructs new circuits based on demand, i.e., the amount of requests sent through its SOCKS5 proxy, and rebuilds circuits every 10 minutes. A key feature of Tor to protect from profiling is that requests to domains are sent through different circuits. For our purposes such protection does not play a role. It *is*, however, critical that all n requests to the *same* domain are sent through different circuits (specifically, different exit nodes) so that we validate a domain from at least k perspectives.

We therefore configure Tor to allow *us* to choose which circuits to attach which requests (so-called "streams") to. This can be done in Python through Tor's official `stem` library, which provides an interface to send control messages to the Tor daemon. This allows us to ensure that requests for the same domain, i.e., that are part of the same validation, are sent through different circuits.

Choice of exit nodes. In establishing the circuits, we ensure **(a)** sufficient distribution of the exit nodes of the circuits used for a given validation, and **(b)** that exit nodes are chosen uniformly at random. As such, we cannot rely on Tor's relay selection algorithm, which is focused on anonymity and effective load balancing. Multiple requests to the same domain would be routed through the same circuit and there is no guarantee that a new circuit will use an exit node with a different

IP address, let alone an exit node in a different network. Exit nodes are also selected based on available bandwidth, which would reduce the unpredictability of validator selection. We therefore build circuits manually using `stem`. We begin by selecting randomly among all nodes marked with the `EXIT` flag. This flag is assigned to relays that allow their use as an exit node and have been monitored by Tor for some time. We exclude nodes that have been flagged as `BADEXIT`, which is assigned to exit nodes that behave suspiciously (~2.2K out of the ~7K total nodes), but additionally impose the requirement that a selected node's IP address does not share the same /8 prefix with any of the other already selected nodes, thereby ensuring distribution in the BGP network. The prefix length can be adapted to the desired granularity.

Cycling circuits. Constructing a new dedicated circuit for every request produces unnecessary latency and overhead. By default, Tor maintains a handful of open circuits and assigns requests to these based on the target domain. That is, requests to different domains are routed through different circuits. If the current pool of circuits does not accommodate this, new circuits are built on-demand. Circuits are then used for up to 10 minutes, by default, and only cycled after.

Thus, similar to Tor's regular operation, we maintain a set of circuits and periodically rebuild these to avoid too many validations using the same endpoints. However, we maintain a larger set of concurrent circuits than Tor normally would, to increase the unpredictability of the endpoints chosen for a given validation.

A larger pool to select from means a larger number of possible configurations, as given by the binomial function, and thus, greater uncertainty for the selection of any particular configuration. In the opposite extreme, where the number of concurrent circuits is equal to the number of validators, there is exactly one configuration. If this "pool" of circuits is maintained for any significant period of time, an attacker could identify the validators with a single validation request, and use this information to potentially manipulate subsequent validation requests.

Ideally, we would maintain a circuit for every available exit node. However, we observe that upwards of around 70 - 80 open circuits become unstable, as the Tor daemon strongly conflicts with our circuit building application, closing circuits that are not in use, and thus limiting the number of circuits we can keep open concurrently. Through experimentation we find that the largest number of circuits that remain stable is at around 50-60 concurrent circuits. We then rebuild circuits every 3 minutes. This can be configured to match the desired trade-off between network overhead and circuit "freshness".

Validation queries for a given domain can then efficiently be assigned to a random subset of circuits (which, as noted above, are already ensured to be distributed).

2-Hop circuits. Another reason to build circuits manually is that Tor by default builds circuits with 3 hops. We do not require anonymity, hence we can increase performance by building our own circuits and reducing the number of hops. Ideally, our Tor client would connect directly to each exit node. However, exit nodes need to be configured explicitly to allow 1-hop circuits; the default configuration is to refuse these. As this would limit our choice of endpoints from which to validate, we opt for building 2-hop circuits.

Choice of entry nodes. Since anonymity is not of vital concern, we optimize our entry nodes for performance. Conveniently, Tor assigns the `GUARD` flag to nodes that have been active for at least 8 days and have been independently measured to confirm their otherwise self-reported available bandwidth. This indicates that these nodes are particularly suitable as entry nodes.

To maintain a degree of load balancing across the network we randomly select an entry node from among the ~2.5K nodes marked `GUARD` and `FAST`. However, we weigh each node by its advertised bandwidth and IP prefix overlap with our own IP address, in order to favor shorter network paths as well as high-throughput nodes.

5.4 Challenge Methods

While both HTTP and HTTPS requests can be sent through the Tor network, thereby allowing for HTTP-based domain validation, Tor currently supports DNS only partially, i.e., for the resolution of domain names via A and AAAA records and reverse-lookup of IP addresses via PTR records. This allows domains to be resolved by the exit node.

Crucially, DNS TXT records, used for DNS-based domain validation, are currently *not* supported. Our proposed system is thus limited to HTTP-based validation until Tor adds support for DNS TXT records. However, since HTTP-based validation is the default for common ACME implementations, we argue that this is not a severe limitation.

5.5 Easy Adoption

Proposals which require substantial changes to the existing infrastructure take decades to deploy, for instance, DNS Security Extensions with DNSSEC RFC9364. Therefore, to facilitate their adoption, new mechanisms should require minimal changes. Our design follows this principle, requiring only to patch ACME's request for the target resource (e.g., `http://example.com/.well-known/acme-challenge/random`) with a request to ValidaTor (e.g., `https://validator.com/http/example.com/.well-known/acme-challenge/random`). If ValidaTor asserts that the responses match, ACME receives the same file content as from the original request and continues operating unchanged. If ValidaTor finds differing responses, it will return an error, causing ACME to fail the validation.

6 Performance Evaluation

In this section we validate the time required for validation and the throughput with ValidaTor. We then show how to use parallelization to further scale the system. Lastly, we discuss

the implications on bandwidth usage and from blocking of Tor traffic.

6.1 Single Container

For our initial evaluation we run our Docker container on a virtual server with 24 cores, allowing a certain degree of parallel processing of validation requests, using an equivalent 24 workers for the webserver in the container. Once the container is running and the 50 Tor circuits have been built, we trigger validation procedures via HTTPS requests to the webserver in the container. We run 80 querying threads in parallel, each continuously submitting a domain to validate and waiting for the response. We select the top 10k Tranco [33] domains that host a webserver and request the `/robots.txt` document to simulate an ACME query.

This initial evaluation provides a baseline performance benchmark for different numbers of validators. We test three different scenarios: 3-out-of-4 (equivalent to Let's Encrypt), 5-out-of-7, and 7-out-of-9, to provide an intuition as to the impact of the number of validators on performance and security. We note here that we observe no disagreement among the validator results across all of our tests. As such, each validation produces exactly k requests.

We note the resulting hardware requirements derived from `docker stats` for interested readers in the Appendix A.

Validation duration. Figure 2 shows the distribution of the duration of each validation. This refers to the time between the server receiving a validation request from an applicant and receiving the last validator response necessary to produce a verdict. While the spread increases as k increases, a client can still expect their validation request to complete within around 20 seconds, even with $k = 7$.

We compare the duration of certificates issuance to certbot [34]. Certbot is a popular open source ACME client by the Electronic Frontier Foundation (EFF) for automatically requesting certifcates from Let's Encrypt via the ACME protocol. When measuring certbot, we start a manual DV procedure, place the created challenge value on our webserver, start our timer, and then trigger certbot to begin the validation. We stop our timer when certbot exits. We find that for $k = 3$ the median duration is comparable to that observed when using certbot, and can be further improved with parallelization, as shown by the multi-container tests.

Since obtaining a certificate is an occasional process, and tools like certbot can often be used to complete the validation process without the user's interactive involvement, we consider the additional latency incurred by the use of Tor circuits to be negligible for the end user.

Validation throughput. Table 1 shows the validations per second we achieve with our prototype implementation. Increasing k leads to an expected decrease in throughput, as more resources are required for each individual validation. However, ValidaTor can also easily be scaled horizontally to accommodate higher validation throughput. By further optimizing the implementation and parallelizing the deployment, it is thus feasible to scale the system to match even the rate of ~65 certificates Let's Encrypt issued on average per second on peak days in late 2024 [35].

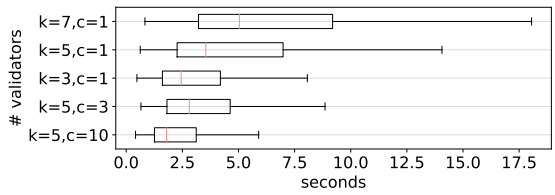


Figure 2: Validation duration with k validators and c containers.

containers	k	validations / s
1	3	2.7
1	5	2.1
1	7	1.6
3	5	6.5
10	5	11.9

Table 1: Validation throughput.

6.2 Scaling

We extend the Tor proxy and circuit management system in ValidaTor to handle multiple validation requests in parallel, assigning each request to a different circuit. As we next show, this parallelization improves the throughput and efficiency of the validation process.

To further scale the throughput of ValidaTor we conduct two additional experiments. For the first we deploy $c = 3$ instances of our container on the same VM, each running 24 workers, building 50 circuits and using $k = 5$ validators per domain. We use 200 client processes to issue the validation requests and load-balance them among the containers.

For the second scaling experiment we increase parallelization even further, deploying $c = 10$ containers, using 20 cores each (on a sufficiently capable server), building 15 circuits each (for the same total of 150 circuits), and also using $k = 5$ validators per domain. We load-balance the same 200 client processes among the containers.

The 3-container deployment achieves the expected, roughly 3x increase over the single-container deployment at 6.5 validations per second. The 10-container deployment does not quite match its expected throughput, but nonetheless increases throughput to 11.9 validations per second.

The increase in performance over the single container is due to parallelization of the Tor daemon (with a separate instance in each container), which mediates the control requests required to manually assign web requests to Tor circuits. This demonstrates encouraging evidence that such a system can be scaled horizontally to production-grade performance through sufficient parallelization.

6.3 Blocking of Tor Traffic

As a well-known anonymity network, Tor may sometimes be discriminated against by networks that seek to prevent anonymous access, potentially limiting its use for DV. We

investigate the significance of this using two validation measurements (2025-02-08), one based on the Tranco top 10K and one sampling 10K domains from Certificate Transparency logs (accessed via certstream), in order to compare potential differences between popular and regular domains, both using a 5-out-of-7 scheme.

We augment our initial implementation to match each exit node and target IP with the request result obtained by that node. If traffic is being blocked we expect to see a `ConnectTimeout` or `ConnectionError`. As shown in Figures 3 and 4, this is the case for around 20% of requests, regardless of the popularity of the domains. CT-sampled domains show no difference to top Tranco domains. In addition, Figure 4 shows that all vantage points tend to agree on the result, i.e. they all either succeed or fail. This strongly suggests that blocking of Tor traffic is done primarily by the domains themselves, rather than transit networks. In the latter case, since the exit nodes are widely distributed, we would expect to see cases where e.g. only one node fails to connect, because its request is routed through a blocking AS, whereas the others are routed differently. This is not the case, however.

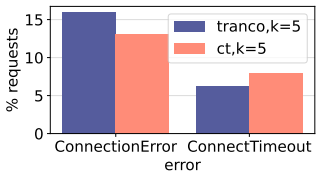


Figure 3: Rates of connection errors.

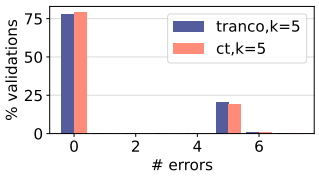


Figure 4: Errors per validation.

We further investigate this by mapping the IP addresses of exit nodes and target servers to their respective AS and simulating the AS paths taken. We then count how often a given AS is on-path for requests that ended in a connection error versus requests that succeeded. If an AS blocks Tor traffic, we would expect to see all requests routed through it to produce connection errors. The impact of such an AS on ValidaTor is significant, if it is implicated in a large number of validation attempts. However, as we see from the data, the impact is not significant. For the CT-based measurement, we see only 10 out of 816 ASes that consistently feature on the paths of failing requests, with only 4 seen in more than 1 validation round and none of them transit ASes. For the Tranco-based measurement it is 35 out of 2531 ASes that consistently produce failures, with each only appearing in 1 validation round and only 8 of them being transit ASes.

These results further solidify the case that blocking of Tor traffic by transit ASes is a rare occurrence without significant impact on the performance of ValidaTor. The key source of validation failures are the destinations of the validation requests. This means it is within the power of domains to allow DV traffic from Tor nodes.

6.4 Bandwidth Impact

In January 2025, the Tor network measured an average 347 GBit/s of consumed bandwidth across all relays (98 GBit/s at exit nodes), a little over a third of the total available advertised bandwidth of 917 GBit/s (less than a third of the 314 GBit/s advertised by exit nodes) [36].

The maximum total traffic produced by our container (as measured by `docker stats`) was 482MB for 10K validations and $k = 7$ (including circuit creation). The average size of the HTTP response bodies observed in our measurements is 528 bytes, whereas an ACME challenge text file is 43 bytes in length. Thus, substituting the proper content for the DV requests, while maintaining the same HTTP headers and Tor circuit building, results in $482MB - 10,000 * 528B + 10,000 * 43B = 477MB$.

As indicated by the steady increase in total traffic and runtime difference between the $k = 3$, $k = 5$ and $k = 7$ conditions (shown in Figure 11), the network load scales linearly with the number of validations and validators. Thus we can fairly reliably extrapolate to the scale of the entire Web PKI. As an estimate of the current certificate volume we track certificate issuances via Certificate Transparency logs (using certstream) for a period of 15 hours, during which we observe an average rate of 104 new certificates per second. Since this accounts only for successfuly validations, we can safely assume at least twice the number of validation attempts. Thus, we can expect:

$$\frac{208}{s} * \frac{477MB}{10,000} * 8 = 79.4MBit/s$$

that is, ~$11.3MBit/s$ per validator. For $k = 5$ and $k = 3$ we get similar rates of $11.7MBit/s$ and $12.7MBit/s$, respectively, matching the expectation that the traffic volume is roughly linear in the number of validators.

In addition to unsuccessful validation attempts, we need to account for chasing of redirects, which is often done during DV, but is not included in our bandwidth data. Redirects are processed by the web client and thus result in two or more distinct requests. Assuming for simplicity an average of 1 redirect per validation, we can expect a rate of $2 * 79.4MBit/s = 158.8MBit/s$. This is a conservative estimate, since we observe redirects on only ~$\frac{1}{3}$ of all domains, with popular Tranco domains using redirects twice as often as the broader Internet, sampled from CT logs.

Our bandwidth measurement also only covers the data flowing in and out of our server. This includes both request and response data, but only for the link between our server and the entry node of our Tor circuit, thus considering only the ingress load at the entry node. This same data is also transmitted between entry and exit node, adding the same load at both the entry node egress and the exit node ingress, as well as between exit node and target server, producing another egress load at the exit node. Thus, the overall load on the Tor network is $4 * 158.8MBit/s = 635.2MBit/s$, with $317.6MBit/s$ hitting the exit node.

Thus, we conclude that even at the current volume of the entire Web PKI and using 7 validators, routing DV traffic through Tor consumes a negligible amount of the network's bandwidth: just 0.11% of Tor's *remaining* total bandwidth and 0.15% of the exit nodes' *remaining* bandwidth. Furthermore, ValidaTor containers could be horizontally scaled to the point where each container has a handful of dedicated Tor exits, avoiding the need for periodic recreation of Tor circuits and thus reducing bandwidth usage.

7 Security Analysis

ValidaTor ensures high path diversity by selecting a random set of exit nodes for each validation request. This randomization makes it difficult for attackers to predict and intercept the validation traffic. We demonstrate experimentally on heuristically derived datasets that path diversity attained with Tor nodes is significantly higher than that obtained with the dedicated infrastructure of Let's Encrypt. Additionally, ValidaTor configures the Tor client to avoid using exit nodes with the same /8 IP prefix, ensuring that the exit nodes are sufficiently distributed across the network. This configuration enhances security by reducing the likelihood that an attacker can control multiple exit nodes in the same circuit.

7.1 Perspective Diversity

Routing domain validation traffic through the Tor network allows us to access many more validation perspectives than a dedicated infrastructure, such as that of Let's Encrypt.

We create a new domain, trigger validation requests for this domain using certbot, and observe the IP addresses logged on our webserver. We repeat this 5 times[3], collecting a total of 11 unique IP addresses. Using Tor's stem library, we obtain all known relays and filter for those marked with the EXIT flag and *not* marked BADEXIT. At the time of our experiments this list contained 2254 nodes with 1221 unique IP addresses.

To consider the diversity in perspectives with regards to BGP routing, we need to map IP addresses to ASes. CAIDA [37] provides a prefix-to-ASN mapping based on BGP origin announcements collected by the RouteViews project. However, as pointed out in [9], this underestimates routing diversity when different prefixes are announced through different paths using the same ASN, as is the case for e.g. AWS datacenters, which host Let's Encrypt validators. Multiple unique origins are thus aggregated under the same ASN. We "de-aggregate" ASNs by considering each prefix as a unique origin, if it is announced through a different path, in line with [9]. This is done based on the most recent available route collector data from RIPE RIS [38] and RouteViews [39].

For the 11 Let's Encrypt IP addresses we obtain 7 unique BGP origins (instead of just 2 when considering aggregated ASNs), whereas for the 1221 Tor exit IP addresses we obtain 280 unique BGP origins (210 with aggregated ASNs).

[3] Certbot allows a maximum of 5 validation attempts in short succession. We see no new BGP origins after the first 3 attempts.

7.2 DNS Perspective Diversity

Tor is implemented such that, when proxying an HTTP request to a given domain through a Tor circuit, the exit node will take care of resolving that domain to an IP address. As pointed out in [9], the DNS resolution path is part of the attack surface for hijacking domain validation. As such, diversifying validators is only useful if the DNS resolvers used by those validators are also diversified. It could be possible that many of the exit nodes use the same few DNS resolvers. While Tor explicitly encourages the use of default resolvers and resolvers within an exit node's network, and discourages the use of public DNS resolvers to avoid centralization, we aim to confirm this by measuring the IP addresses of the resolvers used by exit nodes.

To investigate this we set up a fresh domain with an NS record pointing to a server under our control and log incoming DNS requests on port 53 using `tcpdump`. For Let's Encrypt we again trigger 5 validation requests through certbot, for Tor we send a web request through each of the 2254 exit nodes. In each case this triggers the attempt to resolve our domain using our server, allowing us to track which IP addresses the DNS requests are coming from. By correlating the timeframe of each request as recorded client-side with the timestamps of DNS requests received server-side, we match each Tor exit node IP address with the most likely resolver IP address. Note that, across all certbot measurements, we never actually complete the validation process. Therefore, no certificate is created to be logged in and scraped from Certificate Transparency logs. Thus, our webserver logs are not contaminated by various Internet scans (see for example [40]).

For Let's Encrypt we observe 28 unique resolver IP addresses, corresponding to 9 unique de-aggregated BGP origins. For Tor we matched 1118 unique resolver IP addresses to the 1221 unique exit node IP addresses, corresponding to 174 unique deaggregated BGP origins. While we observed additional IP addresses from another 100 BGP origins, we could not clearly match these to exit nodes and thus use 174 as a lower bound.

7.3 Path Diversity

To test how well diversity of perspectives translates to diversity of BGP paths taken from validators to a target domain we reconstruct the BGP network at the prefix level from route collector data and apply the most recently available CAIDA AS relationships dataset [41]. We map each IP, as observed in our experiments, to its de-aggregated ASN and calculate the paths from each source ASN to the observed destination ASN using the `pybgpsim` library from [42]. We also consider the return paths from destination ASN to source ASN, since these might be different.

Using this method we can recover the transit ASes A for each path and see how many ASes the paths of a given validation have in common. That is, how many ASes can intercept what percentage of validators for how many domains?

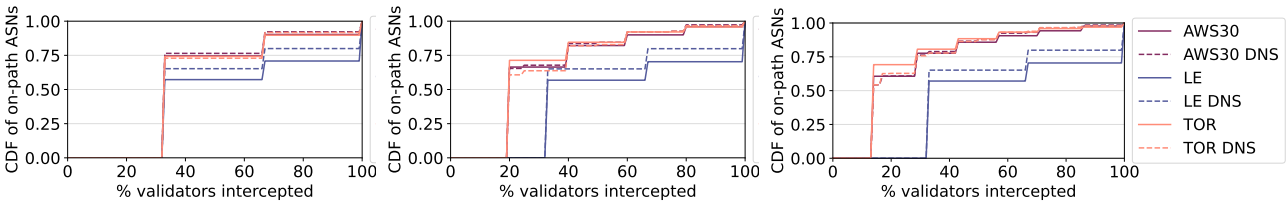


Figure 5: Cumulative distribution of the percentage of validators intercepted by each on-path ASN for each domain ($k = \{3,5,7\}$ from left to right).

In essence, we look at each on-path ASN for each domain validation and determine the percentage of validators that transit through this ASN. We plot the resulting cumulative distribution for each $k \in \{3,5,7\}$ in Figure 5.

For Tor, we use the validator-target pairs as observed in our experiments. For the Let's Encrypt comparison, we simulate the validator-target pairs by truncating to the first 3 pairs (since Let's Encrypt does 3-out-of-4 validation, only using the 4th if the first 3 do not agree) and replacing the first validator with Let's Encrypt's main network (23.178.112.0/24, which is present in every validation attempt triggered with certbot) and the two other validators with a random selection of 2 out of the remaining 6 BGP origins observed in Section 7.1. The same substitution process is also used for the AWS30 comparison, an optimally distributed AWS deployment of 30 locations (discussed in detail in Section 8).

As indicated by Figure 5, using Tor exit nodes as validators significantly decreases the number of ASes in a position to intercept a large fraction of validators for a large number of domains. This is particularly true for ASes that can intercept 100% of validators, and thus forge the overall validation result (reduction of 21%). This number further decreases with additional validators ($k \geq 5$), as the overlap between validation paths decreases (reduction of up to 27%). The comparison with the hypothetical AWS30 deployment shows that such path diversity can also be achieved with a cloud-based deployment, albeit at much greater cost.

7.3.1 Path Overlap Metric

While reducing the number of ASes that can intercept all validators translates directly to fewer threats of fraudulent certificates, it is important to consider the degree of influence of individual ASes on validation more generally. The lower the path overlap between validators, the lower this influence. We therefore propose a metric that summarizes the overall degree of path overlap in a single number, for easier characterization and comparison between deployments.

First, we define the path from a validator i to a domain d as the set of ASes transited both from validator to domain and from domain back to validator, denoted P_{id}. The degree of overlap between the paths can then be calculated as the Jaccard distance between the sets:

$$jaccard(P_{id}, P_{jd}) = \frac{P_{id} \cap P_{jd}}{P_{id} \cup P_{jd}}$$

A value of 0 therefore means the paths of no ASes in common, whereas a value of 1 means both validators transit all the same ASes to reach the target domain. To capture the degree of overlap for a given validation with a group of k validators, we calculate the pairwise Jaccard distance and average over the number of possible pairs:

$$val_overlap(d) = \frac{\sum_{i=1}^{k} \sum_{j=i+1}^{k} jaccard(P_{id}, P_{jd})}{\frac{k(k-1)}{2}}$$

Finally, we average the overlaps across all D validations for a given deployment: $overlap(D) = \sum_{d=1}^{D} val_overlap(d)/D$.

We calculate this metric for ValidaTor $\forall k \in \{3,5,7\}$, as well as the the simulated Let's Encrypt and AWS30 deployments for each of our measurements, and compare the results in Figure 6. We see that the switch to more nodes introduces the most dramatic decrease in average path overlap (~50%), with smaller incremental decreases when using additional validators with the larger Tor pool.

7.4 DNS Path Diversity

To evaluate the diversity of the paths from DNS resolvers to nameservers, we repeat the same evaluation, but with the observed resolver and nameserver IP addresses. Specifically, for ValidaTor, we replace each exit node IP with the matched DNS resolver IP. For the Let's Encrypt comparison we again truncate the validator-target pairs to 3, replace the first exit node IP with the IP of the DNS resolver of the main Let's Encrypt server, and replace the remaining 2 exit node IP addresses with a random selection of 2 IP addresses from the observed DNS resolvers (from different BGP origins).

In both scenarios the target IP of each validator-target pair is replaced with the IP of a random nameserver of the target domain. These IP addresses were previously collected by querying the NS records of each domain and then the A records for each NS result. Due to the randomization we conduct 5 rounds for each deployment and average the resulting path overlap metrics.

The results are also shown (dashed) in Figures 5 and 6. While Let's Encrypt's DNS resolvers show higher path diversity than its validation servers, Tor still offers marked improvements.

7.5 Malicious Exit Nodes

We consider attacks by malicious Tor exit nodes and show how to configure the ValidaTor parameters to substantially decrease the risk of such attacks.

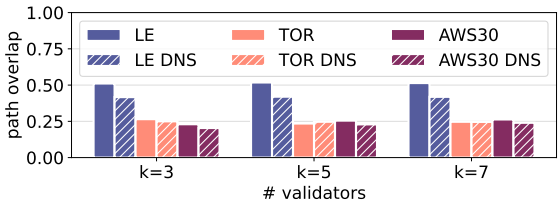


Figure 6: Path overlap for Let's Encrypt, Tor, an optimally distributed AWS deployment of $N = 30$ nodes and respective DNS resolvers.

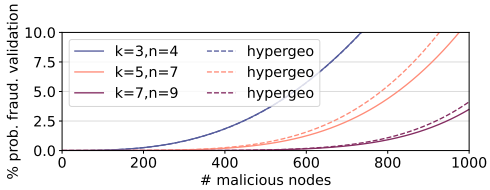


Figure 7: Probability of fraudulent validation using compromised exit nodes.

7.5.1 Probability of Fraudulent Validation

Leveraging Tor exit nodes as vantage points for domain validation introduces the risk of attackers running malicious nodes to intercept and manipulate the validation process. In essence, this is similar to the man-in-the-middle threat posed by BGP hijacks and other routing manipulations. Here, we model the probability of an attacker successfully manipulating validation this way and show that is exceedingly small.

Random selection. Due to the k-out-of-n successful query results required for validation, an attacker would need at least k of his nodes selected for validating the same domain. With a fully random selection, i.e., each node having an equal chance of being selected, the probability of exactly k out of n nodes being malicious is given by the hypergeometric distribution:

$$hg(M,N,n,k) = \frac{\binom{M}{k}\binom{N-M}{n-k}}{\binom{N}{n}}$$

where N denotes the total number of exit nodes in the Tor network (2293 measured at the time of writing) and M denotes the number of nodes under the attacker's control. The probability for *at least* k malicious nodes is thus:

$$p(M,N,n,k) = \sum_{j=k}^{n} hg(M,N,n,j)$$

This is plotted as dashes in Figure 7 for different k and n.

Accounting for prefix-aware selection. However, the node selection process is not fully random, since every time a node is selected, all nodes on the same (configurable) network prefix are excluded as validators for the same validation to ensure distribution. This means an attacker would need to control nodes in at least k different networks to even have the possibility of using them to validate the same domain, significantly raising the attacker's resource requirements.

To simplify the model and give the attacker the highest chance of success, we assume the attacker's M nodes are distributed equally across all networks. Similarly, we assume that each network contains the same total number of nodes. While this is not strictly the case in practice, it allows us to simplify the chance of selecting a malicious node to an average probability of $\frac{M}{N}$, regardless of how many nodes have already been selected. This is because the ratio of malicious to total nodes remains constant if the same number of malicious and legitimate nodes are removed from the pool each round.

Accounting for staged selection. The second difference to a simple k-out-of-n selection is that, for efficiency, the selection is done in stages. That is, we begin with a selection of k validators and only incrementally add additional validators (up to a maximum of n) if agreement has not yet been reached with the validators selected thus far. This means, in order to reach the threshold of k by the n^{th} selection, an attacker would need at least $k-(n-k)$ of his nodes to be selected among the first k. Exactly how many malicious nodes are among the first k then determines how many additional selections are made and need to also be malicious. With the simplifying assumptions above, this selection process results in the following probability, that an attacker can control the outcome of a validation using malicious exit nodes:

$$p(M,N,n,k) = \sum_{j=k-(n-k)}^{k} \binom{k}{j} (\frac{M}{N})^k (1-\frac{M}{N})^{k-j}$$

Figure 7 plots this function for different parameters k and n. While very similar to the hypergeometric distribution, the more accurate modeling of the selection process works slightly to the disadvantage of the attacker for $k > 3$. Overall, we see that, given the current number of $N = 2293$ Tor exit nodes, the attacker's probability of success is very small even when controlling several hundred exit nodes. Note that we assume for the calculation that the attacker controls M of the *existing* N nodes, giving him a stronger advantage than adding M *additional* nodes. Second, we see that the risk can easily and significantly be reduced by simply increasing k. For example, at $M = 700$ the attacker's chance of success can be reduced from ~10% with $k = 3$ to ~2.5% with $k = 5$ and to less than 1% with $k = 7$.

Tor has seen malicious actors adding large numbers of nodes in the past (e.g. ~1000 in 2020 [43]). In this example the attacker would have an 8.7% chance of having all $k = 3$ of their nodes selected to validate their request but only a 0.4% chance of having all $k = 7$ of their nodes selected. A possible mitigation for such attacks could be the temporary increase of k when the Tor network appears to be under attack. For example, at $k = 9$ the attacker's chance of success falls to 0.05%, despite controlling 1000 out of a total of 3293 nodes.

7.5.2 Additional Countermeasures

The threat of colluding exit nodes threatens the security of every other application of Tor. The network is therefore continuously monitored for bad actors by the Tor Project [44], researchers [45] and other volunteers. Implicated nodes are either flagged as `BADEXIT` or removed from the network consensus, aiding the selection against malicious nodes.

Such monitoring could be further extended in the context of our validation system. Across our experiments we very rarely see discrepancies between the results reported by different validators, even when $k >= 7$. Multiple discrepancies involving the same exit IP addresses could therefore easily be detected and flagged, avoiding the use of these exits. Similarly on the the client-side, multiple failed validations in a short period of time could lead to blocklisting of a client's account for that domain, as implemented by Let's Encrypt with certbot. As such, we can effectively prevent an attacker from playing the numbers game and simply attempting many validations until one succeeds.

8 Randomized Distributed DV in the Cloud

Following the comparisons between the Tor network and Let's Encrypt's current MultiVA deployment, it begs the question what a cloud deployment would look like that achieves the same security properties. We begin by fixing the desired maximum probability of a fraudulent validation at a generous 1%. We then consider the following two scenarios: **(a)** using 3-out-of-4 validation as is currently done by Let's Encrypt (more beneficial if traffic is more expensive than additional datacenter nodes) and **(b)** using 7-out-of-9 validation (more beneficial if nodes are more expensive than traffic). We then plot the probability of fraudulent validation, as described in Section 7.5.1, for different numbers of total vantage points N and vantage points hijackable by an attacker x. The results for each scenario are shown in Figure 8.

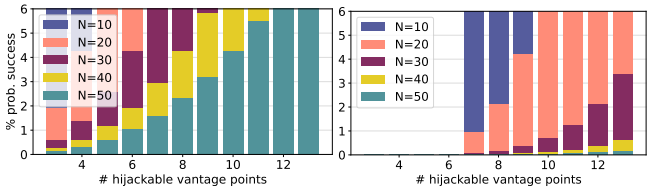


Figure 8: Probability of fraudulent validation by hijacking cloud vantage points, using $k = 3$ (left) and $k = 7$ (right) validators.

For 3-out-of-4 validation, at least $N = 30$ vantage points are required to keep the probability of fraudulent validation under 1%. However, if, for example, the attacker is assumed to be able to hijack up to 6 vantage points, at least $N = 50$ is required. 7-out-of-9 validation naturally remains 100% secure if the attacker cannot hijack at least 7 vantage points. However, even with up to 10 compromised vantage points, the probability of success is still well below 1% at $N = 30$.

To evaluate the path overlap (see Section 7.3.1), for such a deployment, we consult the published IP ranges of AWS EC2 networks and select the 30 vantage points with the least pairwise overlap between network addresses. We then perform the same simulation of AS paths for validation requests as in Section 7.3. For $k = 3$ we see a path overlap of 0.23, comparable to Tor-based validation. For $k = 7$ we see a slightly higher overlap of 0.29 (see Figure 6). It is therefore possible to achieve similar degrees of distribution simply with additional cloud vantage points. However, an increase from 4 to 30 vantage points and the additional traffic of $k = 7$ versus $k = 3$ validators in this scenario would multiply the costs many times. Additionally, a lower risk threshold of under 1% increases the required number of vantage points.

9 Conclusions

Our design of ValidaTor, i.e., DV over Tor, resolves many of the existing limitations of DV and MultiVA. Leveraging Tor as a massively distributed set of vantage points from which to conduct multi-perspective domain validation offers clear advantages over existing systems:

Randomization. Vantage points can be randomly chosen from a large set of available nodes, preventing attackers from knowing in advance which validation nodes to target. This prevents any type of targeted attack, such as prefix BGP hijacks and DNS cache poisoning.

Resilience to interception and malicious Tor exit nodes. Tor exit nodes, as well as the DNS resolvers they use, are more distributed than the vantage points of existing systems. As we show in our experimental evaluations, this results in a significant reduction in the number of ASes that can intercept *all* vantage points used for a validation of a given domain. Such ASes can impersonate validation by intercepting the validation process to obtain fraudulent certificates.

In order to stand a statistically significant chance of fraudulently validating a domain, an attacker needs to control on the order of several hundreds of Tor nodes. Since Tor has a vested interest in avoiding such scenarios to protect from, e.g., traffic correlation attacks, this is unrealistic to persist. Simply increasing the number of validators for a given validation drastically decreases an attacker's chance of success.

DoS resistance. If individual Tor relays, ASes, or routers maliciously drop domain validation traffic, the system can choose new circuits through which to route traffic. Ensuring the same degree of availability with MultiVA requires a much larger deployment.

Lightweight load on Tor network. Since domain validation traffic is lightweight (only querying small text resources, most often via plain HTTP) and circuit management can be implemented without overly frequent rebuilding of circuits, the projected load on the Tor network produced by even a Let's Encrypt-scale volume of validation requests is negligible compared to Tor's available bandwidth.

Open infrastructure. ValidaTor scales with the expansion of the Tor network. The more users operate Tor nodes, the higher the path diversity offered to DV, and the lower load from DV on individual nodes.

Easy adoption. The validation system requires no additional infrastructure to be set up for each vantage point. ValidaTor integrates seamlessly with the CAs and their existing infrastructure, and is easy to adopt into existing DV systems.

Acknowledgements

This work has been co-funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) SFB 1119.

References

[1] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal, "Using bgp to acquire bogus tls certificates," *HotPETS'17*, 2017.

[2] H. Birge-Lee, L. Wang, J. Rexford, and P. Mittal, "Sico: Surgical interception attacks by manipulating bgp communities," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 431–448.

[3] Sharon Goldberg, "The myetherwallet.com hijack and why it's risky to hold cryptocurrency in a webapp," https://medium.com/@goldbe/the-myetherwallet-com-hijack-and-why-its-risky-to-hold-cryptocurrency-in-a-webapp-261131fad278, 2018.

[4] P. Kacherginsky, "Celer Bridge incident analysis," 2022. [Online]. Available: https://www.coinbase.com/de/blog/celer-bridge-incident-analysis

[5] H. Birge-Lee, L. Wang, G. Cimaszewski, J. Rexford, and P. Mittal, "Attackers exploit fundamental flaw in the web's security to steal 2USD million in cryptocurrency," 2022. [Online]. Available: https://freedom-to-tinker.com/2022/03/09/attackers-exploit-fundamental-flaw-in-the-webs-security-to-steal-2-million-in-cryptocurrency/

[6] M. Brandt, T. Dai, A. Klein, H. Shulman, and M. Waidner, "Domain validation++ for mitm-resilient pki," in *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2060–2076.

[7] H. Birge-Lee, Y. Sun, A. Edmundson, J. Rexford, and P. Mittal, "Bamboozling certificate authorities with bgp," *SEC'18: Proceedings of the 27th USENIX Conference on Security Symposium*, pp. 833–849, 2018.

[8] K. Bhargavan, A. Delignat-Lavaud, and N. Kobeissi, "Formal modeling and verification for domain validation and acme," in *Financial Cryptography and Data Security: 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers 21*. Springer, 2017, pp. 561–578.

[9] G. H. Cimaszewski, H. Birge-Lee, L. Wang, J. Rexford, and P. Mittal, "How effective is multiple-vantage-point domain control validation?" in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5701–5718.

[10] T. Hlavacek, P. Jeitner, D. Mirdita, H. Shulman, and M. Waidner, "Behind the scenes of rpki," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1413–1426.

[11] T. Dai, H. Shulman, and M. Waidner, "Let's downgrade let's encrypt," *CCS '21: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1421–1440, 2021.

[12] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*. USENIX Association, 2004, pp. 21–21.

[13] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications." in *Usenix Security*, vol. 2013, 2013.

[14] C. Partridge and M. Allman, "Ethical considerations in network measurement papers," *Communications of the ACM*, vol. 59, no. 10, pp. 58–64, 2016.

[15] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu, "Rfc3647: Internet x. 509 public key infrastructure certificate policy and certification practices framework," 2003.

[16] X. Liu, X. Wang, and W. Chen, "Case study of dns server attacks: Exploits and countermeasures," in *Proceedings of the 2013 ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2013, pp. 79–84.

[17] M. Kranch and J. Bonneau, "Certificate transparency and its implications for the security of domain validation," in *Proceedings of the 2015 ACM Conference on Computer and Communications Security*, 2015, pp. 241–252.

[18] J. Zhang, M. Wu, K. Sun, and X. Du, "Privacy and security risks in domain validation for ssl certificates," *Journal of Network and Computer Applications*, vol. 65, pp. 84–96, 2016.

[19] J. Amann, N. Vallina-Rodriguez, C. Kreibich, and N. Weaver, "Mission accomplished? https security after diginotar," *Proceedings of the 2017 Internet Measurement Conference*, pp. 325–340, 2017.

[20] Q. Scheitle, O. Gasser, and G. Carle, "Measuring the impact of dnssec on dns and http performance," in *Proceedings of the 2018 ACM Conference on Internet Measurement*, 2018, pp. 1–14.

[21] J. Brian, A. Smith, and S. Zander, "An empirical study of the security of dnssec in the wild," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–25, 2019.

[22] T. Fiebig, Q. Scheitle, O. Hohlfeld, and G. Carle, "Something from nothing (there): Collecting global ipv6 dns open resolvers," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 963–979.

[23] Q. Scheitle, O. Gasser, T. Fiebig, and G. Carle, "Rise of the botivo: Evaluating the deployment of dane," in *Proceedings of the 2017 Internet Measurement Conference*, 2017, pp. 1–14.

[24] Internet Systems Consortium. (2024) Bind 9 security release and multi-vendor vulnerability handling, cve-2023-50387 and cve-2023-50868. Internet Systems Consortium. [Online]. Available: https://www.isc.org/blogs/2024-bind-security-release/

[25] H. Birge-Lee, J. Kao, A. Mirian, J. Rexford, and P. Mittal, "Towards improved domain validation in public key infrastructures," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 644–660.

[26] J. Frieß, H. Schulmann, and M. Waidner, "Crowdsourced distributed domain validation," in *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks, HOTNETS 2024, Irvine, CA, USA, November 18-19, 2024*. ACM, 2024, pp. 318–325.

[27] RIPE NCC. (2024) Ripe atlas. RIPE Network Coordination Centre. [Online]. Available: https://atlas.ripe.net

[28] J. Snijders. (2024) Nlnog ring. NLNOG. [Online]. Available: https://ring.nlnog.net/

[29] Bright Data. (2024) Proxy services. Bright Data. [Online]. Available: https://brightdata.com/proxy-types

[30] ExpressVPN. (2024) High-speed, secure & anonymous vpn service. ExpressVPN. [Online]. Available: https://www.expressvpn.com/

[31] P. F. Syverson, D. M. Goldschlag, and M. G. Reed, "Towards an analysis of onion routing security," in *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*. Springer, 2001, pp. 96–114.

[32] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic Certificate Management Environment (ACME)," RFC 8555, Mar. 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8555

[33] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Proceedings of Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[34] Electronic Frontier Foundation. (2024) Certbot. Electronic Frontier Foundation. [Online]. Available: https://certbot.eff.org/

[35] Let's Encrypt. (2024) Let's encrypt stats. Internet Security Research Group. [Online]. Available: https://letsencrypt.org/stats/

[36] The Tor Project. (2024) Traffic. The Tor Project. [Online]. Available: https://metrics.torproject.org/bandwidth-flags.html

[37] CAIDA. Routeviews prefix-to-as mappings (pfx2as) for ipv4 and ipv6. [Online]. Available: https://publicdata.caida.org/datasets/routing/routeviews-prefix2as/

[38] R. NCC. Ris raw data. [Online]. Available: https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data

[39] University of Oregon, "Route views project," http://bgplay.routeviews.org/, 2012.

[40] S. Pletinckx, T.-D. Nguyen, T. Fiebig, C. Kruegel, and G. Vigna, "Certifiably vulnerable: Using certificate transparency logs for target reconnaissance," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023.

[41] CAIDA. The caida as relationships dataset, 2024-05-01. [Online]. Available: https://www.caida.org/catalog/datasets/as-relationships/

[42] M. Brandt and H. Shulman, "Optimized bgp simulator for evaluation of internet hijacks," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2021.

[43] C. Cimpanu. (2021, May) Thousands of tor exit nodes attacked cryptocurrency users over the past year. [Online]. Available: https://therecord.media/thousands-of-tor-exit-nodes-attacked-cryptocurrency-users-over-the-past-year

[44] G. Koppen. (2021) Recent rejection of relays. [Online]. Available: https://lists.torproject.org/pipermail/tor-relays/2021-December/020048.html

[45] P. Winter, R. Ensafi, K. Loesing, and N. Feamster, "Identifying and characterizing sybils in the tor network," in *Proceedings of the 25th USENIX Security Symposium*, 2016.

A Hardware Requirements

Using `docker stats`, we track the CPU and memory usage of our container over the course of the experiments, in order to give an idea of the hardware required for deployment at scale. As shown in Figures 9 and 10, respectively, the container uses an equivalent of 5 - 8 CPU cores and 800 - 1000MB of memory. There is some dependence on the number of validators used, particularly for CPU usage, but overall the resource usage is fairly stable, both across time and number of validators. This allows fairly accurate extrapolation of the hardware requirements for a large-scale system.

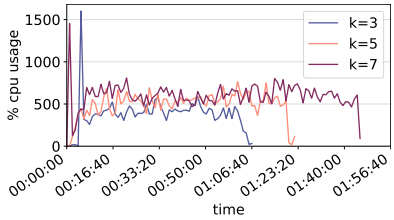


Figure 9: CPU usage per ValidaTor container over time.

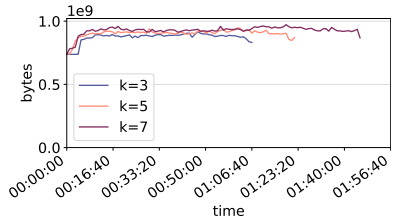


Figure 10: Memory per ValidaTor container over time.

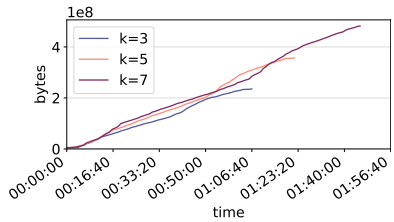


Figure 11: Cumulative network traffic per ValidaTor container over time.

From Address Blocks to Authorized Prefixes: Redesigning RPKI ROV with a Hierarchical Hashing Scheme for Fast and Memory-Efficient Validation

Zedong Ni[1,2,#], Yinbo Xu[1,3,#], Hui Zou[1,3], Yanbiao Li[1,3,*], Guang Cheng[2,4], and Gaogang Xie[1,3,*]

[1]Computer Network Information Center, Chinese Academy of Sciences
[2]School of Cyber Science & Engineering, Southeast University
[3]University of Chinese Academy of Sciences
[4]Purple Mountain Laboratories

Abstract

Route Origin Validation (ROV) with Route Origin Authorizations (ROAs), built on top of the Resource Public Key Infrastructure (RPKI), serves as the only formally standardized and production-grade defense mechanism against route hijackings in global interdomain routing infrastructures. However, the widespread adoption of RPKI has introduced escalating scalability challenges in validating high volumes of route messages against massive ROA entries.

In this paper, we attribute the performance bottleneck of existing ROV schemes to their underlying validation model, where the route is matched against rules in the form of address blocks. To overcome this bottleneck, we propose the *Authorized Prefix (AP) model* that enables validation at the prefix granularity, and redesign RPKI ROV based on this new model with a hierarchical hashing scheme named h^{2}ROV. Extensive evaluations verify h^{2}ROV's superiority over state-of-the-art approaches in IPv4, with a speedup of 1.7× $\sim$ 9.8× in validation and a reduction of 49.3% $\sim$ 86.6% in memory consumption. System emulations using real-world network topologies further demonstrate h^{2}ROV confines its impact to routing convergence to below 8.5% during update burst events, while reducing ROV-induced delays by 30.4% $\sim$ 64.7% compared to existing solutions.

1 Introduction

1.1 BGP Security and Route Origin Validation

The Border Gateway Protocol (BGP) [1] is a fundamental building block of the Internet, which enables reachability information exchange among autonomous systems (ASes). Despite its critical role in maintaining global connectivity, BGP exhibits inherent vulnerabilities to malicious exploitation, particularly route hijacking attacks that can propagate invalid routing announcements across AS boundaries, causing persistent and large-scale disruptions to the global Internet.

To mitigate these threats, Route Origin Validation (ROV) has emerged as the sole standardized and production-ready security mechanism. Built upon the Resource Public Key Infrastructure (RPKI) [2], ROV utilizes cryptographically signed Route Origin Authorizations (ROAs) to establish a trust anchor for BGP route origination. Network operators implement ROV through router-side verification processes that cross-check advertised BGP routes against RPKI-validated ROAs, thereby identifying and filtering unauthorized origin AS claims.

1.2 Concerns about the Efficiency of ROV

While ROV significantly enhances the security of BGP route origination, its effectiveness remains constrained by incomplete global RPKI adoption. Notably, despite ROAs covering over 50% of BGP route prefixes, only approximately 12% of ASes actively enforce ROV in production networks [3]. This adoption gap persists even among operators with deployed ROAs, highlighting systemic barriers beyond mere infrastructure readiness. A longitudinal study [4] identifies operational efficiency concerns as the primary deterrent to ROV adoption.

ROV introduces two critical operational challenges for routers. First, the validation process imposes per-route computational overhead, requiring real-time verification of ROAs against BGP routes. Second, the sequential validation of bulk routing updates compounds latency, directly impacting BGP convergence—a key performance metric for global routing stability. Our experimental analysis reveals that state-of-the-art ROV implementations increase BGP convergence time by 8.0% $\sim$ 25.6% under update bursts[1], a frequent phenomenon driven by recurring route anomalies (e.g., misconfigurations and hijacks). Worse still, inefficient ROV introduces a new attack vector, enabling adversaries to overwhelm ROV-enforcing routers by amplifying validation overhead [6].

# The first two authors contributed equally to this work. Zedong Ni participated in this study during his master's degree program at the Computer Network Information Center of the Chinese Academy of Sciences.

* Corresponding authors: lybmath@cnic.cn and xie@cnic.cn.

[1]BGPMon [5] shows that route anomalies occur frequently in recent years, which may lead to update bursts.

Another concern arises from ROV's escalating storage demands as RPKI adoption increases. Our empirical analysis demonstrates that contemporary ROV implementations exhibit memory footprints approaching parity with the core Routing Information Base (Table 3, Appendix D.1). This substantial overhead originates from two fundamental requirements: 1) maintaining synchronized replicas of cryptographically validated ROA entries, and 2) supporting specialized lookup structures for accelerated route validation processes. These combined components directly compete with allocation priorities for essential routing functions, creating an operational paradox where network operators face pressure to disable ROV - despite its security benefits - to preserve baseline routing stability.

1.3 Prior Arts and Limitations

Existing research has explored two dominant approaches to optimize ROV performance: hash-based and trie-based validation architectures. Hash-based methods [7] organize ROAs in hash tables, achieving memory efficiency by compressing prefix-to-AS mappings. Nevertheless, this approach increases validation latency, as resolving super/prefix containment requires multiple hash probes. Conversely, trie-based implementations [6–8] improve validation speed by structuring address blocks into hierarchical prefix trees, achieving O(log n) lookup times. However, they introduce higher memory overhead due to trie node proliferation.

Although existing optimizations partially mitigate ROV inefficiencies, our large-scale emulations demonstrate that both trie- and hash-based architectures exhibit critical scalability limitations under projected RPKI adoption trajectories. Specifically, the accelerating coverage of BGP routes by ROAs will substantially increase the proportion of ROV-validated routes. Counterintuitively, validation latency in state-of-the-art systems sharply increases with valid route volume. This divergence leads to an operational dilemma: improved security via RPKI adoption directly degrades real-time route processing capabilities.

1.4 Key Insights and Our Contributions

Regarding the aforementioned challenge in ROV with existing solutions, we conduct a comprehensive investigation to identify the root cause of their performance bottlenecks. Our systematic analysis reveals that the fundamental limitation stems from inherent deficiencies in the conventional validation model [9]. To address this critical issue, we propose a groundbreaking *Authorized Prefix (AP)* validation framework in Section 3, a novel paradigm that establishes two pivotal advancements: (1) superior operational efficiency compared to the traditional model especially when valid route volume is large, and (2) guaranteed equivalence in validation accuracy through formal verification.

Then, in Section 4, we present `h²ROV`, a hierarchical hashing scheme for fast and memory-efficient ROV based on the *AP model*. It employs two hierarchical hash tables to manage authorized prefixes extracted from ROAs with bitmap encoding, enabling ROV at prefix granularity. We further propose two optimizations to improve its validation performance in IPv4 and IPv6 respectively. In addition, we validate `h²ROV`'s practical applicability through a reference implementation integrated with two widely-deployed open-source software routers, `FRRouting`[2] and `BIRD`[3], in Section 5.

At last, in Section 6, we evaluate `h²ROV` using real-world RPKI repositories, BGP routing tables and update traces from operational networks. Compared to state-of-the-art ROV solutions, `h²ROV` reduces memory usage by 49.3% $\sim$ 86.6% and accelerates IPv4 route validation by 1.7 $\sim$ 9.8 times. System emulations using real-world network topologies further demonstrate `h²ROV` mitigates ROV-induced routing convergence delays by up to 64.7%, outperforming conventional validation architectures under routing instability scenarios.

2 Background and Motivation

2.1 Route Origin Authorizations

As defined in RFC 9582 [10], an ROA is a cryptographically signed object, containing an ASN and a sequence of `ipAddrBlocks`. An `ipAddrBlock` encodes a tuple `(IP Prefix, maxLength)`, encompassing all sub-prefixes of `IP Prefix` with prefix lengths up to the specified `maxLength`. We call such a tuple as an *address block (AB)*. The Replying Party (RP) retrieves ROAs from the RPKI repository and validates them locally. Then, all validated ROAs are distributed to routers via the RPKI-to-Router (RTR) protocol [11]. Routers receive these ROAs and extract the key authorization information to guide the ROV process. We define the extracted authorization information as ROV rules. Additionally, to clarify terminology, in this paper, references to ROAs specifically denote an AB paired with its authorized AS, represented by the tuple `(IP Prefix, maxLength, ASN)`, abbreviated as `(pfx, ml, ao)`.

2.2 ROV Model with Address Blocks

Current ROV schemes validate BGP routes using a list of ROV rules, where each ROV rule, extracted from received ROAs, can be represented as a tuple *(pfx, ml, ao)* comprising an *AB* and its authorized AS [9]. We define the *AB model* as the ROV model that uses AB-based ROV rules. Validity states of the *AB model* is defined as follows.

Definition 1 *Let r denote a route, represented as a tuple (rp, ro), where rp is the route prefix and ro is the origin ASN.*

[2] https://frrouting.org/
[3] https://bird. network.cz/

Prefix-to-ASN Mappings	
192.0.2.0/25, AS X	192.0.2.64/31, AS Z
192.0.2.64/26, AS Y	192.0.2.64/32, AS Z
192.0.2.64/27, AS Y	192.0.2.65/32, AS Z
192.0.2.96/27, AS Y	

Figure 1: Prefix-to-ASN Mappings.

List of ROV rules
192.0.2.0/25, 25, AS X
192.0.2.64/26, 27, AS Y
192.0.2.64/31, 32, AS Z

Figure 2: List of ROV rules.

Definition 2 *Let $|p_i|$ denote the prefix length of p_i. $p_i \subseteq p_j$ means p_i is a sub-prefix of p_j.*

Definition 3 ***Validity states:*** *Given a list of ROV rules RL and a route r, the validity state can be valid, invalid, or notFound.* ***Valid*** *means $\exists R \in RL$, R* ***matches*** *r.* ***Invalid*** *means $(\exists R \in RL$, R* ***covers*** *$r) \wedge (\forall R \in RL$, R* ***covers*** *$r \implies$ R does not* ***match*** *r).* ***NotFound*** *means $\forall R \in RL$, R does not* ***cover*** *r.*

Definition 4 ***Cover/Match:*** *Given an ROV rule R and a route r, R* ***covers*** *r means $r.rp \subseteq R.pfx$. R* ***matches*** *r means R* ***covers*** *$r \wedge |r.rp| \leq R.ml \wedge r.ro = R.ao$.*

In the *AB model*, two conditions are defined: *ExistCover (EC)*, which finds an ROV rule that covers route *r*, and *CheckMatch (CM)*, which examines whether the covering ROV rule *R* satisfies $R.ml \geq |r.rp| \wedge R.ao = r.ro$. If the *EC* condition fails, the ROV result is *notFound*. Otherwise, if both *EC* and *CM* conditions are met, the ROV result is *valid*. In this case, the *EC* and *CM* conditions are performed α times, where α ranges from 1 to *n*, with *n* being the number of ROV rules covering *r* in *RL*. If the *EC* condition is satisfied but the *CM* condition is not, the ROV result is *invalid*, and both conditions are performed *n* times. We abstract a decision tree from the *AB model*, namely the *CoverFirst (CF) tree*, as shown in Figure 3.

2.3 Hanging ROA and Bitmap Encoding

Hanging ROA (`h-ROA`) is a bitmap-based encoding scheme for ROAs [12]. It encodes authorized prefixes within ROAs into split sub-trees. In this way, an ROA is transformed into an ASN and a list of encoded sub-trees, each represented as a tuple `(id, bm)`. The `id` uniquely identifies each sub-tree, derived from the root prefix of that sub-tree. The operation of computing `id` can be defined as `ID(p, n)`, which concatenates a set bit ('1') with the first `n` bits of prefix `p` and converts the resulting binary string to a decimal value. The `bm` records authorized prefixes within the sub-tree. Each bit represents a prefix, where '1' indicates authorization and '0' otherwise. Due to its design of encoding authorized prefixes into bitmaps, `h-ROA` supports ROA updates at prefix level.

2.4 Motivation and A Running Example

However, the ROV with *AB model* is inherently limited. We use an example to illustrate the limitations of the *AB model*. There are seven prefixes assigned to three ASes, mapped in Figure 1, generating three ROV rules shown in Figure 2. For

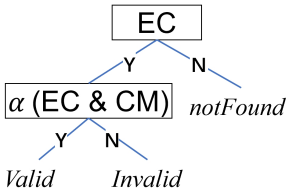


Figure 3: Decision tree of the *AB model*.

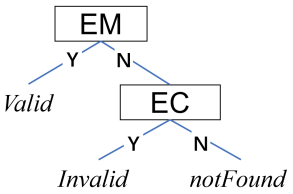


Figure 4: Decision tree of the *AP model*.

a route `(192.0.2.64/32, AS Z)`, the ROV-enforcing router determines its validity state as *valid*. In this example, the router performs three queries, sequentially checking whether each ROV rule covers the route prefix and ultimately identifying a matching ROV rule.

The *AB model*'s performance bottleneck lies in its coarse-grained address block management. The bitmap-based encoding enables fine-grained management of authorized prefixes and provides valuable insights for improving ROV. We propose the *Authorized Prefix (AP) model*, which uses authorized prefixes for route validation and builds on bitmap-based encoding to achieve fine-grained management of ROV rules. Through careful design and optimization, the *AP model* can overcome the performance bottlenecks of the *AB model*.

3 ROV Model with Authorized Prefixes

3.1 Model Definition

Instead of encoding an ROV rule as an *AB* and its authorized AS, the *AP model* maintains ROV rules at prefix granularity. Each ROV rule is encoded as a tuple *(pfx, ao)* comprising a prefix and its authorized AS. Validity states of the *AP model* remain the same as those of the *AB model*, while the definition of Cover/Match differs, as defined below.

Definition 5 ***Cover/Match:*** *Given an ROV rule R and a route r, R* ***covers*** *r means $r.rp \subseteq R.pfx$. R* ***matches*** *r means $r.rp = R.pfx \wedge r.ro = R.ao$.*

In the *AP model*, we define two conditions: *ExistCover (EC)*, which finds an ROV rule that covers route *r*, and *ExistMatch (EM)*, which finds an ROV rule that matches *r*. If the *EM* condition is satisfied, the ROV result is *valid*. Otherwise, the ROV result is *invalid* or *notFound*, depending on whether the *EC* condition is met. Both conditions are executed at most once. We abstract a decision tree from the *AP model*, namely the *MatchFirst (MF) tree*, as shown in Figure 4.

3.2 Cost Analysis of ROV

Cost of the *AB model*. To analyze the validation cost of the *AB model*, we define the execution costs of *EC* and *CM* as X_1 and Y_1, as the complexities of these two conditions obviously differ. Then, the validation costs determining ROV

results are $\alpha X_1 + \alpha Y_1$, $nX_1 + nY_1$, and X_1 for *valid*, *invalid*, and *notFound* routes, respectively. Finally, we formalize the minimal cost of the *AB model* according to the *CF tree* based on the distribution across three validity states (with $\alpha = 1$ and $n = 1$ for minimal cost):

$$\begin{aligned} cost(CF) &= P_1(X_1 + Y_1) + P_2(X_1 + Y_1) + P_3 X_1 \\ &= X_1 + (1 - P_3)Y_1 \end{aligned} \tag{1}$$

where P_1, P_2, and P_3 are the proportions of *valid*, *invalid*, and *notFound* routes, respectively, and their total equals 100%. Clearly, as RPKI deployment increases, the proportion of *notFound* routes decreases correspondingly. Therefore, the validation cost of the *AB model* increases.

Cost of the *AP model*. The costs of executing the *EC* and *EM* conditions are defined as X_2 and Y_2. Thus, the validation costs determining ROV results are Y_2, $X_2 + Y_2$, and $X_2 + Y_2$ for *valid*, *invalid*, and *notFound* routes, respectively. Finally, we formalize the cost of the *AP model* using the *MF tree* based on the distribution across three validity states:

$$\begin{aligned} cost(MF) &= P_1 Y_2 + P_2(X_2 + Y_2) + P_3(X_2 + Y_2) \\ &= (1 - P_1)X_2 + Y_2 \end{aligned} \tag{2}$$

The validation cost of the *AP model* is affected by P_1. As the proportion of *valid* routes increases, the validation cost decreases.

Cost comparison. We compare the validation costs of the two ROV models based on the previous analysis. In both models, the *EC* condition tries to find one or more ROV rules that cover a given route. This can be regarded as the *subset rank problem*, which has a time complexity of $O(\log n)$, where n is the number of ROV rules [13]. Therefore, we assign the same time consumption to X_1 and X_2, denoted as $X_1 \equiv X_2 \equiv x$. The *CM* and *EM* conditions, on the other hand, can be performed in constant time, so we define $Y_1 \equiv Y_2 \equiv y$. Furthermore, it is clear that $x \gg y$. With these assumptions, we can formalize the difference between the minimum cost of $cost(CF)$ (where $n = 1$ and $\alpha = 1$) and the maximum cost $cost(MF)$ as follows:

$$\begin{aligned} \Delta(cost) &= cost_{min}(CF) - cost_{max}(MF) \\ &= X_1 + (1 - P_3)Y_1 - ((1 - P_1)X_2 + Y_2) \\ &= P_1 x - P_3 y \end{aligned} \tag{3}$$

Currently, $P_1 > P_3$, so the difference in cost between the two models is greater than 0, indicating that the maximum cost of the *AP model* is lower than the minimum cost of the *AB model*. Moreover, as RPKI gets more widely deployed, the proportion of *valid* routes (P_1) will increase and that of *notFound* routes (P_3) will decrease. This will further widen the cost difference between the two models.

3.3 Equivalence in Validation Accuracy

This subsection presents the proof that the *AP model* is equivalent to the *AB model* in terms of ROV outcomes.

Definition 6 *Given an ROV rule R (pfx,ml,ao), the set of prefixes contained in R is defined as:* $AP_{set}(R) = \{p \mid p \subseteq R.pfx \wedge |p| \leq R.ml\}$.

Definition 7 *Given a list of ROV rules RL_{AB} under the AB model, the AP model generates its list of ROV rules RL_{AP} as follows:* $RL_{AP} = \{(pfx, ao) \mid pfx \in AP_{set}(R) \wedge ao = R.ao, \forall R \in RL_{AB}\}$

Theorem 1 *Given an ROV rule R (pfx, ml, ao) and a route prefix rp,* $rp \nsubseteq R.pfx \iff \forall p \in AP_{set}(R), rp \nsubseteq p$.

The proof of this theorem is provided in Appendix B.

Theorem 2 *For a given route r (rp, ro), the ROV result under the AB model is identical to that under the AP model.*

By Definition 3, if the ROV result for route *r (rp, ro)* is *notFound* under the *AB model*, then $\forall R \in RL_{AB}, rp \nsubseteq R.pfx$. According to Theorem 1, we can infer that $\forall p \in AP_{set}(R), rp \nsubseteq p$ for all $R \in RL_{AB}$. Combining with Definition 7, $\forall R \in RL_{AP}, rp \nsubseteq R.pfx$, the ROV result for route r under the *AP model* is also *notFound*, and conversely.

If the ROV result for r is *valid* under the *AB model*, then $\exists R \in RL_{AB}, rp \subseteq R.pfx \wedge |rp| \leq R.ml \wedge ro = R.ao$. Let this ROV rule be *MR*. According to Definition 6, $rp \in AP_{set}(MR) \wedge ro = MR.ao$. Since $MR \in RL_{AB}$, by Definition 7, we deduce that $(rp, ro) \in RL_{AP}$. Consequently, $\exists R \in RL_{AP}, R.pfx = rp \wedge R.ao = ro$. Hence, the ROV result under the *AP model* is also *valid*, and conversely.

If the ROV result for r is *invalid* under the *AB model*, it is neither *valid* nor *notFound*. Since the *AB* and *AP models* agree on classifying routes as *valid* or *notFound*, the *AP model* also deem it neither *valid* nor *notFound*. Hence, the result under the *AP model* is *invalid*, and conversely.

In conclusion, given identical ROAs and routes, the validation results produced by the *AB model* and the *AP model* are equivalent.

4 h²ROV: Hierarchical Hashing-Based ROV

In this section, we introduce the basic version of our proposed ROV scheme, h²ROV. Subsequently, we introduce two optimizations specifically tailored for IPv4 and IPv6.

4.1 Maintaining ROV rules with Bitmaps

Instead of refining ROV under the limitations of the *AB model*, we propose a novel ROV scheme, h²ROV, which performs ROV based on the *AP model*. To achieve this, h²ROV employs the basic version of h-ROA for managing ROV rules at the prefix level, which sets the hanging levels as multiples of 5. Figure 5 illustrates how h²ROV maintains the ROV rules at the prefix level and performs ROV based on these finer-granularity

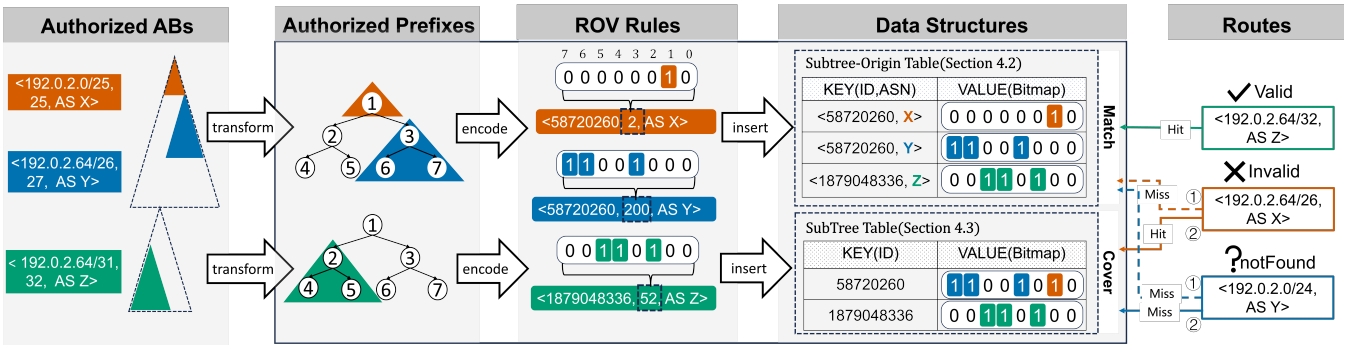


Figure 5: In h²ROV scheme, when the router receives ROAs, it first encodes them into a series of ROV rules and then inserts them into the corresponding data structures.

authorizations. For example, the ROA (192.0.2.64/26, 27, AS Y) (blue square) authorizes three prefixes for AS Y. h²ROV first locates the sub-tree containing these prefixes, rooted at 192.0.2.0/25. The sub-tree's id is calculated as 58720260, derived from the binary string formed by concatenating a set bit ('1') with the first 25 bits of 192.0.2.0. Then, h²ROV encodes the three authorized prefixes into the bitmap 11001000 (only the last 8 bits are shown, as the higher bits are 0). Bits 3, 6, and 7 are set to 1, corresponding to these prefixes. Similarly, ROAs (192.0.2.0/25, 25, AS X) (orange square) and (192.0.2.64/31, 32, AS Z) (green square) are encoded into (58720260, 00000010, AS X) and (1879048336, 00110100, AS Z). In summary, when routers receive ROAs, h²ROV transforms them into ROV rules, which are represented as (id, bm, ASN) tuples.

4.2 Matching the Route

Given a route (rp, ro), h²ROV first calculates the sub-tree's id as ID (rp, |rp|). If a bitmap is found in the SOT for (id, ro), h²ROV calculates the index of rp in the sub-tree as i and checks whether the i-th bit in the bitmap is set to 1. If the bit is set, it indicates that rp is authorized by an ROA for ro, and the route is *valid*. By leveraging the SOT, this matching process achieves a time complexity of O(1).

As shown in Figure 5, the router encodes the fetched ROAs into three ROV rules: (58720260, 00000010, AS X), (58720260, 11001000, AS Y), and (1879048336, 00110100, AS Z). These entries are then inserted into the SOT as ((58720260, AS X), 00000010), ((58720260, AS Y), 11001000), and ((1879048336, AS Z), 00110100). For a received route (192.0.2.64/32, AS Z), h²ROV first calculates the sub-tree's id as 1879048336. It then identifies the index of the route within the sub-tree as 4. Next, h²ROV queries the SOT using the key (1879048336, AS Z) and retrieves the corresponding bitmap 0011**0**0100. Finally, h²ROV determines the route's validity state by checking whether the 4th bit in this bitmap is set to 1. In this case, the bit is set, confirming that the route is *valid*.

4.3 Covering the Route Prefix

For *valid* routes, the SOT enables direct matching. However, for *invalid* or *notFound* routes, where SOT alone cannot confirm status, further validation is needed. We deploy a secondary hash table STT to locate ROV rules covering the route prefix. The STT utilizes sub-tree's id as key and stores a bitmap union of all authorized prefixes within corresponding sub-tree.

When a route (rp, ro) fails SOT matching, h²ROV retrieves the bm in the STT using id=ID(rp, |rp|). If there exists an authorized prefix covering rp marked in the bm, the ROV result is *invalid*. If no covering prefix is marked in the bm or no bm is found for the given id, h²ROV proceeds to query the sub-trees associated with the super-prefixes of rp. This process continues until either an authorized prefix covering rp is found (resulting in *invalid*), or the root of the IP prefix trie is reached without a match (resulting in *notFound*).

As shown in Figure 5, the router inserts the three ROV rules into the STT without their authorized ASes. These entries are (58720260, 00000010), (58720260, 11001000), and (1879048336, 00110100). Since the first two entries share the same id, their bms are merged with a bitwise OR, resulting in an updated entry with the key 58720260 and the value 11001010 (i.e., 11001000 OR 00000010). For route (192.0.2.64/26, AS X), h²ROV calculates the id of the sub-tree as 58720260, and determines its index within the sub-tree as 3. After an SOT query fails, h²ROV proceeds to query the STT using the id as the key. The merged bitmap 1100**1**010 is retrieved, and since the 3rd bit is set to 1, the ROV result for this route is *invalid*. This outcome indicates that the rp is covered but not matched by an ROV rule; otherwise, a matching ROV rule would have been found in the SOT. In contrast, route (192.0.2.0/24, AS Y) receives a *notFound* verdict due to missing matches in both SOT and STT.

4.4 Basic Design of h²ROV

The h²ROV validation process operates through two distinct phases: **matching the route** and **covering the route prefix**.

Algorithm 1: Validate a given route with h²ROV

Input : the Subtree-Origin Table: SOT, the SubTree Table: STT, the Route: (rp,ro)
Output : the ROV result of the given route: res

```
1  res ← NotFound;
2  id, pos ← calculate_ID_AND_POS(rp);
3  if (bm1 = SOT[id + ro] and bm1[pos] == 1) then
4  |   res ← Valid;
5  |   return res;
6  end
7  while id > 0 do
8  |   if bm2 = STT[id] and ExistCover(bm2, pos) then
9  |   |   res ← Invalid;
10 |   |   return res;
11 |   end
12 |   id >>= 5;
13 end
14 return res;
```

First, h²ROV determines whether a route is *valid* by querying the SOT for matching ROV rules (in Section 4.2). If the SOT query fails, h²ROV recursively searches the STT for ROV rules that cover the route prefix to determine whether the route is *invalid* or *notFound* (in Section 4.3).

The pseudo code of h²ROV is shown in Algorithm 1. In the worst case, h²ROV requires one SOT query and $\left\lceil \frac{|rp|}{5} \right\rceil$ STT queries to validate a given route (rp, ro). Therefore, the time complexity of h²ROV for validation is $O(|rp|)$.

4.5 Optimization for IPv4 Route Validation

Obviously, the performance bottleneck of h²ROV arises from repeated lookups in the STT, termed the backtracking issue. This issue occurs as a bitmap only encode authorized prefixes within the sub-tree, leaving coverage by other sub-trees uncertain. To tackle this challenge, we propose optimizations for IPv4 and IPv6 scenarios to reduce backtracking.

In the IPv4 scenario, to expedite the process of checking whether `rp` is covered, we introduce additional bitmap arrays to mark covered sub-tree root prefixes. For root prefixes at hanging levels 5, 10, 15, and 20, we construct bitmap arrays B_i of 2^i bits, respectively. For instance, B_5 has 2^5=32 bits and is initially set to 0. These bitmap arrays are referred to as the **Level Bitmap (LB)**. For each root prefix `p` at these levels, we use the decimal value of first `|p|` bits of `p` as its index in B_i. For example, the index of 8.0.0.0/5 in B_5 is 1(0b00001). When a prefix is authorized, we mark all covered root prefixes at these hanging levels with 1. For a given route `(rp, ro)`, if h²ROV fails to match the route in the SOT, it uses the LB to check if `rp` is covered by other authorized prefixes, thus avoiding multiple STT queries. This optimization enables efficient processing of routes with prefix lengths of 24 or shorter. For routes whose prefix lengths exceed 24, h²ROV requires up to three STT queries and one LB access in the worst case. This is because sub-trees rooted at hanging levels 20, 25, and 30 may contain authorized prefixes covering `rp`, while the LB only records the coverage status up to hanging level 20.

To balance validation speed and memory usage, we refrain from constructing bitmap arrays for hanging levels 25 and 30 for two main factors. First, the vast majority of IPv4 prefixes have lengths of 24 or shorter [14]. Second, constructing bitmap arrays for hanging levels 25 and 30 would demand 32MB and 1GB of memory respectively. By limiting LB construction to hanging levels 5, 10, 15, and 20, h²ROV incurs only an additional 1MB of memory overhead while reducing time complexity of validation to $O(1)$.

4.6 Optimization for IPv6 Route Validation

In the IPv6 scenario, for a route with a prefix length of 128, h²ROV requires 26 ($\left\lceil \frac{128}{5} \right\rceil$) STT queries in the worst case. Despite applying LB, the number of STT queries merely decreases to 22 ($\left\lceil \frac{128-20}{5} \right\rceil$), which is a limited improvement in validation performance. In fact, most IPv6 routes have prefix lengths between 32 and 48 [15]. However, constructing bitmap arrays for hanging levels 30 and 45 is impractical due to the excessive storage requirements. Therefore, a more efficient optimization tailored to IPv6 scenario is necessary.

We introduce **Cover Flag (CF)**, which is the 0th bit of the `bm`, to indicate whether at least one super-prefix covers the sub-tree. The CF enables early termination of STT backtracking by instantly checking coverage upon retrieving `bm`. However, even with the CF effectively reducing the number of STT backtracking queries, 26 queries can still be required in the worst case when intermediate entries are missing from the STT. To further optimize the validation performance, we incorporate a binary search strategy. However, the binary search can only be performed if all intermediate sub-trees along the path from the root of the sub-tree containing rp to the root of the IPv6 address prefix trie exist in the STT. Therefore, before performing binary search, we ensure that all intermediate sub-trees exist in the STT by inserting placeholder entries to fill in the missing ancestor sub-trees. These placeholder entries are initialized with bitmaps set to 0, ensuring structural continuity for binary search.

For binary search on a route `(rp,ro)`, two pointers are initialized: the left pointer (`lft`) to 0 and the right pointer (`rt`) to `|rp|`. The middle pointer (`mid`) is calculated as `(lft+rt)/2`. If the bitmap corresponding to `id = ID(rp, mid)` is absent in the STT, `mid` is recalculated as `(lft+mid)/2`; otherwise, if the corresponding bit in the obtained bitmap for this route is set to 1 or the CF of the obtained bitmap is set to 1, the route is determined to be *invalid* and the search terminates. If neither condition is satisfied, `mid` is updated to `(mid+rt)/2`

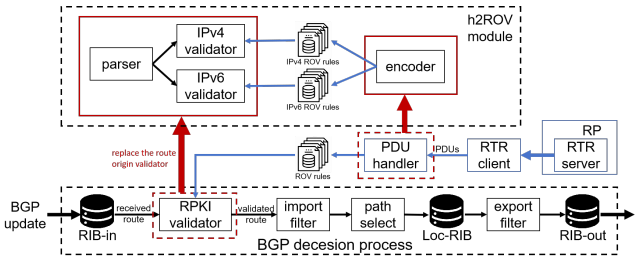


Figure 6: The implementation of the h²ROV scheme in the software router.

and the query to the STT continues. This approach reduces the time complexity from $O(|rp|)$ to $O(log(|rp|))$.

5 Implementation with Software Routers

5.1 System Design

We implemented the h²ROV module in two open-source software routers, FRRouting [16] and BIRD [7]. The module consists of three components: *encoder*, *parser*, and *validator*, whose interactions with the original components of the software router are illustrated in Figure 6. The *encoder* identifies whether ROAs received from the RP are IPv4 or IPv6 by examining the Type field in the RTR Prefix PDU, encodes them into ROV rules, and maintains separate storage for each address family. The *parser* processes BGP updates from neighbors, distinguishing IPv4 and IPv6 routes via the Address Family Identifier field. It extracts essential information for ROV, including the prefix from the Network Layer Reachability Information (NLRI) attribute and the origin AS (the rightmost AS in the AS Path attribute). The *validator* performs origin validation on parsed routes using the encoded ROV rules, applying the appropriate ROV algorithm based on the prefix type (IPv4 or IPv6). Validated routes, along with their ROV results, are forwarded to the *path selection* module.

h²ROV validator. As discussed in Section 6.1, h²ROV excels at validating IPv4 routes but is slightly less efficient for IPv6 routes. This limitation has a negligible impact on overall performance due to two factors. First, IPv4 updates occur $1.8 \sim 3$ times more frequently than IPv6 updates, reducing the proportion of IPv6 validation tasks [17]. Second, expanding RPKI deployment will increase *valid* routes, including those for IPv6, enhancing the validation performance of h²ROV beyond existing ROV schemes. Therefore, we have integrated IPv4 and IPv6 optimizations of h²ROV into our *validator*, ensuring robust performance.

Modifications to the software router. As shown in Figure 6, the software router handles two distinct data flows: the routing message flow (black arrows) and the ROA data flow (blue arrows). The integration of the h²ROV module replaces the original *ROV validator* component within the *RPKI validator* with a *parser* and *validator* for routing messages, and the original *PDU handler* with an *encoder* for received ROAs. This design limits modifications to a minimal scope, ensuring backward compatibility and the integrity of the software router's core functionality.

5.2 h²ROV with Wide ROAs

We define δ as the difference between the maxLength and the prefix length of an ROA. The number of prefixes authorized by an ROA is $2^{(\delta+1)}-1$. With hanging levels as multiples of 5, each sub-tree can encode 31 prefixes. In the best case, an ROA generates $\left(\left\lceil \frac{(2^{(\delta+1)}-1)}{31} \right\rceil\right)$ ROV rules.

Consequently, an ROA with a large δ generates numerous ROV rules. To avoid excessive memory consumption, ROAs with δ exceeding a predefined threshold are classified as *wide ROAs* and stored in the **Wide ROA Trie (WRT)**, a Patricia trie-based structure. The threshold values are chosen to optimize the validation performance and storage overhead in h²ROV (details in Section 6.2.1). WRT allows direct insertion of wide ROAs, bypassing ROV rule encoding. Since the authorized prefixes in wide ROAs are not recorded in the SOT, h²ROV queries the WRT whenever no matching ROV rule is found in the SOT during route matching.

5.3 ROA Update for h²ROV

We introduce the ROA update process for h²ROV in this subsection. For a normal ROA, the first step is to encode it into ROV rules as described in Section 4.1, followed by inserting or deleting these entries in the SOT and STT. For IPv4 ROAs, the LB needs to be updated based on the authorized prefixes contained in the ROAs. For IPv6 ROAs, the process involves checking whether encoded sub-trees from the ROA are covered by existing authorized prefixes and identifying which existing authorized prefixes are covered by the newly encoded sub-trees. Based on these checks, the corresponding CFs should be updated. Wide ROAs are inserted or deleted at the appropriate nodes in the WRT, with their prefixes encoded for insertion or deletion in the STT. Additionally, h²ROV uses a reference count to track bit marking frequency, preventing erroneous deletions when a bit is marked multiple times.

6 Evaluation

In this section, we conduct extensive experiments to evaluate the performance of h²ROV. Each experiment is run 100 times, and the average results are reported [4].

[4] A detailed breakdown of the 100 experimental runs is provided in Appendix D.2.

Table 1: The datasets used for evaluation.

Name	Date/Time Period	Source	Type
ROA_2024	2024/8/25	RIPE NCC	real-trace ROAs
ROA_13Y	2012/1/1∼2024/1/1	RIPE NCC	real-trace ROAs
BGP_RRC00	2024/8/25	RIS RRC00	real-trace BGP updates
BGP_13Y	2012/1/1∼2024/1/1	RIS RRC00	real-trace BGP updates
BGP_20C	2024/8/25	RIS 20 collectors	real-trace BGP updates
BGP_ROV	N/A	N/A	randomly-generated BGP updates
BGP_IP46	2024/8/25	RIS 20 collectors	randomly-selected BGP updates
BGP_SIZE	2024/8/25	RIS 20 collectors	randomly-selected BGP updates

Table 2: The experimental environments.

Experimental &System Evaluation	CPU	AMD EPYC 7742(2.2GHz, 128cores)
	RAM	DRAM 128G
	NIC	BCM57412 NetXtreme-E 10Gb
Emulation Platform	CPU	ARM Kunpeng-920 (344MHz, 128cores)
	RAM	RAM 560GB
	NIC	Intel Corporation 82599ES 10-Gigabit

6.1 Methodology

Evaluated schemes. We evaluate our proposed ROV scheme, h²ROV, and all existing ROV schemes mentioned in Section 7 except for *x*BGP. These schemes are represented by their core data structures namely LPFST (RTRLib), HT (BIRD), HT+PT (BIRD), and Patricia (BGP-SRx).

Data sets. We evaluate h²ROV using two types of datasets: ROA and BGP update. ROA datasets, sourced from RIPE NCC [18], include the dataset ROA_13Y, comprising 13 annual sets collected on January 1st from 2012 to 2024 for scalability experiments, and the dataset ROA_2024, collected on August 25, 2024, for other experiments. For BGP update datasets, we use three categories: real-trace, randomly-generated, and randomly-selected updates. Real-trace BGP updates are obtained from RIPE Routing Information Service (RIS) [19] collectors. The dataset BGP_20C comprises BGP updates from 20 RIPE RIS collectors on August 25, 2024. The dataset BGP_13Y contains annual RRC00 BGP updates on January 1st from 2012 to 2024 for scalability experiments. The dataset BGP_RRC00 comprises BGP updates from RRC00 on August 25, 2024. Furthermore, based on the dataset ROA_2024, we generate the dataset BGP_ROV by varying the proportion of randomly generated BGP updates corresponding to each ROV result (*valid*, *invalid*, *notFound*) from 0% to 100% in 10% increments, ensuring the sum of proportions equals 1. This results in 66 sets of BGP updates. The dataset BGP_IP46 consists of 11 sets with varying IPv4/IPv6 ratios, proportionally sampled from BGP_20C. The dataset BGP_SIZE includes 3 sets of BGP updates with sizes of 10K, 100K, and 1M, randomly sampled from BGP_20C. For further details, refer to Table 1 and Appendix E.

Performance metrics. For experimental evaluation, we introduce three key performance metrics to compare ROV schemes. First, validation speed is measured as the average number of routes validated per second, in million validations per second (MVPS). Second, memory consumption is measured in megabytes (MB). Lastly, update performance is measured as the average time taken to update an ROA. For the system implementation, we also introduce three performance metrics to evaluate the impact of ROV schemes. First, the average proportion of delay introduced by ROV in BGP decision process under normal router operation is evaluated. Then, the impact of ROV on BGP update processing speed during an update burst is measured by the average packet processing delay per second. Finally, the impact of ROV deployment on BGP convergence time is measured.

Experimental environment. For system evaluation, a small-scale RPKI and BGP testbed is constructed, with Routinator as the RP and FRRouting and BIRD as ROV-enforcing routers (abbreviated as *validators*). Furthermore, a BGP sender and receiver, both modified from FRRouting, are deployed. The sender replays BGP updates by extracting AS Path and NLRI attributes from archived BGP update files, encapsulating them into BGP updates, and sending them to the *validator* according to their timestamps. The *validator* synchronizes ROAs from the RP, performs ROV on the received BGP updates, and forwards the validated updates to the receiver, which logs the timestamp of each validated update upon arrival. Moreover, we measure BGP convergence time based on our team's self-developed emulation platform [20]. The configuration information of the experimental environment is shown in Table 2.

6.2 Algorithm Performance Evaluation

6.2.1 Configurations of h²ROV

The threshold for δ determines the set of wide ROAs, influencing both validation speed and memory usage in h²ROV. We set the optimal threshold for both original and refined versions of h²ROV via experiments using datasets ROA_2024 and BGP_RRC00, considering both speed and memory costs.

As shown in Figures 7 and 8, the optimal threshold for δ in IPv4 and IPv6 scenarios for the original version of h²ROV is 8 and 5, respectively, and 8 and 7 for the refined version. These thresholds are used in our evaluations.

6.2.2 Performance optimization

Figures 9 and 10 show validation speed and memory cost when performing ROV on BGP updates with the original and refined versions of h²ROV, using datasets ROA_2024 and BGP_RRC00. The refined version achieves an increase in validation speed of 20.1% and 48.6%, respectively, for IPv4 and IPv6 compared to the original version. Correspondingly, memory usage increases by 11.4% for IPv4 and 72% for IPv6 but remains within acceptable limits. Furthermore, we evaluate the validation speed of both versions of h²ROV using BGP_20C. The experimental results in Appendix D.3 demonstrate that

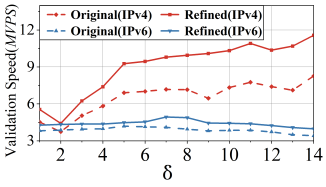


Figure 7: The validation speed with different thresholds for δ.

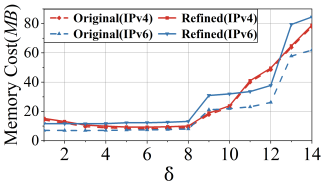


Figure 8: The memory cost with different thresholds for δ.

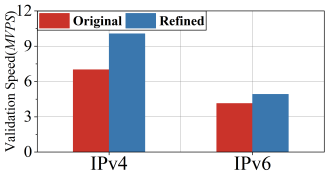


Figure 9: The validation speed of two versions of h²ROV.

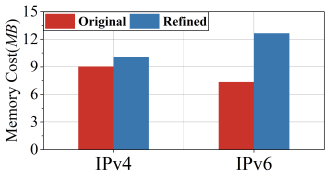


Figure 10: The memory cost of two versions of h²ROV.

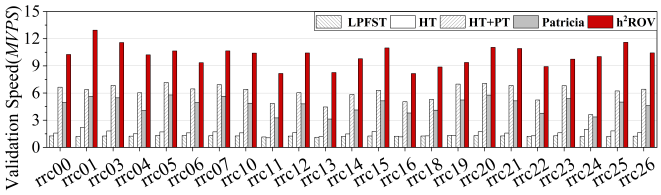


Figure 11: The validation speed of five ROV schemes in IPv4.

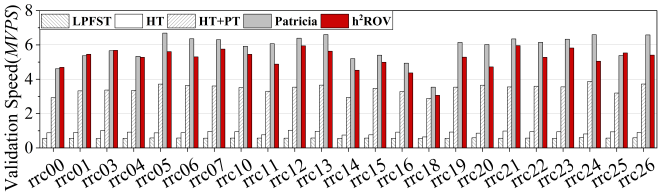


Figure 12: The validation speed of five ROV schemes in IPv6.

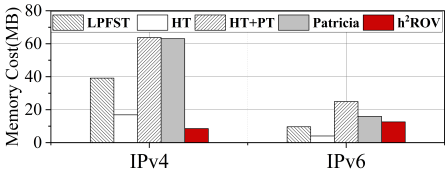


Figure 13: The memory cost.

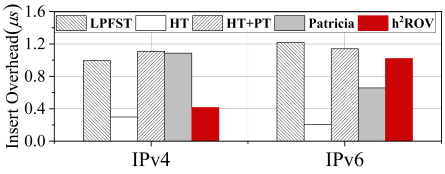


Figure 14: The average insertion time.

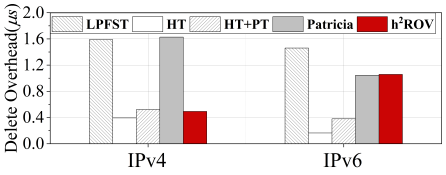


Figure 15: The average deletion time.

the refined version exhibits a clear advantage in validation speed. Therefore, we adopt the refined version for further experiments and system implementation.

6.2.3 Comparison with existing ROV schemes

Validation speed. We evaluate the validation speed of five schemes using datasets `ROA_2024` and `BGP_20C`, as shown in Figures 11 and 12. In the IPv4 scenario, `h²ROV` achieves validation speed of 8.1 to 12.9 MVPS across different BGP collectors. Its validation speed is 6.8× to 9.8×, 4.3× to 6.2×, 1.7× to 2.4×, and 2.2× to 2.9× that of `LPFST`, `HT`, `HT+PT`, and `Patricia`, respectively. These results demonstrate a significant performance advantage of `h²ROV` in the IPv4 scenario. In the IPv6 scenario, `h²ROV` achieves validation speed of 3 to 5.96 MVPS. While it outperforms three ROV schemes—`LPFST`, `HT`, and `HT+PT`—it is slightly slower than `Patricia` across most collectors.

Memory consumption. Figure 13 shows the memory consumption of five ROV schemes. In the IPv4 scenario, `h²ROV` consumes 8.5 MB memory, reducing memory consumption by 78.1%, 49.3%, 86.6%, and 86.4% compared to `LPFST`, `HT`, `HT+PT`, and `Patricia`, respectively. In the IPv6 scenario, `h²ROV` consumes 12.6 MB memory. Although it uses less memory than `HT+PT` and `Patricia`, it requires more memory than `LPFST` and `HT`. Overall, `h²ROV` achieves combined memory consumption of 21.1 MB for IPv4 and IPv6, exceeding `HT` by only 0.2 MB while remaining significantly lower than other evaluated schemes.

Update performance. We evaluate the update performance of five ROV schemes. For insertion evaluation, we use the dataset `ROA_2024`. For deletion evaluation, we randomly select part of the dataset `ROA_2024` for removal. Figures 14 and 15 show that the update performance of `h²ROV` is better than other ROV schemes except for `HT` in the IPv4 scenario. However, in the IPv6 scenario, `h²ROV` outperforms only `LPFST`. Overall, `h²ROV` completes an ROV rule insertion or deletion in under 1μs. Appendix C indicates that the frequency of ROA updates per second is significantly low. Consequently, the ROV update performance does not pose a bottleneck for route validation.

6.3 Scalability of h²ROV

6.3.1 Volume of ROAs

With the ROAs and BGP updates collected from the operational networks every year (datasets `ROA_13Y` and `BGP_13Y`), we measure the validation speed and memory consumption of the five schemes.

As shown in Figure 16, `h²ROV` demonstrates a more stable and upward trend in validation speed for IPv4 BGP updates, whereas the validation speeds of other ROV schemes decline over time, except for `HT`. From the trend demonstrated in Figure 17, we can deduce that the superiority of `h²ROV` over other ROV schemes will become clearer with the increase in the number of ROAs in IPv6 scenario. In terms of memory cost, `h²ROV` clearly outperforms other schemes in IPv4, as shown in Figure 18, and for IPv6 (Figure 19), its memory usage increases but at a slower rate than `Patricia` and `HT+PT`.

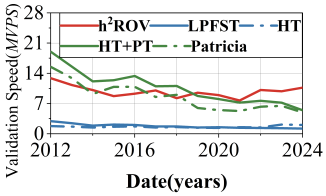


Figure 16: The validation speed at varying times in IPv4.

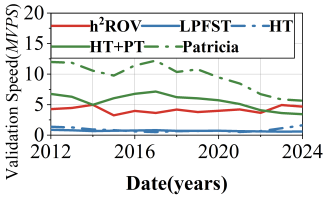


Figure 17: The validation speed at varying times in IPv6.

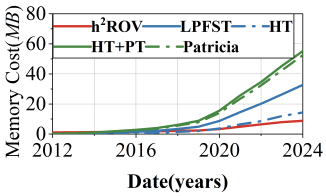


Figure 18: The memory cost at varying times in IPv4.

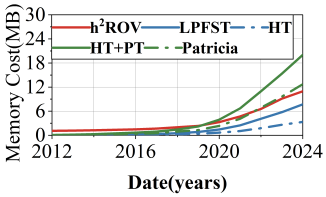


Figure 19: The memory cost at varying times in IPv6.

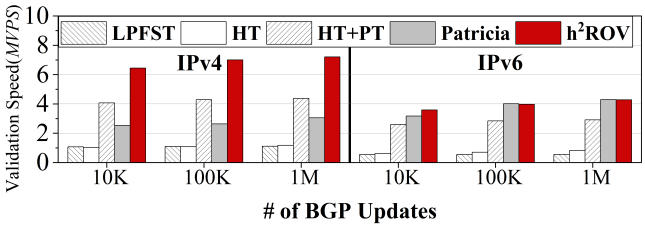


Figure 20: The validation speed with different volume of BGP updates.

6.3.2 Volume of BGP updates

To investigate the impact of BGP update volume on ROV validation speed, we use the datasets `ROA_2024` and `BGP_SIZE`. As depicted in Figure 20, our results demonstrate a consistent pattern across all ROV schemes: validation speed gradually increases with the growing number of BGP updates. This is because extracting more BGP updates increases the cache hit rate due to redundancy in BGP updates, leading to a gradual rise in validation speed.

6.3.3 Distribution of validity states

As RPKI adoption grows, the increasing issuance of ROAs will alter the distribution of route validity states. To measure the impact of these changes on validation speed, we evaluate ROV schemes using the datasets `ROA_2024` and `BGP_ROV`.

The experimental results are shown in Figure 21 and Figure 22. `LPFST`, `HT+PT`, and `Patricia` show a decreasing trend in validation speed compared to the current state (nearly 50% *valid*), as the ratio of *valid* results increases in both IPv4 and IPv6. These ROV schemes achieve the fastest validation when the ratio of *notFound* results is highest. In contrast, `HT` shows an increasing trend in validation speed as the ratio of *valid* results rises. However, it remains slower than `h²ROV` due to the *AB Model*, which prevents `HT` from finding a matching ROA directly. Overall, `h²ROV` outperforms all other ROV schemes when the ratio of *notFound* results is below 40%, a scenario reflecting the future RPKI deployment levels.

6.3.4 Incremental IPv6 deployment

To analyze the impact of the incremental IPv6 deployment, we use datasets `ROA_2024` and `BGP_IP46` to measure the validation performance of five ROV schemes under different proportions of IPv4 and IPv6 BGP updates.

As shown in Figure 23, as the proportion of IPv6 BGP updates increases, a noticeable decline in validation speed is observed for `h²ROV`, `LPFST`, `HT`, and `HT+PT`, while `Patricia` shows an increase in validation speed. The validation speed of `h²ROV` decreases from 6.53 MVPS to 3.99 MVPS as the IPv6 proportion rises from 0% to 100%. However, it still demonstrates significantly better validation performance than `LPFST`, `HT`, and `HT+PT`, except at 100% IPv6 updates, where its speed is just 0.028 MVPS lower than `Patricia`.

6.4 System Performance Evaluation

In the system evaluation, we first measure the average delay introduced by ROV in the *BGP decision process* during normal BGP update reception. We implement five ROV schemes in `FRRouting` and `BIRD` and evaluate them using datasets `ROA_2024` and `BGP_RRC00`. Based on experimental results, `h²ROV` introduces the least average delay to the *BGP decision process* at 19.8%(5.9%) , achieving reduction of 10.4%(13.3%) $\sim$ 56.1%(61.4%) compared to other ROV schemes in `FRRouting` (`BIRD`).

Next, we replicate a real update burst event to evaluate the impact of ROV on average BGP update processing delay during sudden high-volume updates. Using 340K BGP updates from `BGP_RRC00` replayed in a small-scale testbed, Figures 24 and 25 reveal a significant increase in average processing delay during the burst, exacerbated by ROV enforcement. Overall, `h²ROV` exhibits the least impact compared to other ROV schemes. When the average processing delay reached its peak (282nd second), `h²ROV` introduces the smallest average processing delay, reducing it by 48.8%(46.9%) $\sim$ 83.2%(70.9%) compared to other schemes with `FRRouting` (`BIRD`).

Finally, we evaluate the impact of ROV on BGP convergence time in the fully ROV-deployed scenario across real-world and generated topologies, based on our team's self-developed emulation platform [20]. The real-world topologies are constructed using data from the CAIDA's AS relationships [21]. The first real topology includes the top 100 ranked ASes, where ROV deployment is assumed to be more prevalent, following suggestions from studies [3, 22] to prioritize high-ranking or Tier-1 ASes. The second real topology is the South Korean network, chosen for its regional completeness, supported scale in the emulation platform, and the real-world KT network outage incident [23]. We source the top 100 ranked ASes from AS Rank [24] and Korea ASes

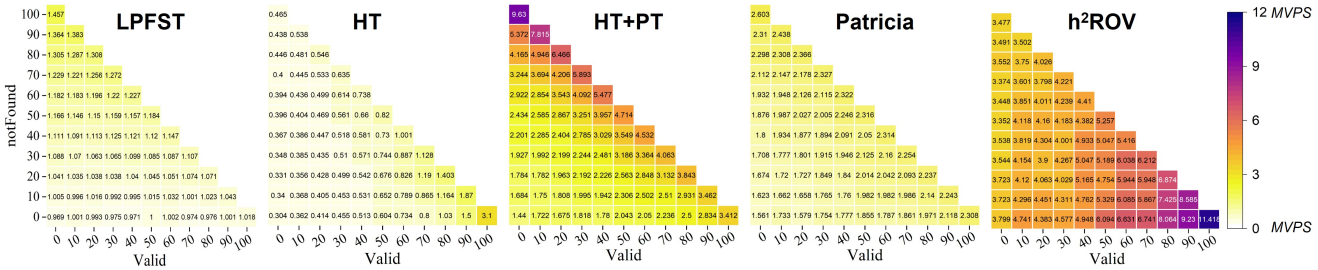


Figure 21: The validation speed of five schemes with different distributions of validity states of routes in IPv4.

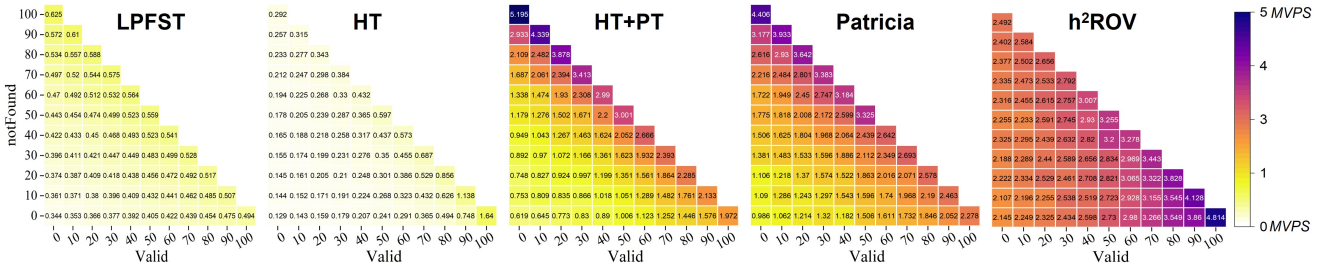


Figure 22: The validation speed of five schemes with different distributions of validity states of routes in IPv6.

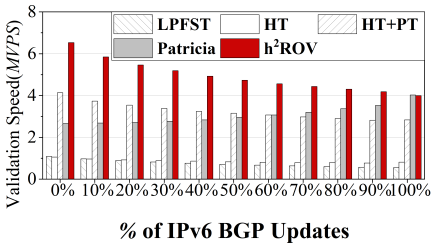


Figure 23: The validation speed with different IPv4/IPv6 updates ratio.

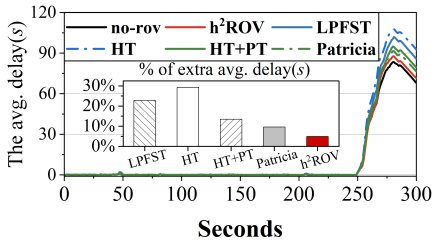


Figure 24: The average processing delay with `FRRouting`.

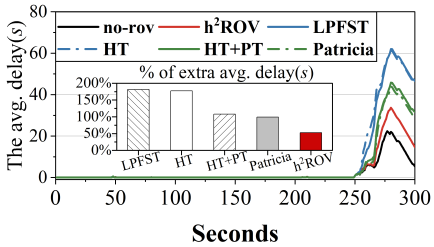


Figure 25: The average processing delay with `BIRD`.

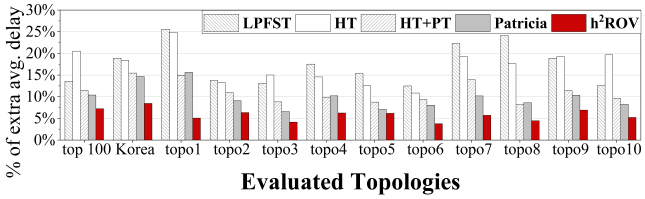


Figure 26: The average proportion of ROV delay in the BGP convergence time.

from Hurricane Electric [25]. Then, we derive sub-graphs containing top 100 ranked ASes and Korea ASes from the complete topology built using CAIDA AS Relationship data on June, 2024 as top 100 topology and Korea topology. The top 100 topology has 100 nodes and 1,916 edges, and the Korea topology has 803 nodes and 1,413 edges. Furthermore, ten AS topologies with statistical parameters aligned with known AS topologies are constructed using the Hyperbolic Graph Generator [26]. Each topology has 1K ASes, an average node degree of 6.4 which is the value observed in the CAIDA AS level topology [21] on June 2024, and a power-law degree distribution with an exponent of 2.1, as in [27, 28]. The AS transit delay is set to 1 ms for all topologies. Once the BGP nodes in the topology are initiated and neighbor relationships are established, BGP updates from the dataset `BGP_RRC00` are injected into the node with the highest connectivity. The BGP convergence time is measured from the start of route reception until all routing tables stabilize without further changes [29]. We evaluate the impact of ROV implementation on convergence time under both normal route updates and update burst scenarios. In the normal scenario, BGP updates were transmitted according to the timestamps in the dataset, whereas in the burst scenario, all updates were transmitted simultaneously. Under normal operating conditions, the convergence time increased only by 1.1% $\sim$ 9.5%. However, during the update burst scenario, ROV-induced routing convergence delays were substantial, reaching up to 8.0% $\sim$ 25.6%. As shown in Figure 26, `h²ROV` increases the convergence time by only 3.8% $\sim$ 8.5%. Specifically, in real-world network topologies, it mitigates ROV-induced routing convergence delays by up to 30.4% $\sim$ 64.7%, while in generated topologies, the mitigation reaches up to 12.5% $\sim$ 81.6%. In summary, `h²ROV` exhibits the minimal impact on convergence time.

7 Related Work

Measurement of ROV enforcement. The quantification of ROV adoption is important to assess its effect in protecting the

inter-domain routing system. Generally, measurement of ROV enforcement can be conducted with uncontrolled BGP data analysis [22] or enhanced passive monitoring [30]. On the other hand, active probing techniques [3, 31] have been used to improve the accuracy. While hybrid approaches [32] and multi-plane analysis [33] achieve better trade-offs between accuracy and efficiency.

Boosting the security benefit of ROV. Partial RPKI deployment limits the security benefit of ROV. It has been demonstrated that Tier-1 ISP adoption [22, 34] and transit provider prioritization [3, 30] maximize effect of ROV in stopping route hijackings. While ROV++ [35] introduces a series of optimizations that can prevent over 90% of subprefix hijackings with a low rate of ROV adoption. Besides, provider-origin validation [4] has been proposed towards asymmetric trust relationships, and RPKIN [36] enables collaborative hijack containment via neighbor notifications.

Advances in ROV implementation. IETF has standardized RPKI ROV [9] for years, its reference implementation using linear search has been outdated. RTRLib [6] employs a Longest Prefix First Search Tree (LPFST) to simplify the validation process at the cost of poor scalability. BIRD [7] introduces a hybrid approach combining prefix trie and hash tables together to improve the comprehensive performance. While BGP-SRx [8] employs a Patricia tree to accelerate the validation, outperforming other solutions.

8 Discussion

8.1 Performance Limitation of h²ROV in IPv6

The basic h²ROV scheme incurs linearly scaling hash probes on the STT relative to route prefix length. In IPv4, our level-bitmap optimization bounds this to a constant three probes, yielding O(1) time complexity. In IPv6, however, the 128-bit address space precludes direct application of level bitmaps. While our binary search approach reduces the complexity to O(log n), it achieves less pronounced speedups than IPv4's constant-time validation. Consequently, h²ROV's performance in IPv6 shows marginal improvements over existing schemes; it is even slightly slower than Patricia in some cases. Our future work will explore heuristic strategies [37] to bridge this performance gap, potentially aligning IPv6 efficiency with IPv4's deterministic guarantees.

8.2 Hardware Implementation Guidelines

Modern hardware routers implement control plane functions—including BGP decision processes and RPKI-based validation—via software stacks [38, 39] running on general-purpose CPUs. Critically, these workflows exhibit structural parity with software router implementations. Consequently, empirical insights we have learned from integrating h²ROV into software routers directly inform its deployment in hardware environments. Two implementation requirements persist across both platforms: (1) augmenting the PDU handler to construct ROV rules with the bitmap encoding, and (2) substituting the legacy route origin validator with h²ROV's validator module that can boost the performance.

8.3 Incremental Deployment of the AP Model

The *AP model* advances the conventional *AB* framework through a redefined validation process with structured encoding format of validation rules. Its deployment occurs in two backward-compatible phases. In the first phase with legacy compatibility, routers operate without RTR protocol modifications, but require h²ROV's validator as well as its encoder. While the encoder introduces marginal overhead, the cost is dwarfed by h²ROV's aggregate performance gains. Crucially, adoption of the *AP model* remains optional per router, allowing autonomous cost-benefit analysis by operators. In the second phase, when RTR PDUs with bitmap-encoded ROV rules become available[5], h²ROV's encoder can be bypassed. Critically, both phases can operate concurrently within a single administrative domain, enabling incremental *AP model* adoption without stranding legacy infrastructure.

9 Conclusion

While ROV remains a critical defense against BGP hijacking, existing AB-model systems struggle to scale with the increasing volume of ROV rules and velocity of BGP updates. Our proposed AP model and its instantiation, h²ROV, address this challenge through prefix-granular validation, reducing IPv4 lookup complexity to O(1) via level-bitmap optimization. However, IPv6's sparse address hierarchy limits these gains, as h²ROV's binary search adaptation achieves only O(log n) complexity. The integration of h²ROV into BIRD and FRRouting demonstrates its compatibility with classic router control planes, providing a practical pathway for hardware router adoption. The source code of the reference implementation of h²ROV is available at https://github.com/FIRLab-CNIC/h-2ROV.

Acknowledgment

We sincerely thank our shepherd Hongqiang Liu and the anonymous reviewers for their insightful and constructive suggestions. This work was supported by the National Key R&D Program of China under Grant No.2022YFB3104800 and the National Natural Science Foundation of China under the Grant No. 62072430. The corresponding authors of this paper are Yanbiao Li and Gaogang Xie.

[5] How this kind of PDUs are produced depends on the deployment of hanging ROAs [12] and is out of the scope of this work.

References

[1] Yakov Rekhter, Tony Li, and Susan Hares. Rfc 4271: A border gateway protocol 4 (bgp-4), 2006.

[2] Matt Lepinski and S Kent. Rfc 6480: an infrastructure to support secure internet routing, 2012.

[3] Weitong Li, Zhexiao Lin, Md Ishtiaq Ashiq, Emile Aben, Romain Fontugne, Amreesh Phokeer, and Taejoong Chung. Rovista: Measuring and analyzing the route origin validation (rov) in rpki. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, pages 73–88, 2023.

[4] Lancheng Qin, Li Chen, Dan Li, Honglin Ye, and Yutian Wang. Understanding route origin validation (rov) deployment in the real world and why manrs action 1 is not followed. NDSS, 2024.

[5] Bgpmon. https://www.bgpmon.net, 2024.

[6] Matthias Wählisch, Fabian Holler, Thomas C. Schmidt, and Jochen H. Schiller. RTRlib: An Open-Source library in c for RPKI-based prefix origin validation. In *6th Workshop on Cyber Security Experimentation and Test (CSET 13)*, Washington, D.C., August 2013. USENIX Association.

[7] The bird internet routing daemon. https://bird.network.cz/, 2024.

[8] Oliver Borchert, Kyehwan Lee, Kotikalapudi Sriram, Douglas Montgomery, Patrick Gleichmann, and Mehmet Adalier. Bgp secure routing extension (bgp-srx): Reference implementation and test tools for emerging bgp security standards. 2021.

[9] P Mohapatra, J Scudder, D Ward, R Bush, and R Austein. Rfc 6811: Bgp prefix origin validation, 2013.

[10] J Snijders, B Maddison, M Lepinski, D Kong, and S Kent. Rfc 9582: A profile for route origin authorizations (roas), 2024.

[11] R. Bush and R. Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 1. RFC 8210, September 2017.

[12] Yanbiao Li, Hui Zou, Yuxuan Chen, Yinbo Xu, Zhuoran Ma, Di Ma, Ying Hu, and Gaogang Xie. The hanging roa: A secure and scalable encoding scheme for route origin authorization. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 21–30. IEEE, 2022.

[13] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354, 1989.

[14] Bgp in 2023 – have we reached peak ipv4? https://labs.apnic.net/index.php/2024/01/05/bgp-in-2023-have-we-reached-peak-ipv4/, 2024.

[15] Donghong Jiang, Yanbiao Li, Yuxuan Chen, Jing Hu, Yi Huang, and Gaogang Xie. Heuristic binary search: Adaptive and fast ipv6 route lookup with incremental updates. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, pages 47–53, 2023.

[16] Frrouting project. https://frrouting.org/, 2024.

[17] Bgp in 2023 – bgp updates. https://labs.apnic.net/index.php/2024/01/08/bgp-in-2023-bgp-updates/, 2024.

[18] Index of /rpki/. https://ftp.ripe.net/rpki/, 2024.

[19] Routing information service (ris). https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/, 2024.

[20] Sernes. https://www.sernes.cn/, 2024.

[21] The caida as relationships dataset. https://www.caida.org/catalog/datasets/as-relationships, 2024.

[22] Yossi Gilad, Avichai Cohen, Amir Herzberg, Michael Schapira, and Haya Shulman. Are we there yet? on rpki's deployment and security. *Cryptology ePrint Archive*, 2016.

[23] South korean telco goes down, blames ddos attack, later admits its bgp broke. https://www.theregister.com/2021/10/26/kt_outage_bgp_not_ddos/, 2021.

[24] As rank. https://asrank.caida.org/, 2024.

[25] Hurricane electric. https://bgp.he.net/, 2024.

[26] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguná. Hyperbolic geometry of complex networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 82(3):036106, 2010.

[27] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. Swift: Predictive fast reroute. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 460–473, 2017.

[28] Thomas Alfroy, Thomas Holterbach, Thomas Krenc, KC Claffy, and Cristel Pelsser. The next generation of bgp data collection platforms. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 794–812, 2024.

[29] Almekdad Abaid, Mahmoud Hraib, Abdulrazig Ben Ghazzi, and Salem Sati. Convergence time analysis of border gateway protocol using gns3. In *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, pages 689–694. IEEE, 2021.

[30] Cecilia Testart, Philipp Richter, Alistair King, Alberto Dainotti, and David Clark. To filter or not to filter: Measuring the benefits of registering in the rpki today. In *Passive and Active Measurement: 21st International Conference, PAM 2020, Eugene, Oregon, USA, March 30–31, 2020, Proceedings 21*, pages 71–87. Springer, 2020.

[31] Andreas Reuter, Randy Bush, Italo Cunha, Ethan Katz-Bassett, Thomas C Schmidt, and Matthias Wählisch. Towards a rigorous methodology for measuring adoption of rpki route validation and filtering. *ACM SIGCOMM Computer Communication Review*, 48(1):19–27, 2018.

[32] Nils Rodday, Ítalo Cunha, Randy Bush, Ethan Katz-Bassett, Gabi D Rodosek, Thomas C Schmidt, and Matthias Wählisch. Revisiting rpki route origin validation on the data plane. In *Proc. of Network Traffic Measurement and Analysis Conference (TMA), IFIP*, 2021.

[33] Tomas Hlavacek, Amir Herzberg, Haya Shulman, and Michael Waidner. Practical experience: Methodologies for measuring route origin validation. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 634–641. IEEE, 2018.

[34] Tomas Hlavacek, Haya Shulman, Niklas Vogel, and Michael Waidner. Keep your friends close, but your routeservers closer: Insights into {RPKI} validation in the internet. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4841–4858, 2023.

[35] Reynaldo Morillo, Justin Furuness, Cameron Morris, James Breslin, Amir Herzberg, and Bing Wang. Rov++: Improved deployable defense against bgp hijacking. In *NDSS*, 2021.

[36] Man Zeng, Xiaohong Huang, Pei Zhang, Dandan Li, and Kun Xie. Improving prefix hijacking defense of rpki from an evolutionary game perspective. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[37] Donghong Jiang, Yanbiao Li, Yuxuan Chen, Jing Hu, Yi Huang, and Gaogang Xie. Heuristic binary search: Adaptive and fast ipv6 route lookup with incremental prefix updates. *IEEE/ACM Transactions on Networking*, 2024.

[38] Examine border gateway protocol frequently asked questions. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/5816-bgpfaq-5816.html, 2024.

[39] H3c routers document shelf-6w100. https://www.h3c.com/en/Support/Resource_Center/EN/Home/Routers/00-Public/Doc_Sets/Doc_Shelf/H3C_Routers_Document_Shelf-13071/?CHID=900688, 2024.

[40] index of rpkiviews. https://www.rpkiviews.org/, 2024.

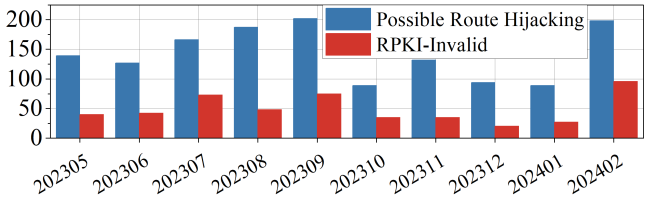


Figure 27: The number of hijacked route prefixes and those protected by ROAs.

A ROA-Protected yet Hijacked Route Prefixes

This appendix provides supplementary results for the measurement described in Section 1.

We measured the number of hijacked prefixes and the number of prefixes for which ROAs were issued. BGPMon [5], a well-known BGP anomaly detection platform[6], reports daily hijacking events. We use reported hijacking events collected from May 1, 2023-February 29, 2024, to measure the number of hijacked prefixes. Then, we identify whether these hijacked route prefixes are protected by RPKI using a set of ROAs archived by RIPE NCC on the same day. As shown in Figure 27, 21% $\sim$ 48% of route prefixes are protected by ROAs but are still hijacked, primarily due to the low ROV adoption rate.

B Theorem Proof

This appendix provides proof of Theorem 1 introduced in Section 3.3.

Definition 8 *Given a prefix p, the first n bits of p are denoted by* $BIT(p,n)$.

Definition 9 $p_i \subseteq p_j$ *if and only if* $BIT(p_i,|p_j|) = BIT(p_j,|p_j|)$ *and* $|p_j| \leq |p_i|$.

Proof. If $rp \nsubseteq R.pfx$, there are two cases according to Definition 9:

[6] It has ceased to offer free service to the public since February 2024.

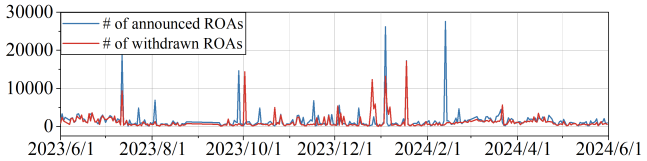


Figure 28: The number of daily announced and withdrawn ROAs.

1. $|R.pfx| > |rp|$, or
2. $BIT(R.pfx, l) \neq BIT(rp, l)$, where $l = |R.pfx|$

For the first case, according to the Definitions 6 and 9, $\forall p \in AP_{set}(R), |p| \geq |R.pfx|$, thus we can get $\forall p \in AP_{set}(R), |p| > |rp|$. Therefore, $\forall p \in AP_{set}(R), rp \nsubseteq p$.

For the second case, according to the Definitions 6 and 9, $\forall p \in AP_{set}(R), BIT(p,l) = BIT(R.pfx,l)$, where $l = |R.pfx|$, thus we can get $\forall p \in AP_{set}(R), BIT(p,l) \neq BIT(rp,l)$. Therefore, $\forall p \in AP_{set}(R), rp \nsubseteq p$.

To sum up, in both cases, rp cannot be a sub-prefix of any authorized prefix in $AP_{set}(R)$. Therefore, we conclude

$$rp \nsubseteq R.pfx \implies \forall p \in AP_{set}(R), rp \nsubseteq p. \tag{4}$$

If $\forall p \in AP_{set}(R), rp \nsubseteq p$, according to Definition 7,we have $R.pfx \in AP_{set}(R)$, then $rp \nsubseteq AB.pfx$. Therefore, we can obtain

$$\forall p \in AP_{set}(R), rp \nsubseteq p \implies rp \nsubseteq R.pfx \tag{5}$$

By combining Eqs. 4 and 5, we prove Theorem 1.

C The Number of Daily Updated ROAs

This appendix provides additional results regarding the number of ROA updates, supplementing the analysis presented in Section 6.2.3 of the main text.

In this section, we measured the number of daily updated ROAs, including both announced ROAs and withdrawn ROAs. We collected the updated ROAs at 30-minute intervals from June 1, 2023 to May 31, 2024 provided by RPKIViews [40]. Then we calculated the number of updated ROAs generated in each interval and sum them up to obtain the total number of updates per day. The result is shown in Figure 28. Obviously, the number of daily updated ROAs is relatively small.

D More Experimental Results

D.1 Memory Usage Compared to RIB

This appendix provides additional results regarding the memory usage of five ROV schemes compared to RIB, supplementing the analysis presented in Section 1.

Table 3: The memory consumption of ROAs of five schemes compared to that of RIB in the current scenarios.

the size of ROAs (MB)			the size of RIB (MB)
	current	future	current
LPFST	48.79	109.40	111.92 (FRRouting) 102.56 (BIRD)
HT	20.97	43.31	
HT+PT	88.70	199.25	
Patricia	78.98	179.00	
h²ROV	21.19	23.25	

To evaluate the additional memory overhead introduced by ROV across five schemes, we conducted a comparative analysis in two scenarios: the current real-world environment and a projected future scenario reflecting increased RPKI adoption. For the current scenario, we used the dataset ROA_2024 to calculate the size of ROV rules. In the future scenario, we assume that all route prefixes will be authorized by ROAs. Therefore, we used BGP updates in dataset BGP_RRC00 as a reference for the issued ROAs. In addition, we use BGP RIB from RRC00 on August 25, 2024 to evaluate the size of RIB. In both scenarios, we evaluated the memory overhead of five schemes used for ROV rule maintenance on two widely-used software routers: FRRouting and BIRD. As shown in Table 3, the memory requirements of existing ROV schemes are significantly lower than those of the RIB in the current environment. However, as RPKI adoption progresses, the memory consumption of these schemes (except for h²ROV) may become comparable to or even significantly exceed that of the RIB. It is noteworthy that h²ROV maintains consistently minimal memory overhead in both current and future scenarios, representing a significant improvement over existing solutions.

D.2 Stability of Results

This section explains why the average of 100 runs was used as the experimental result at the beginning of Section 6.

To evaluate the performance stability of all ROV schemes, both implementations were subjected to 100 independent experimental trials with datasets BGP_RRC00 and ROA_2024, and visualized the results using box plots. As depicted in Figure 29 and Figure 30, the box lengths of all box plots are relatively short, and the median is close to the mean, indicating stable validation speed across trials. Therefore, all experimental results presented in this paper are based on the average of 100 runs.

D.3 Original VS. Refined

This appendix provides additional results for the experiments described in Section 6.2.2.

We compare the validation speeds of the original and refined versions of h²ROV using the BGP_20C dataset. The results, shown in Figures 31 and 32, clearly indicate that the

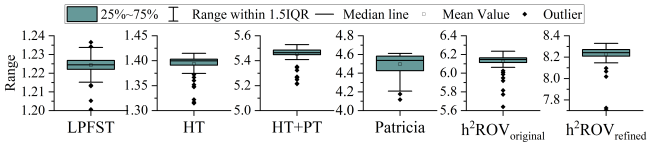


Figure 29: The validation speed (MVPS) distribution across 100 runs in IPv4.

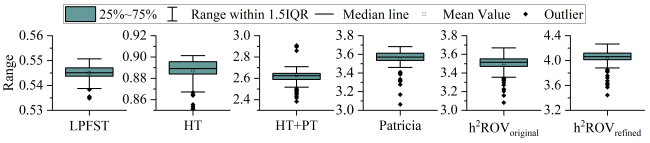


Figure 30: The validation speed (MVPS) distribution across 100 runs in IPv6.

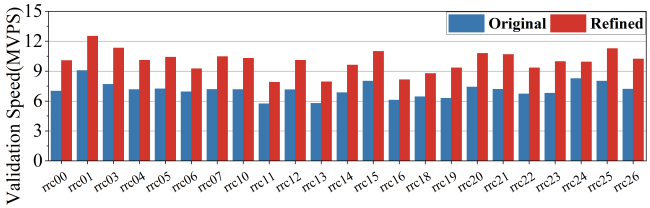


Figure 31: The validation speed of original and refined versions of h²ROV across different sets of BGP updates in IPv4.

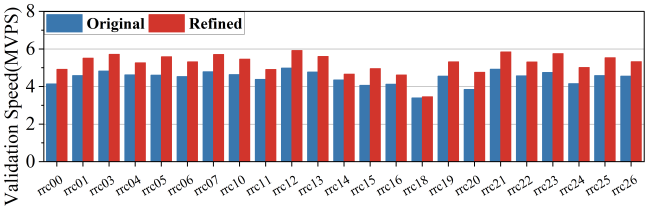


Figure 32: The validation speed of original and refined versions of h²ROV with different sets of BGP updates in IPv6.

refined version consistently outperforms the original version across different collectors.

E More Details about Datasets

This appendix provides further details regarding datasets utilized in the experimental evaluation presented in Section 6.1.

The ROA datasets comprise `ROA_13Y` and `ROA_2024`, both sourced from RIPE NCC. The dataset `ROA_13Y` contains ROAs collected annually on January 1 from 2012 to 2024, while the dataset `ROA_2024` contains ROAs from August 25, 2024. The BGP datasets consist of `BGP_RRC00`, `BGP_13Y`, `BGP_20C`, `BGP_ROV`, `BGP_IP46`, and `BGP_SIZE`. The dataset `BGP_RRC00` contains BGP updates collected by `RRC00` on August 25, 2024. The dataset `BGP_13Y` contains BGP updates provided by `RRC00` on January 1 of each year between 2012 and 2024. The dataset `BGP_20C` contains BGP updates collected from 20 RIPE RIS collectors on August 25, 2024. The dataset `BGP_ROV` is randomly generated based on `ROA_2024`, forming 66 subsets with 10% incremental ROV result proportions, each containing 100K updates. These randomly generated BGP updates are constructed as follows: the origin AS in each BGP update is randomly generated, and the IP prefix is generated by first creating a random 32-bit or 128-bit IP address, then selecting a mask length at random, and finally setting the bits beyond the mask length to 0. The dataset `BGP_IP46` is derived by sampling 400K updates from `BGP_20C` and adjusting the proportion of IPv4 and IPv6 updates in 10% increments, producing 11 subsets. Similarly, The dataset `BGP_SIZE` constructed from `BGP_20C` but varying the number of BGP updates to three predefined sizes: 10K, 100K, and 1,000K. Among these, `BGP_RRC00`, `BGP_13Y`, and `BGP_20C` are based on real-world BGP traces, whereas `BGP_ROV`, `BGP_IP46`, and `BGP_SIZE` are either randomly generated or selected according to predefined rules. Details about `BGP_RRC00`, `BGP_13Y`, and `BGP_20C` are presented in Table 4.

Table 4: The details about datasets.

Dataset	Date	# of IPv4 entries	# of IPv6 entries	Source
ROA_13Y	2012/1/1	1282	292	ARIN, RIPENCC, APNIC, LACNIC, AFRINIC
	2013/1/1	3729	753	
	2014/1/1	7538	1301	
	2015/1/1	13852	2331	
	2016/1/1	25714	3599	
	2017/1/1	46143	5638	
	2018/1/1	77960	14557	
	2019/1/1	59924	10127	
	2020/1/1	108419	18370	
	2021/1/1	184381	30267	
	2022/1/1	256707	50590	
	2023/1/1	340052	73849	
	2024/1/1	405157	92160	
ROA_2024	2024/8/25	487751	116911	
BGP_RRC00	2024/8/25	473412	507589	RRC00
BGP_13Y	2012/1/1	92486	547	RRC00
	2013/1/1	26813	3545	
	2014/1/1	55163	1283	
	2015/1/1	29280	100109	
	2016/1/1	56114	8484	
	2017/1/1	129369	17985	
	2018/1/1	24043	9662	
	2019/1/1	419878	27081	
	2020/1/1	1563401	560336	
	2021/1/1	732461	622754	
	2022/1/1	395531	321171	
	2023/1/1	246565	469999	
	2024/1/1	542245	868546	
BGP_20C	2024/8/25	473412	507589	RRC00
		548969	243079	RRC01
		438326	264281	RRC03
		47883	26104	RRC04
		111812	78794	RRC05
		45943	29425	RRC06
		146265	80305	RRC07
		113893	63433	RRC10
		222002	62804	RRC11
		468027	280173	RRC12
		119613	50995	RRC13
		94229	58061	RRC14
		200757	112557	RRC15
		51621	31763	RRC16
		3619	3198	RRC18
		131208	19635	RRC19
		292710	261981	RRC20
		212891	107878	RRC21
		15089	23071	RRC22
		155382	84208	RRC23
		340887	114207	RRC24
		806405	638830	RRC25
		51992	27539	RRC26

PreAcher: Secure and Practical Password Pre-Authentication by Content Delivery Networks

Shihan Lin[1], Suting Chen[2], Yunming Xiao[3]*, Yanqi Gu[4], Aleksandar Kuzmanovic[2], Xiaowei Yang[1]
[1] *Duke University,* [2] *Northwestern University,* [3] *University of Michigan,* [4] *University of California, Irvine*

Abstract

In today's Internet, websites widely rely on password authentication for user logins. However, the intensive computation required for password authentication exposes web servers to Application-layer DoS (ADoS) attacks that exploit the login interfaces. Existing solutions fail to simultaneously prevent such ADoS attacks, preserve password secrecy, and maintain good usability. In this paper, we present PreAcher, a system architecture that incorporates third-party Content Delivery Networks (CDNs) into the password authentication process and offloads the authentication workload to CDNs without divulging the passwords to them. At the core of PreAcher is a novel three-party authentication protocol that combines Oblivious Pseudorandom Function (OPRF) and Locality-Sensitive Hashing (LSH). This protocol allows CDNs to pre-authenticate users and thus filter out ADoS traffic without compromising password security. Our evaluations demonstrate that PreAcher significantly enhances the resilience of web servers against both ADoS attacks and preserves password security while introducing acceptable overheads. Notably, PreAcher can be deployed immediately by websites alone today, without modifications to client software or CDN infrastructure. We release the source code[1] of PreAcher to facilitate its deployment and future research.

1 Introduction

Websites have been using web authentication to identify users and protect user accounts for decades. Despite alternatives exist [57, 74, 87], websites still heavily rely on password authentication because of its high usability [28, 33]. According to the estimate by prior studies [28, 101] in 2023, more than one-third of websites contain password login interfaces. However, researchers have expressed the concerns about password authentication's security defects, and a long-standing one is Application-layer Denial of Service (ADoS) attacks [2, 41].

ADoS refers to the attacks where attackers aim to deplete a server's resources by sending merely a small number of requests. Differing from Distributed Denial of Service (DDoS) attacks, the traffic volume in ADoS is much less than that in DDoS, so it will not trigger the commonly used DDoS defense [7, 15]. Password authentication is vulnerable to ADoS attacks because it demands intensive computation. Presently, a server stores a password' hash value instead of the cleartext in its database to prevent password cracking in case the database is compromised [14, 31, 36]. Thus, the server computes the hash value of a password and check it against the record in its database. However, a server must use a slow hash function, such as scrypt [81], PBKDF2 [77], or Argon2 [32], for password hashing, in order to slow down brute force guessing when an attacker captures the database. Therefore, this practice makes password authentication compute-intensive. In § 2.1, we present a proof of concept for such ADoS attacks on the Internet, where merely 150 login requests per second can deplete all four modern CPU cores [10] on a server.

In today's Internet, ADoS can occur during credential stuffing [18, 19] incidents. In such cases, attackers use automated bots to repeatedly attempt login with various combinations of usernames and passwords, aiming to impersonate legitimate users. Akamai reports that it detects an average of 280 million suspicious bot logins per day [18]. Such a large number of illegal logins not only poses security concerns on the safety of user accounts, but also imposes a prohibitively high workload on a web server , due to the slow hashing requirement of password authentication.

Presently, there is no effective defense against the ADoS attacks that exploit password authentication. Existing approaches fail to simultaneously achieve the goal of avoiding server overload, preserving good usability, and providing password secrecy. In practice, websites usually adopt rate limit, CAPTCHA, and two-factor authentication (2FA) in login interfaces [25]. However, attackers can bypass rate limits by switching to different usernames and IP addresses and thus keep sending login traffic. As for CAPTCHA and 2FA, they are criticized for their usability by both users and researchers [39, 47, 48, 55, 64, 99]. Although 2FA prevents user account cracking, it does not reduce the workload of failed

*Yunming Xiao is the corresponding author.
[1] https://github.com/SHiftLin/NSDI2025-PreAcher

logins because the server needs to verify the password as the first factor. Currently, some Content Delivery Network (CDN) providers such as Akamai and Cloudflare have deployed bot detection mechanisms to help websites filter out the login traffic generated from bots [18, 25]. However, these bot detection mechanisms use machine learning and behavior analysis to detect bot requests and cannot deterministically avoid false positive or false negative results. Besides, websites have to allow a CDN to inspect their traffic from users, leading to password exposure to the CDN [35, 101].

In addition, a website may delegate password authentication to a third party by using the service of an authentication provider like Auth0 [3] or using the Single Sign-On (SSO) scheme such as OpenID [21] and OAuth [53]. Such delegated authentication releases the burden of user verification on the server. However, it is not suitable for security-sensitive websites such as banks that cannot trust a third party for user authentication. Additionally, delegated authentication introduces a single point of failure where a vulnerability in a single delegate affects millions of user accounts [89], as exemplified by the recent Google OAuth incident [26].

Motivated by the concerns above, we propose a novel web authentication architecture named *PreAcher* to securely offload the password authentication to a third-party CDN without divulging user passwords to the CDN. Our insight is that traditional password authentication were designed between two parties, *i.e.* a client and a server, before CDNs became an essential component in the web ecosystem. Consequently, they were not able to prevent ADoS attacks and password leakage simultaneously. Differently, we design the PreAcher architecture to incorporate a CDN into the password authentication process. At a high level, PreAcher deploys password-agnostic "*pre-authentication*" on a CDN to intercept failed login requests, relieving the server from unnecessary authentication workload. Specifically, when a user sends a login request to a website, the CDN first pre-filters the password without knowing the exact user password. Only when the password is highly likely to be correct, then will the CDN forward the password to the origin server for final authentication. This design enables a CDN to filter out most of the login requests with incorrect passwords, including those malicious ones intended for ADoS/DDoS attacks and account cracking.

To meet our goal of preventing ADoS attacks while protecting password secrecy, we face two main design challenges. The first challenge arises from enabling a CDN to efficiently pre-filter a password without exposing the actual password to it. In this paper, we assume a CDN as a honest-but-curious adversary where it will operate the functionalities and protocols faithfully but may have unintentional bugs exploited by attackers to peek at or guess user passwords through the messages in transmission [49, 75, 76]. To defend against such an adversary, we combine Oblivious Pseudorandom Function (OPRF) [37] and Locality-Sensitive Hashing (LSH) [58] to design a novel three-party authentication protocol involving a client, a CDN, and a server in PreAcher. This protocol not only hides the password in the login requests but also prevents a CDN from guessing the passwords, thereby efficiently safeguarding and verifying user passwords.

The second challenge concerns compatibility issues. The proposed pre-authentication protocol involves new operations of the CDN, the client, and the web server, which inevitably raises the concerns on compatibility. We elaborately design PreAcher's architecture to be compatible with the current web ecosystem. Specifically, we implement PreAcher's CDN operations on the existing serverless computing services [61], such as Akamai EdgeWorkers [1], AWS Lambda [9], and Cloudflare Worker [8]. These services allow web developers to run customized request processing functions on a CDN's edge servers. Besides, we implement all client operations as a JavaScripts library, which are imported into the web pages and seamlessly installed in the browser when a user visits the website. Therefore, a website can immediately deploy PreAcher unilaterally to safeguard its server.

To sum up, this paper makes the following contributions:

1. We design and implement the first secure and practical three-party password authentication architecture that incorporates a client, a CDN, and a server, *PreAcher*. PreAcher not only protects websites from ADoS/DDoS attacks exploiting the login interfaces but also prevents password exposure to third-party CDNs.
2. PreAcher can be immediately deployed in the current web ecosystem by a website unilaterally without any modification of web clients, CDNs, or hardware.
3. We conduct comprehensive analysis and evaluations on PreAcher, and the results show that it enhances the password security and servers' resistance to ADoS attacks. Besides, the security of the proposed authentication protocol in PreAcher is formally proved (Appendix § D). We also show that it introduces acceptable overhead to the throughput and latency of the login process.

Ethical concerns: This work does not raise any ethical concerns. We use our own server as the victim in all experiments, and the network traffic volume we send to the Internet is small ($< 10Mbps$).

2 Background, Threat Model, and Goals

In this section, we first review the ADoS attacks exploiting password login interfaces. We also present a proof-of-concept experiment of initiating such ADoS attacks towards a web server behind a commercial CDN's protection service. Then we introduce the service model of the CDN used in this paper before we present our threat model and assumptions. Finally, we propose the design goals of our solution.

2.1 ADoS Attacks on the Password Login

Password logins are one of the most popular features on websites. To securely verify user passwords, web servers must compute the hash values of passwords [28], as they store only

the salted hash values [14], rather than the plaintext, due to the risk of data breach [24]. Furthermore, the hash function used in password verification must be slow hashing, such as scrypt [81], PBKDF2 [77] and Argon2 [32], so that an attacker who captures a password database still needs a long time to brute-forcedly crack the passwords [14, 31, 36]. Therefore, password hashing is deliberately designed to require intensive computation. As the modern processors become more powerful, password hashing also evolves to remain workload intensive on those processors. For example, PBKDF2's iteration number is recommended to increase from 1000 in 2000 [77] to 10000 in 2017 [51]. Additionally, new algorithms like Argon2 [32] are computationally intensive on both modern CPUs and GPUs. However, such intensive computation also affects the server's throughput in verifying user login requests, leading to not only possible server overload during concurrent user logins but also the vulnerability of ADoS attacks [2, 22, 23].

Different from DDoS attacks where attackers generate volumetric network traffic to the victim server, ADoS usually needs much less traffic to crash the server because of the asymmetric computation between a client and the server, such as the login request processing as mentioned above. An attacker can send a few login requests to consume up all the server's computing resources and block valid users from login.

Proof of concept. We present an experiment to show the feasibility of ADoS attacks on a server behind a commercial CDN through the login interface. We set up a Virtual Machine (VM) of with 4 vCPUs (Intel Xeon E-2288G [10]) and 16 GiB memory on Azure as the web server. We also employ one of the largest commercial CDNs and pay for its DDoS protection, bot detection, and Web Application Firewall (WAF). The CDN applies a rate limit of around 10 req/sec per IP address on the requests. We use a desktop computer in our university as the client. The server hosts a typical password login page, and every login request is enforced to traverse through the CDN to reach the server. Our goal is to use the client as an attacker to deplete the server's CPUs by login requests.

We implement two password hashing algorithms on the server: PBKDF2 [77] and Argon2 [32]. The former is a traditional algorithm standardized in 2000, while the latter is the state-of-the-art one proposed in 2016. We configure the iteration number of PBKDF2 as 10000, which is the default value of OpenSSL [13] and is recommended by NIST [51]. As for Argon2, we use the recommended parameters by OWASP [14]: 19 MiB of memory, 2 iterations, and 1 degree of parallelism. In both cases of server implementation, the client uses Puppeteer [16] to control Chrome to send 150 login requests per second as the attack traffic to the server. This attack lasts for one hour and the requests contain randomly generated login credentials. The client also adopts the distributed proxies of Bright Data [5] to send requests with different source IP addresses.

In both cases of PBKDF2 and Argon2, we observe that the CDN does not intercept any request, and the server's CPU utilization stays around 100% during the one-hour attack. Besides, the server cannot respond to any other user requests. Therefore, we successfully bypass the commercial CDN's protection and launch ADoS attacks on the victim server.

2.2 CDN Service Model

PreAcher leverages third-party CDNs, such as Akamai and Cloudflare, for pre-authentication due to their extensive networks of distributed edge servers around the world. These edge servers provide sufficient computing and network resources to defend against DoS attacks. However, researchers have raised security concerns about CDNs: to enable a third-party CDN to serve the HTTPS connections from clients, websites have to share their TLS private keys to the CDN [35, 68]. This practice allows a third-party CDN to access the private data transferred in HTTPS connections between users and websites [54, 69], such as user passwords [101]. In addition, researchers also showed that attackers can exploit a CDN's configuration error to fetch the transferred private data [49, 75, 76]. Therefore, in PreAcher's design, we aim for not only defending against ADoS attacks but also preventing the password exposure to CDNs.

In this paper, we refer to the web server hosted by a website and hidden behind a CDN as the "*origin server*" or simply "*server*". We use the term "*user*" to denote a website's user rather than a CDN's user. A user accesses a website through a web "*client*" such as a browser. We describe an attacker as being "*inside*" a CDN when it exploits a vulnerability within the CDN to gain internal privileges, or when it is a rogue insider of the CDN provider. For simplicity, we sometimes use "*CDN*" to refer to an attacker inside a CDN. Additionally, an attacker "*outside*" of a CDN refers to any malicious entity except for the CDN itself.

2.3 Threat Model and Trust Assumptions

Attackers inside CDNs. Prior studies [54, 69, 101] describe two threat models of a CDN as below:

1. **Passive attacker**: A CDN will honestly provide the functionalities it promises, but its components such as the log and the storage may contain unintentional bugs like configuration errors. An attacker can exploit these bugs to access to the data, but cannot modify a CDN's behavior. The attacker can eavesdrop on the messages transferred by a CDN but *cannot* tamper with, duplicate, or fabricate any message. For example, the attacker can observe the passwords or attempt to infer the passwords from the transferred messages. This model is also called a "*honest-but-curious*" model.
2. **Active attacker**: A CDN's component such as the control plane contains severe vulnerabilities that an attacker can exploit to not only eavesdrop on transferred messages but also tamper with, duplicate, and fabricate the messages. An active attacker is more powerful than a passive attacker.

In PreAcher, we assume a *passive attacker* inside a CDN due to the following reasons. 1) Researchers have shown that passive attackers could exist in commercial CDNs, such as web cache deception attacks [49, 75, 76], where attackers deceive CDNs to accidentally cache some users' private data and grant public access to the data. 2) Prior studies have proposed methods to ensure the integrity of CDN-hosted content based on a secure DNS channel [46, 69], which prevents a CDN from tampering with the message without being detected. PreAcher can adopt these methods to further defend against an active attacker. We leave the defense against active attackers for future work.

We note that a CDN will not launch DDoS/ADoS attacks on the hosted websites in our study, as the CDN is assumed to be a passive attacker.

Attackers outside CDNs. An attacker outside a CDN can actively manipulate the messages it interacts with. For example, it may gain access to a client's Wi-Fi router and alter, duplicate, or forge the transferred messages. We also consider it has the capability to launch ADoS or DDoS attacks. This may involve sending send a large number of login requests with various combinations of usernames and guessed passwords to the login interface of a website.

Origin servers. Since the origin server is the source of all web content, we trust the server for all users' sensitive data. We do not consider data breaches or web providers' misoperations on the server in this paper.

ADoS-resist registration. We assume the server's registration interface is resistant to ADoS. This assumption is achievable in reality. Firstly, since the registration is a one-time cost, a website can adopt complex CAPTCHA on the registration page without affecting the user experience of future visits. Besides, for those web services that are highly security conscious, the web provider can require a user to register an account offline. For example, a user may need to visit a local branch to open a bank account.

Trusted Computing Base. We trust the implementation of existing cryptographic libraries in the programming languages. We also trust the implementation of the hardware, browsers, and operating systems used by the client and server.

2.4 Design Goals

In this paper, we aim to design and implement a password pre-authentication system on CDNs with the following goals:

1. **Security**: Our system focuses on securing password authentication. It should prevent the ADoS attacks exploiting the login interface, while avoiding password exposure to CDNs or other attackers. Besides, the system should retain the CDN's security benefits such as DDoS protection and WAF. However, we do not aim to prevent ADoS attacks using other channels beyond password authentication, nor do we protect other sensitive data or private communications handled by the current CDNs beyond passwords.
2. **Compatibility**: Our system should be compatible with the current web ecosystem. Its deployment should not involve multiple stakeholders on the Internet. Our design achieves this by enabling websites to deploy the system unilaterally. The system does not require any modification on current browsers, CDN infrastructure, or operating systems. It only requires updating the website's login page and processing.
3. **Efficiency**: Our system should not introduce much overhead to the login procedure from the perspectives of throughput and latency.

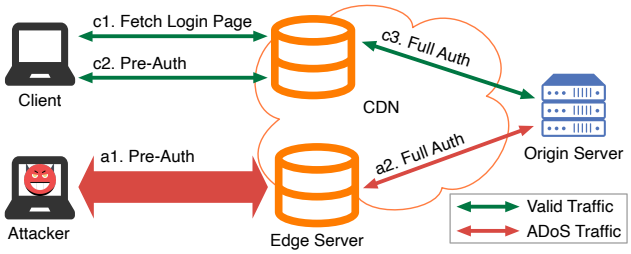


Figure 1: **PreAcher architecture. The width of arrows shows the traffic volume.**

Achieving these three goals simultaneously is non-trivial. We utilize both techniques of authentication protocols and the features of current web development practice and thus finally design and implement PreAcher shown in this paper.

3 Design Rationales

In this section, we first illustrate an overview of PreAcher. Then we discuss the design rationales of PreAcher.

3.1 Overview

The core of PreAcher is an authentication protocol involving three parties: the client, CDN, and the server. It includes a *pre-authentication* stage on the CDN and a *full authentication* stage on the server, which we will elaborate in § 4.

Figure 1 shows an overview of PreAcher. All components of PreAcher run on the application layer of the Internet for compatibility. When a user visits a website and intends to login, the web client (browser) fetches the HTML files of the website's login page from the CDN (`c1`). Since we assume a passive attacker in the CDN (§ 2.3), the CDN cannot modify the login page and thus cannot conduct phishing on user passwords. After the user types in her username and password on the web page, the client communicates with CDN to pre-authenticate the user (`c2`), which may include multiple round trips. Only if the client passes the pre-authentication, the CDN will forward the client's request to the origin server for full authentication (`c3`). If the server also successfully verifies the user's identity, it will return an indication of success to the CDN and the CDN finally notifies the user of a successful login. Otherwise, the client is rejected to login.

When an attacker attempts to compromise some accounts, it can instrument programs to directly send usernames and passwords to the CDN and start the pre-authentication (`a1`), without loading the HTML files into a browser. In most cases,

the pre-authentication will fail, and the CDN immediately rejects the requests without forwarding them to the server. Therefore, if the attacker sends login requests to initiate ADoS on the server, the CDN will filter out most of them to protect the server. Since the CDN conducts a pre-authentication which does not ensure 100% correctness of the credentials, some attack requests may still be forwarded to the server for full authentication (a2). However, the server will reject this portion of requests. Besides, we elaborately design the authentication protocol to ensure that the amount of these requests is small enough and will not lead to ADoS on the server.

3.2 HTTPS Compatibility

PreAcher is compatible with the current web ecosystem, and thus all communications among the client, the CDN, and the origin server use HTTPS connections, including the requests/responses of page loading, pre-authentication, full authentication, *etc.*. We keep current TLS private key-sharing practice between CDNs and websites, so the CDN can still access to the HTTPS payloads. In contrast, an attacker outside of a CDN cannot break HTTPS and view the content, since it does not have the TLS private key.

3.3 Private Communication After Login

PreAcher does not aim to protect general private communication between the client and the server. Instead, PreAcher focuses on preventing the password exposure and ADoS/DDoS attacks caused by current password authentication. A website can adopt the existing proposals, InviCloak [69], to build a secure channel between the client and server with the existence of a CDN and thus transfer the private data securely through the channel. Besides, after a success login, websites usually grant the client authentication cookies to avoid logins for future visits. PreAcher does not provide the protection of these cookies either, but websites can also use InviCloak to protect the authentication cookies.

3.4 Password vs Additional Key

Some design ideas may require a client to hold an additional key—such as a private key—after the registration. We decided not to adopt these designs because they conflict with current user behaviors: users are required to remember or save additional materials beyond the passwords. Although a web client can help a user to store the additional key, a user may change her web client. Thus, PreAcher adopts a design compatible with the current user behavior of memorizing the username and password only, which is arguably the most universal authentication method.

3.5 Preventing Offline Dictionary Attack

In an offline dictionary attack [29, 34, 82], an attacker obtains a dictionary of user passwords, enumerates all password candidates in the dictionary, and verifies the guessed password offline (locally) without communications with the server. Such an attack requires the attacker to find an approach to determine the correctness of the guessed password locally. For example, an attacker may obtain the hash value and the salt of the actual password so that it can compare the hash value of the guessed password with the actual hash value to know the correctness. Therefore, the attacker can enumerate all passwords in the dictionary locally until it finds the correct one, and it will not be detected by the server.

The defense against such an attack becomes particularly challenging when we consider a passive attacker inside a CDN in PreAcher's design. Generally, if a CDN can fully authenticate users to filter out all invalid requests, then the CDN can launch offline dictionary attacks. This is because the passive attacker can observe all transmitted messages and all computational states of the CDN during a legitimate user login. With this information, for each password the attacker guesses, it can simulate the authentication protocol between the client and the CDN to determine the password's correctness. Note that such an attack remains within the bounds of passive attacker behaviors since the simulation relies solely on observed data and is conducted offline.

Furthermore, if we prevent the CDN from obtaining any information about password correctness to thwart such offline dictionary attacks, the CDN will be unable to filter out the failed login requests for the server.

We addresses such a challenge based on two observations:

1. Although offline dictionary attacks are hazardous, *online* password-guessing attacks are considered much less risky since they are detectable and can be easily rate-limited by the origin server.
2. Currently, the state-of-art dictionary generation algorithms generate the passwords based on a user's historical passwords [80,102] or a user's personal information [91]. Thus, the password candidates in a dictionary should be similar to each other to some extent [56].

Therefore, in PreAcher, instead of granting a CDN the capability of determining the password correctness, we propose to only enable the CDN to determine the similarity between the provided password and the actual password. The CDN can still help the server filter out massive failed login requests with erroneous passwords that are not similar to the actual password, and it forwards the login requests with probably correct passwords to the server for further authentication.

Such a design prevents offline dictionary attacks initiated by a CDN because a CDN cannot verify similar passwords offline. By PreAcher's design, for those similar passwords in the dictionary, CDN cannot determine which one is the correct one, and it has to query the server to verify every similar password. Therefore, CDN can only guess the password online with the server.

The design still prevents ADoS attacks. For attackers outside the CDN, if it randomly generates the passwords, the portion of passwords that can pass the CDN's pre-authentication will be very small (see § 5). Moreover, as we analyze in

§ 6, it is difficult for attackers outside the CDN to identify the passwords that can reach the origin server and trigger an ADoS attack. Furthermore, even if they gain access to such passwords, since the password authentication is enforced to be online and shares a pattern of similarity, the attack is detectable by the server.

We describe the algorithm to enable a CDN to determine the similarity of passwords in § 5, which is based on Locality-Sensitive Hashing (LSH).

4 Authentication Protocol

As previously discussed, our protocol consists of two stages: the pre-authentication on the CDN and the full authentication on the server. Although the idea of this two-stage protocol is simple, instantiating such a protocol is non-trivial. In this section, we first present the protocol used in PreAcher, and then we discuss an intuitive but less efficient design.

4.1 PreAcher Protocol

The core idea of our protocol is to combine Oblivious Pseudo-random Function (OPRF) [37] and Locality-Sensitive Hashing (LSH) [58] to enable pre-authentication at a CDN while limiting its knowledge of password correctness.

OPRF is a protocol that allows two parties (Alice and Bob) to obliviously evaluate a PRF function $F(K;X)$, where Alice inputs K and learns nothing, and Bob inputs X and learns $F(K;X)$ without knowing K. In this paper, we use the 2HashDH OPRF proposed in prior work [59]. OPRF is also used by existing Password Authenticated Key Exchange (PAKE) protocols like OPAQUE [60]. Our protocol is inspired by OPAQUE but is different from its design in two aspects:

1. OPAQUE is designed for two-party authentication between the client and server. We extend OPAQUE to three parties.
2. By adding the CDN into the protocol, OPRF itself cannot solve the offline dictionary attacks by the CDN. Our protocol incorporates LSH to defend against the attacks.

Furthermore, the security of this proposed new protocol is also formally proved. We describe the workflow of the protocol in this section, and present the proof in Appendix D.

We first describe the notations used in the protocol below before we elaborate on the protocol details.

- The client obtains the username u and the password p from a user's input. The server generates a new key pair $\langle \mathsf{pk}_s, \mathsf{sk}_s \rangle$ during the deployment of the system. Note that this key pair is different from the TLS key pair.
- For clarity, the symbols with a subscript "$_u$" are related to a specific user u, and the symbols with a superscript "$'$" are related to pre-authentication, *e.g.* a user's key pair used for pre-authentication $\langle \mathsf{pk}'_u, \mathsf{sk}'_u \rangle$.
- The protocol builds upon a Diffie-Hellman cyclic group $G = \langle g \rangle$, where g is the generator, and n is the bit length of the group.
- H_1 is a cryptographic hash function, while H_2 is password hashing. *LSH* denotes a locality-sensitive hashing.

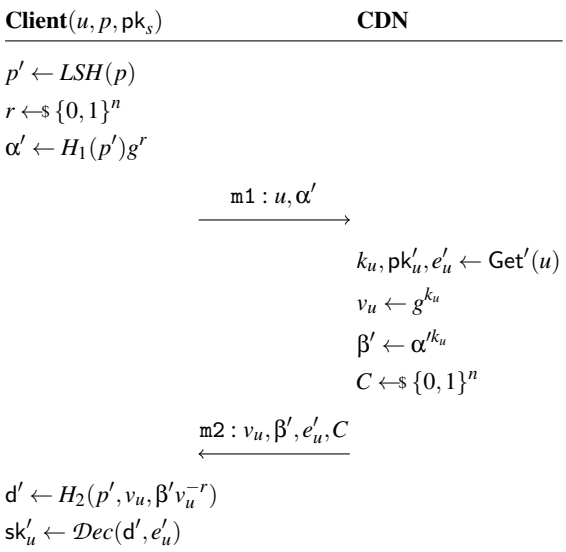


Figure 2: **First round of PreAcher login.**

- The protocol uses a CPA-secure symmetric encryption scheme $(\mathcal{K}g, \mathcal{E}nc, \mathcal{D}ec)$ and a CCA-secure asymmetric encryption scheme $(\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$.
- We use a standard digital signature scheme $(\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vf})$ with existentially unforgeability under chosen-message attack (EU-CMA).
- $\mathsf{Save}(u, \cdot)$ (resp. $\mathsf{Save}'(u, \cdot)$) denotes saving the input into a user-associated record stored by the origin server (resp. CDN), while $\mathsf{Get}(u)$ and $\mathsf{Get}'(u)$ denote the retrieval of the corresponding records associated with a user u.

The protocol is divided into two phases: registration and login. During the registration, we assume trust on first use so the client can communicate directly to the server. The server first generates a random number k_u. Then the client and the server will use OPRF to compute a symmetric key d' through the password and k_u. The client generates a key pair $\langle \mathsf{pk}'_u, \mathsf{sk}'_u \rangle$, and encrypts the sk'_u into an envelope e'_u with the key d'. The server sends $k_u, \mathsf{pk}'_u, e'_u$ to the CDN after the registration, and the CDN will use them to help the server pre-authenticate users and filter out invalid login requests. The client also registers the user with the server by traditional password hashing (without exposing the password to the CDN), which will be used for full authentication on the server during login. Given space constraints, we leave more details of registration in Appendix A. For the rest of this section, we focus on the login protocol.

Initialization. A user inputs u and p to the client. The client obtains pk_s, and the server holds the corresponding sk_s. The server can delivered pk_s to the client with integrity by embedding it in the HTML of the login page as discussed in § 3.1. Besides, the CDN already received corresponding $k_u, \mathsf{pk}'_u, e'_u$ for registered users from the server.

Generating the secret key from the password. Similar to the registration, the client obtains the secret key d' through

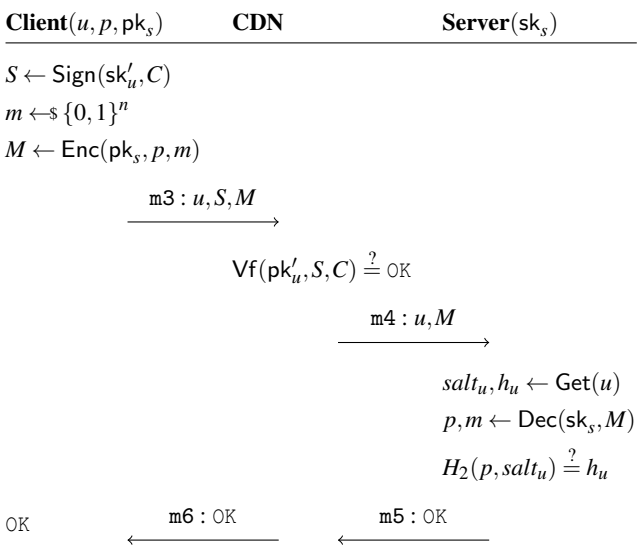


Figure 3: **Second round of PreAcher login.**

the combination of OPRF and LSH, but it communicates with the CDN instead of the server as shown in Figure 2.

Specifically, the client uses LSH to map the password p to a pseudo-password p'. Then it generates a random nonce r and computes α'. After receiving u and α' from the client (m1), the CDN retrieves k_u, pk'_u, and e'_u from the storage according to u. Then it computes v_u and β', and returns them to the client with e'_u and a challenge C (m2). The client computes d' through the same function in the registration.

Retrieving the private key from the envelope. With d', the client can decrypt e'_u to obtain the private key sk'_u which is generated in the registration. This completed the first round of PreAcher login.

Pre-authentication and full authentication. In the second round, the client computes the credentials for pre-authentication and full authentication, respectively. Then it sends them to the CDN and the server together to save the RTTs of the protocol. The procedure is shown in Figure 3.

For pre-authentication, the client signs C with sk'_u and send the signature S to the CDN (m3). Meanwhile, for full authentication, it encrypts p and a nonce m with pk_s and sends the encrypted message M to the CDN without concerns of exposing p (m3). Since the CDN stores pk'_u, it can verify the signature S. The pre-authentication succeeds if and only if the verification succeeds. If the pre-authentication fails, the CDN stops the protocol and sends an error message to the client, which is not shown in the diagram. When the pre-authentication succeeds, the CDN sends u, M to the origin server for full authentication (m4). Note the CDN cannot decrypt M to know p since it does not know sk_s.

When the server receives requests for full authentication, it retrieves $salt_u$ and h_u from records and decrypts M to authenticate the password p by comparing the calculated hash value with h_u. The authentication result is returned to the CDN (m5) and then to the client (m6).

Summary. The protocol uses two different asymmetric key pairs for pre-authentication and full authentication, respectively. For pre-authentication, the client's private key is enclosed in an envelope stored on the CDN with a symmetric encryption key. The encryption key is computed by OPRF from the password in the login. We combine LSH with the OPRF to limit the CDN's capability of guessing passwords. We will discuss how LSH achieves this in § 4.2.

As shown in the protocol, the pre-authentication uses the signature to verify the user, while the full authentication uses the traditional password hashing where the server sees the password and checks against the stored hash value. We expose the password to the server because we trust the server in our threat model (§ 2.3). Besides, we keep such a traditional method in the server to be compatible with the popular authentication implementation in practice. In addition, if we use signature verification and hide passwords from the server, one more RTT will be added to the protocol since a challenge number should be delivered to the client after the pre-authentication. Nevertheless, web operators can easily modify this full authentication approach if they consider the traditional method does not fit their demands.

4.2 Intuitive but Less Efficient Design

In the early stage of our study, we adopted straightforward hashing methods for pre-authentication. Overall, the client hashes p' to obtain $H_1(p')$. The CDN computes password hashing of $H_1(p')$, *i.e.* $H_2(H_1(p'))$, and checks it against with the one in the storage. We call such a double-hash protocol as "*DuoHash*". We illustrate the detail protocol of DuoHash in Appendix C.

However, we found that DuoHash is less efficient in practice, since the protocol runs a slow password hashing on the CDN for pre-authentication. Besides, this slow hashing cannot be replaced by a usual hash function like SHA-256, because such a replacement accelerates the brute-force guessing of p' when an attacker captures the CDN's storage [14,31,36]. For example, using OpenSSL implementation on our testbed, SHA-256 is over 50,000 times faster than PBKDF2. With p', the attacker can simply construct the login requests that pass pre-authentication and thus launch ADoS attacks on the server. Therefore, password hashing is necessary in the double-hash design, leading to a significant performance downgrade of the CDN's edge servers. As a defense of ADoS and DDoS for the origin server, the CDN's operations should be as efficient as possible, so we opt for the OPRF design in this paper. Web developers could adopt DuoHash if they find it suitable for their applicatons, as outlined in our comparison of DuoHash and PreAcher in § 8.

5 Locality-Sensitive Hashing

As presented in § 4.1, we use LSH to map the password p to a pseudo-password p', and use p' in the pre-authentication. The

rationale behind this design is to reduce the CDN's ability to determine password correctness, thereby preventing offline dictionary attacks from the CDN as discussed in 3.5.

The LSH is designed to map a set of similar passwords to the same p' value, and thus, a portion of erroneous passwords can also pass the pre-authentication. Specifically, an attacker inside the CDN may launch offline dictionary attacks by generating a dictionary of password guesses and testing them locally to see which ones pass the pre-authentication. However, by using LSH, the best this attacker can do is to find that a portion (determined by LSH) of passwords are hashed to the correct p', and they are all possibly the actual password. Thus the attacker has to enumerate this portion of passwords and send online requests to the server for full authentication, but the server will reject this login request unless the correct password is guessed. Leveraging LSH, the CDN only knows that a provided password may be similar to the correct one locally. This achieves our purpose of enforcing the CDN to conduct online queries to know the correctness.

For the passwords in a dictionary, we expect to see many of them mapped to the p' of the actual password, so that CDN gains little information through enumerating the passwords in the dictionary. However, to defend against ADoS attacks, we expect to reduce the probability of the passwords in the attack traffic mapped to the actual p' so that the CDN can filter out most failed logins. These two requirements align well with the design of LSH, and we can tune the parameters of LSH to balance the CDN's ability of inference and authentication.

In PreAcher, we use weighted K-mer MinHash [72] as the algorithm of LSH. A string s of length N is first converted to lowercase and then split into $N-K+1$ substrings called "K-mers" by a sliding window of length K. These K-mers form a sequence S, following the order of their positions in s. Formally, let $s[i:j]$ denotes the substring of s from index i to index j. We have

$$S = (s[1:K], s[2:K+1] \ldots s[N-K+1:N]) \quad (1)$$

Then we use weighted MinHash [96] to generate the hash value from S as follows. Since a K-mer (denoted as m) may appear multiple times in S, we use $m^{(w)}$ to denote w-th appearance of m in S. We use w as the weight in weighted MinHash. Then we adopt HMAC [63] with SHA-256 to hash the K-mers in S. Thus, we have a set T consisting of the hashed K-mers.

$$T = \{\mathsf{HMAC}(u, w \parallel m) \mid m^{(w)} \in S\} \quad (2)$$

In Formula 2, we use $\parallel$ to denote concatenation. Finally, we take the minimum element in T as the hash value of LSH.

We emphasize that our LSH design is preimage-resistant. Specifically, an attacker only knows that there exists a cluster of similar p to be mapped onto the same p', but it is difficult to determine the exact p that corresponds to a specific p'. This property originates from our two designs:

1. We use HMAC with SHA-256 to construct the LSH as shown above. As SHA-256 is preimage-resistant, the LSH is also preimage-resistant.
2. We include the username u in HMAC's input. Thus, the mappings from the p to p' vary across different users, further enhancing the resistance to preimage attacks.

We then analyze the collision probability of this LSH. Let c be the alphabet size of a password. Since the K-mer has a length of K and the hash value is generated from K-mers, the size of the hash value space is c^K. Therefore, in the case of ADoS/DDoS initiated by an attacker outside of a CDN, it generates passwords randomly. Then the probability of the collision between a generated password and the actual password is $P_{col} = \frac{1}{c^K}$. This is the fraction of the attack requests that pass the pre-authentication by the CDN. For example, let $K = 4$ and $c = 66$, where c includes 26 case-insensitive letters, 10 digits, and 30 special characters allowed by Linux passwords, then P_{col} is less than 10^{-7}. It means that PreAcher can filter out almost all DoS traffic.

To identify the correct password, an attack inside the CDN could offline determine the password candidates that are mapped to the actual p' and then send these candidates to the server for online verification. In this case, the server will see multiple failed login requests of an account from the CDN. For an account, the server should consider it is under attacks if more than Q times of failed logins happen on the server. The server could temporarily freeze the user account and ask the CDN to inspect the possible insider attackers. However, these frequent failed logins may also come from a valid user's erroneous input. Thus, the server can tune Q to adjust the sensitivity to the attacks. We run an empirical experiment simulating this scenario to show the defense effectiveness of LSH as below.

We collect 5000 distinct users and their passwords from 4iQ dataset [38]. For each user, we collect two passwords, one is regarded as a historical password leaked to a CDN, the other one is regarded as the actual password of the user's account on a website served by the CDN. Suppose the website adopts the CDN for pre-authentication, and the CDN aims at cracking those 5000 user accounts in this website through the collected historical passwords. Suppose the CDN adopts a state-of-the-art dictionary generation algorithm, *pass2path* [80] for password guessing. For each account, it uses this algorithm to generate a dictionary of 10,000 passwords from the historical password. The CDN can locally filter the password candidates that pass pre-authentication and send them to the server for full authentication. When the pre-authentication does not adopt LSH or the website opts for full authentication on the CDN, as long as the dictionary contains the correct password, the CDN will find it locally and crack the account successfully. However, with LSH, only if less than Q password candidates are left after the offline filtering can the CDN ensure itself cracks the account through less than Q online queries without being detected by the server. We compute the success rate of such undetected cracking on 5000 accounts for different values of Q and K in our LSH.

Figure 4 shows the success rate of cracking. Firstly, the

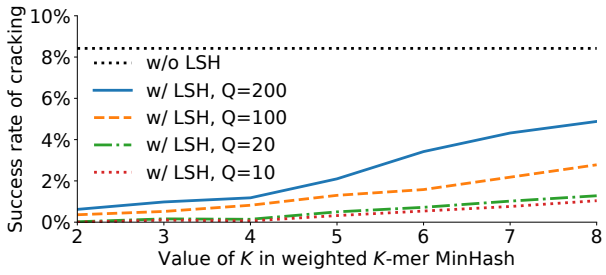


Figure 4: **The CDN's success rate of cracking accounts varying with K and Q in LSH. When LSH is not used (the black line), the success rate does not depend on K or Q.**

black line is about 8.42%, indicating a relatively high chance that the CDN can find out an account's password offline when LSH is not adopted. However, the success rate is dramatically reduced with LSH applied. It is less than 1% When $K \leq 4$ and $Q \leq 100$. The success rate also rises as K and Q increases, because larger K leads to fewer password collisions with p' in LSH, and larger Q allows more trials from the CDN. Nevertheless, a small K also reduces the effectiveness of the ADoS/DDoS defense of the CDN, and a small Q also downgrades the tolerance of the typos from valid users. Therefore, a server adopting PreAcher should select K and Q properly to defend against both DoS attacks and password cracking and reduce the false alarms. According to Figure 4, we set K as 4 and Q as 20 by default in PreAcher, since the cracking rate is less than 0.20% and almost all DoS traffic cannot pass the pre-authentication ($P_{col} < 10^{-7}$). Websites deploying PreAcher can adjust K and Q based on their demands.

We note that the 0.20% cracking rate is not introduced by PreAcher but results from users frequently reusing similar passwords across websites. A website without PreAcher faces the same cracking rate when an attacker outside of the CDN conducts online password guessing. Additionally, without PreAcher, the CDN sees the passwords directly. With OPRF but not LSH in PreAcher, the CDN can guess passwords offline with a success rate of 8.42% (Figure 4). Our LSH design contributes to the reduction from 8.42% to 0.20%.

6 Security Analysis

In this section, we analyze the security properties of PreAcher and explain how our design defends against attacks. We have formal security proof for our protocol, and we provide a proof sketch in Appendix D.

Registration security: As discussed in our threat model 2.3, we consider registration is ADoS-resist. Besides, the server still hides behind the CDN for registration, and the CDN can provide the DDoS defense. Moreover, the password is not exposed to the CDN during the registration according to the protocol shown in Appendix A.

User impersonation: If an attacker tries to impersonate a user without knowing the correct password, it will be rejected by either the CDN or the server. However, there exist cases in which the attacker provides a wrong password but obtains the correct p' because LSH maps the wrong and correct passwords to the same p'. In such cases, the attacker can pass the CDN's pre-authentication because he can compute the correct d' to decrypt e'_u. However, he is guaranteed to fail the authentication on the server since the wrong password cannot pass the password check against the stored hash.

Password invisibility to the CDN: With our protocol, the CDN cannot observe user passwords, as the passwords are neither transferred in plaintext nor decryptable by the CDN during the communication. During the login, the oblivious nature of OPRF allows the client to retrieve the stored private key sk'_u. Then the client and the CDN use $\langle sk'_u, pk'_u \rangle$ for pre-authentication. Moreover, the client encrypts the password with pk_s during full authentication. As a result, both authentication stages ensure the secrecy of the password. We provide two examples of attack attempts in Appendix B to demonstrate how this protocol can defend against a passive attacker inside the CDN.

ADoS/DDoS attacks: PreAcher prevents ADoS/DDoS attacks by filtering out most failed login requests by pre-authentication on the CDN. When an attacker generates brute-force login requests to overload the server, the server is never involved in the pre-authentication while the CDN will intercept the requests accurately.

An attacker may try to find passwords that pass pre-authentication but fail the full authentication to repeatedly involve the server and drain its resources. However, the attacker cannot identify such passwords in PreAcher. Both authentication stages return the same failure message. When a wrong password is guessed, the attacker cannot tell if the failure is due to pre-authentication or full authentication. Therefore, attackers outside the CDN cannot find passwords accepted by pre-authentication to launch ADoS attacks. While the CDN knows the pre-authentication result, as assumed in § 2.3, a CDN would be improbably involved in ADoS/DDoS attacks.

An attacker may also register an account and use the account's credentials to launch ADoS. The origin server can defend against such attacks by monitoring the login behavior of users. If a user repeatedly logs into her account in a very short time, the website can block the account.

Dictionary attacks: The key idea of PreAcher's defense against dictionary attacks is to force attackers to guess passwords online, and the server enforces a rate limit on the failed login attempts to detect such attacks.

For an attacker outside of the CDN, since it does not know k_u, it can only conduct online guessing for each of its password guesses. The only information the attacker gains is (1) the OPRF output which is indistinguishable from random bits to the attacker due to the pseudorandomness of OPRF and (2) envelops e'_u which the attacker cannot learn anything from due to the semantic security.

PreAcher also prevents offline dictionary attacks from a passive CDN attacker. As discussed in § 3.5, we have to

allow a CDN to guess p' offline to enable it to conduct pre-authentication. However, even with the knowledge of p', the CDN cannot guess p easily online due to our LSH design. As shown in § 5, our design of LSH reduces the cracking rate from 8.42% to less than 0.20%, even when a CDN uses a user's historical password and a state-of-the-art password generation algorithm to make 20 guesses for a single account.

Credential stuffing: Credential stuffing is a type of dictionary attacks where the attacker collects the existing dataset (dictionary) to guess the password. Since PreAcher defends against dictionary attacks, it also mitigates credential stuffing. In addition, PreAcher provides the benefit of reducing the origin server's workload in credential stuffing since most incorrect credentials are filtered out by the CDN.

Replay attacks: For the attacker outside of the CDN, it cannot replay any message of a valid user, as all communications in PreAcher are inside HTTPS. Moreover, the CDN cannot replay messages either, since it is assumed as a passive attacker (§ 2.3). Furthermore, the server can also record the recent messages of M (Figure 3) to prevent the CDN from replaying them, which is adopted by prior work [69].

Side-channel attacks: An attacker outside of the CDN may attempt to identify the passwords passing the pre-authentication by a side channel so that it can launch ADoS attacks with these passwords.

For example, an attacker may exploit response times by sending login requests with different passwords, as failed pre-authentication attempts are rejected by the CDN and replied faster than the successful ones. However, PreAcher can mitigate such a timing side channel by delaying failed responses, masking the pre-authentication results. Such a delay should approximate to the latency between the CDN's edge server and the origin server. Moreover, valid users remain unaffected, since this delay is not applied to successful logins.

In additional, an attacker could send many login requests with a specific password and infer the the server's workload through the website's page load time. A significant increase in the page load time indicates this password pass the the pre-authentication. PreAcher can generally prevent this side-channel attack as well as others aiming at differentiating pre-authentication results because

1. The values of p' used for pre-authentication spread in a large space ($> 10^7$, see § 5), requiring many trials to guess a correct p'. Moreover, each trial takes considerable cost as it involves one or more online requests.
2. The LSH mappings are different across users, and the server freezes an account after Q failed full authentication attempts. Thus, an attacker is limited to Q requests per p' inferred through the side channel, and it must repeat the trails of guessing p' across numerous users to gather enough p' to carry out a meaningful ADoS attack.

Finally, researchers have discovered other side channels that may compromise user data, such as through CPU cache [107], keystroke video [104], improper app isolation [105]. These attacks pose threats not only to PreAcher but also to many other systems. Addressing these attacks is beyond the scope of this paper and requires broader efforts extending beyond websites and CDNs.

7 Implementation

We provide a prototype implementation of PreAcher to facilitate the deployment and future research.

We implement the client operations in the protocol as a JavaScript library so that web developers can easily apply PreAcher by importing the library into their login web pages. The implementation does not require any modification or extension of existing browsers. Therefore, the websites can be seamlessly upgraded without cooperation from users.

We implement the server in C++ with Sogou Workflow framework [17], which is an efficient C++ server engine used in the industry to serve billions of requests per day. Web developers can adapt our server implementation into their web servers, or they can simply run our implementation as a separate process and redirect the received login requests to the process by setting up a reverse proxy [62, 83].

As for CDN operations, we implement them as a JavaScript code snippet running on a CDN's serverless computing service. Currently, CDN providers offer serverless computing to allow websites to run customized code at their edge server to process requests, such as Akamai EdgeWorkers [1], AWS Lambda [9], and Cloudflare Workers [8]. Therefore, our implementation runs seamlessly without requiring changes to the CDN infrastructure, despite introducing new operations on CDN edge servers. Web developers can deploy our JavaScript code snippet directly to the CDN to enable pre-authentication, as all the mentioned CDNs support JavaScript.

For cryptography operations, we adopt the existing libraries Web Crypto API [94] and libsodium [11] in JavaScript implementation and OpenSSL [12] in C++ implementation.

Overall, we elaborately design and implement PreAcher so that it is compatible with the current web ecosystem and can be immediately deployed by websites. Furthermore, PreAcher supports incremental deployment across websites since each website can unilaterally manage its deployment progress.

8 Evaluation

In this section, we evaluate PreAcher's efficacy in defending against ADoS attacks and the overhead introduced to throughput and latency on both the testbed and the Internet.

8.1 Experiment Setup

We set up a testbed on Azure using three VMs as a client, a CDN, and an origin server. Each VM has 4 vCPUs and 8 GiB of RAM and runs Ubuntu. As for Internet experiments, we deploy PreAcher on a commercial CDN, *i.e.* Cloudflare [6] and evaluate it using Azure VMs as the client and the server.

Besides PreAcher, we implement three strawman methods for comparison. The first one models the current password au-

thentication practice in a CDN-enabled website. The CDN's edge servers forward login requests to the origin server for authentication. Although this method directly exposes passwords to the CDN, it is simple and widespread [101]. We call this method as "*Baseline*" in this section. Another method is "DuaHash" discussed in § 4.2. Theoretically, DuaHash also prevents password exposure and ADoS attacks. Finally, we implement a method based on Intel SGX [40]. Specifically, we adopt Occlum [85] to run a traditional password hashing to authenticate users in SGX. We call this method as "*SGX-CDN*', which theoretically prevents password exposure and ADoS attacks but requires hardware support.

PBKDF2 is used for password hashing as shown in § 2.1.

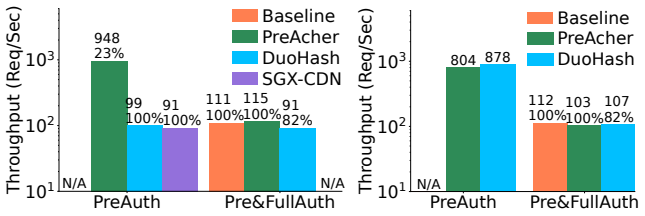
8.2 ADoS Defense

Method	Testbed		Internet	
	w/o ADoS	w/ ADoS	w/o ADoS	w/ ADoS
Baseline	100	0	99	0
PreAcher	100	97	100	100
DuoHash	91	0	98	97
SGX-CDN	90	1	N/A	N/A

Table 1: **The successful logins per second for four methods with and without ADoS attackers. Since Cloudflare does not deploy SGX, we cannot measure the SGX-CDN on Cloudflare.**

We simulate ADoS attacks on the server and compare the efficacy of four proposed methods on the testbed and the Internet. We set up two processes on the client VM to act as valid users and an attacker, respectively. The user process sends login requests at the speed of 100 req/sec to simulate the case where 100 valid users intend to login to the website per second. All requests contain correct usernames and passwords. As for the attacker process, it sends login requests at the speed of 400 req/sec to simulate the case where an attacker initiates ADoS traffic to crash the server. These requests contain erroneous credentials since the attacker does not know the correct one. We run the experiments in two settings on both the testbed and the Internet. In the first setting, only the user process sends requests, representing a normal network condition without attacks (`w/o ADoS`). The second setting contains both the user process and the attacker process, representing a network condition under the ADoS attacks (`w/ ADoS`).

Table 1 shows the throughput (the number of successful logins per second) of each method. On the testbed without attack traffic, the server can respond to the user process timely with any of the methods deployed. However, during ADoS attacks, only PreAcher can maintain high throughput, *i.e.* 97 req/s, while the other methods fail to serve nearly all valid requests. For the baseline, it does not conduct the pre-authentication on the CDN and all attack requests will be forwarded to the server for processing. As for DuoHash and SGX-CDN, the pre-authentication overloads the CDN VM due to the intensive computation for password hashing, and thus the CDN cannot forward the valid requests to the server.



(a) Testbed (b) Internet

Figure 5: **Throughput of pre-authentication (`PreAuth`) and the whole system (`Pre&FullAuth`) for four methods. "N/A" indicates the absence of corresponding operations. Numbers above bars represent the values of throughput and CPU utilization. Some CPU utilizations are missing because Cloudflare does not provide the information.**

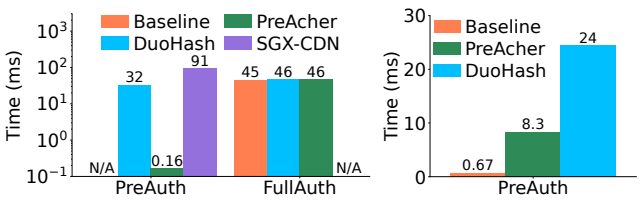
For the experiments on the Internet, the methods also serve user requests normally when there are no attacks. The throughput of DuoHash is improved because Cloudflare has plentiful CPU resources for password hashing. Due to the same reason, DuoHash can serve valid users with 97 req/s even under attacks. PreAcher also prevents ADoS attacks when it is deployed on Cloudflare, and it consumes less CPU than PreAcher as we will presented in § 8.4.

8.3 Throughput

We separate the experiments in this section into two groups: one measures the throughput of failed logins rejected by the CDN, and the other one measures the throughput of successful logins accepted by the server. The former represents the pre-authentication's throughput on the CDN, and we indicate this group as "`PreAuth`". The latter represents the whole system's throughput including both authentications on the CDN and the server, indicated by `Pre&FullAuth`. Specifically, we use `wrk` [20] at the client to send login requests to the CDN by 64 parallel connections (`wrk -c64`).

Figure 5a and Figure 5b show the throughput of the two groups for the four methods on the testbed and the Internet, respectively. Firstly, the baseline does not use the CDN for pre-authentication, and SGX-CDN does not run authentication on the server (it totally relies on the CDN for authentication), so we put "N/A" in the corresponding positions of bars. Figure 5a shows that PreAcher achieves a much higher throughput (948 req/s) than DuoHash (99 req/s) and SGX-CDN (91 req/s) in the `PreAuth` group. Moreover, it takes PreAcher only 23% of CPU to handle these requests, while DuoHash and SGX-CDN use up all CPU resources (100%). These results demonstrate the high efficiency of PreAcher's pre-authentication, which explains the effectiveness of its ADoS protection in § 8.2. The pre-authentication throughput of DuoHash and SGX-CDN is due to the slow password hashing, and SGX-CDN also introduces the overhead of SGX. As for the `Pre&FullAuth` group, all methods' throughput is around 100, because they all use password hashing on the server.

For the Internet experiments shown in Figure 5b,



(a) Testbed (b) Internet

Figure 6: **Median CPU time of pre-authentication (`PreAuth`) and full authentication (`FullAuth`) for four methods. "N/A" indicates the absence of corresponding operations. We omit `FullAuth` results on the Internet since they match those on the testbed due to the identical operations and server machine. The numbers above bars indicate the time values.**

PreAcher's `PreAuth` throughput reduces. We find that this reduction originates from the 2-RTT design of PreAcher and `wrk`. Specifically, the 2-RTT design increases the latency, and in `wrk`, each connection waits for a request to complete before sending the next one. As a result, a higher latency leads to a lower throughput when all 64 connections are saturated. In contrast, DuoHash's `PreAuth` throughput improves because the computation is no longer a bottleneck on Cloudflare.

8.4 CPU Time

We measure the CPU Time of the pre-authentication and full authentication operations. Figure 6 shows the median CPU time measured on the testbed and the Internet.

In Figure 6a, both DuoHash and SGX-CDN take much over 100× more CPU time than PreAcher for pre-authentication on the CDN VM, because they both use password hashing while PreAcher does not. Such a difference in CPU times explains the results of ADoS defense and throughput on the testbed in § 8.2 and § 8.3. As for full authentication on the server VM, PreAcher and DuoHash are comparable with the baseline since they all use the same password hashing algorithm.

For the Internet experiments, full authentication results should remain consistent with Figure 6a as we run identical operations on the same server. Therefore, we focus on pre-authentication's CPU time, which is provided by Cloudflare's statistics. As shown in Figure 6b, PreAcher's CPU time enormously increases to 8.3 ms, compared to 0.16 ms on the testbed. This increment arises from the implementation difference: PreAcher adopts C++ for pre-authentication on the testbed but use JavaScript on Cloudflare, as C++ is not supported in its serverless computing. The performance gap reflects the efficiency difference between these two languages.

However, DuoHash's CPU time does not increase on Cloudflare even though it also uses JavaScript. The reason stems from the cryptographic library. For DuoHash, we implement password hashing with Web Crypto API [94], which actually executes native code with little performance downgrade. Since this API does not support the primitive elliptic curve operations required by PreAcher, we adopt a pure JavaScript library libsodium [11] for it, downgrading the performance.

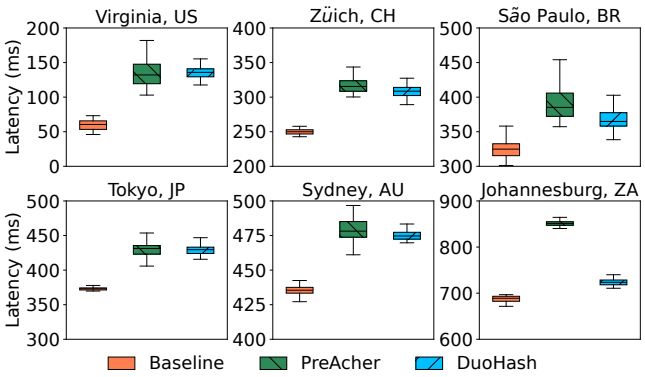


Figure 7: **The latency of a successful login for the clients in six regions around the world.**

Nevertheless, DuoHash still consumes 3× the CPU time of PreAcher on Cloudflare due to password hashing, which will be charged more to websites. Overall, DuoHash is a simple solution, but PreAcher achieves better CPU efficiency. We leave the choice between them for web developers.

8.5 Latency

We measure the end-to-end latency of a successful login for each method. To show a realistic latency, we set up six geographically distributed VMs as clients in six Azure regions covering all continents except Antarctica, including Virginia (US), Zürich (Switzerland), São Paulo (Brazil), Tokyo (Japan), Sydney (Australia), and Johannesburg (South Africa). We set up the origin server in Virginia (US). In each region, we send 100 valid login requests to Cloudflare edge servers. The edge servers will run deployed pre-authentication and request the origin server for full authentication. We define the latency as the time from starting the request to receiving the login success indication.

As shown in Figure 7, in all regions, PreAcher and DuoHash has a higher latency than the baseline. The latency overhead of PreAcher comes from its 2-RTT design in the protocol. However, the additional RTT happens between the client and the CDN, and thus it should not introduce much overhead because the clients are usually close to a CDN's edge server. We spot that PreAcher introduces a high latency overhead in Johannesburg, South Africa. This is because Cloudflare redirects the client's request to an edge server far from Johannesburg, leading to about 140 ms inflation. Except for this region, the overhead of PreAcher's median latency ranges from 42 ms to 72 ms in all other regions, compared to the baseline. Besides, DuoHash introduces overheads ranging from 34 ms to 75 ms to the median latency compared to the baseline in all regions, which is close to PreAcher. The latency overheads of PreAcher and DuoHash primarily stem from their computation time and the access delay of Cloudflare's storage. Since the login process is usually a one-time cost for users during web browsing, we consider the overhead of PreAcher and DuoHash to be acceptable.

9 Discussion

ADoS attacks through other interfaces. Although PreAcher avoids the ADoS attacks exploiting the login interface, other interfaces of a website may still be vulnerable. If the exploited interface requires authentication, the website can track and block accounts that initiate attacks. If the interface is open to public, the website can enforce authentication to remove the attack surface. Websites can also adopt the general ADoS defense proposed in prior studies [44, 67, 73, 90].

Password reuse and weak passwords. A user may reuse exactly the same password across websites. PreAcher cannot prevent account cracking caused by password reuse, which is an inherent limitation of password authentication. To mitigate password reuse, websites can adopt existing password reuse prevention [92] and detection [84] methods, which are compatible with PreAcher.

Similarly, a user may use a common or weak password, and an attacker inside a CDN could collect these passwords and enumerate them offline to guess the user's password. Nevertheless, PreAcher still resists such attacks. Since those common or weak passwords also share similarities, the LSH in PreAcher will mitigate the attacks as discussed in § 5. Additionally, a website can prevent users from using these common or weak passwords during registration.

Alternative solution. An alternative solution to achieve ADoS mitigation and password protection simultaneously is to simply use Trusted Execution Environment (TEE), such as Intel SGX [40]. We evaluate this solution in § 8. However, as TEE requires both hardware and software upgrades and is not enabled by CDNs yet, it is difficult to deploy such a solution. In contrast, PreAcher does not require upgrading of CDNs and can be deployed by websites unilaterally.

10 Related Work

DoS protection. In the past decades, researchers have explored plenty of DDoS defense mechanisms by using the network capabilities [70, 103, 106], the cloud defense [50, 71, 108], the routing polices [86], and the new Internet architecture [97]. In practice, websites widely adopt the DDoS protection offered by a CDN provider or a cloud provider [4, 7, 15]. We do not innovate DDoS protection in PreAcher, but we retain the DDoS protection of CDNs in PreAcher.

Protecting ADoS is more difficult than DDoS because the requests are application-specific and the request amount is usually too small to be detected [2]. Prior studies have proposed ADoS defense mechanisms. However, their solutions focus on the ADoS caused by either regular expressions [66] or specific programming languages [43, 73]. Researchers also propose to detect and filter out ADoS traffic by profiling requests' resource usage and learning the pattern of attack traffic [44, 67, 90], which are less effective to login requests due to the consistent payload pattern of a username and a password. In this paper, we show that ADoS attacks exploiting password login are feasible, even for a server hidden behind a CDN (§ 2.1), and our design of PreAcher mitigates such attacks.

User privacy leakage to CDNs and countermeasures. Researchers have spotted the user privacy leakage to third-party CDNs caused by TLS private key sharing for years [35, 68]. A line of research has proposed countermeasures, including certificate delegation [68], Keyless SSL [88], TLS modifications [30, 65, 79], TEE-based CDNs [27, 54, 78, 95], differential privacy [100], homormophic encryption [98], and a new encryption channel for web [69].

Although these proposals are useful for protecting users' sensitive data against an attacker inside a CDN, none of them mitigates ADoS attacks. PreAcher is related but has different goals. It is not meant to protect all private communication between the client and the server. Instead, it focuses on preventing password leakage and addressing ADoS vulnerability caused by current password authentication.

Password authentication. A series of literature has discussed the security of the password authentication, including the password exposure on CDNs [101], password guessing techniques [80, 102], password storage mechanisms [14, 31], new efficient protocols of password authentication [45, 52, 60], the password reuse practice [42, 93] and its defense [84, 92], etc. PreAcher's authentication protocol is inspired by the advance of Password Authenticated Key Exchange (PAKE) protocol [60], which enables authenticated key exchange between a client and a server without cleartext password checking on the server. However, existing PAKE protocols cannot be directly applied to our system, because the origin server must engage directly with the user in each PAKE session which makes DoS attack unavoidable. Thus, we design a novel three-party authentication protocol in PreAcher to securely offload password authentication to a CDN.

To sum up, PreAcher is the first solution that simultaneously prevents ADoS/DDoS attacks, preserves password secrecy, and provides good usability.

11 Conclusion

This paper introduces PreAcher, a comprehensive solution designed to combat the ADoS/DDoS attacks exploiting password authentication and preserve password secrecy. PreAcher incorporates a three-party authentication protocol that allows CDNs to perform password pre-authentication without accessing the actual passwords. We extensively analyze PreAcher's security features and evaluate its performance. Websites can deploy PreAcher independently and immediately, without requiring any modifications to clients or CDNs.

Acknowledgments

We sincerely thank our shepherd Boris Pismenny and the anonymous reviewers for their constructive comments. We are also grateful to Bobby Bhattacharjee, Zonghao Huang, Matthew Lentz, and Bruce Maggs for their insightful feedback. This work is supported in part by NSF awards CNS-2225448, CNS-2226107 and CNS-2310927.

References

[1] Akamai EdgeWorkers. Retrieved in May, 2024 from https://techdocs.akamai.com/edgeworkers/docs/welcome-to-edgeworkers.

[2] Application Layer DDoS Attack. Retrieved in June, 2024 from https://www.cloudflare.com/learning/ddos/application-layer-ddos-attack/.

[3] Auth0 by Okta. Retreived in June, 2024 from https://auth0.com/.

[4] AWS Shield Features. Retrieved in May, 2024 from https://aws.amazon.com/shield/features/.

[5] BrightData - Scraping Browser API. Retrieved in Aug, 2024 from https://brightdata.com/products/scraping-browser.

[6] Cloudflare. Retrieved in June, 2024 from https://www.cloudflare.com/.

[7] Cloudflare DDoS Protection. Retrieved in May, 2024 from https://developers.cloudflare.com/ddos-protection.

[8] Cloudflare Workers. Retrieved in May, 2024 from https://developers.cloudflare.com/workers/.

[9] Compute Service - AWS Lambda. Retrieved in May, 2024 from https://aws.amazon.com/pm/lambda.

[10] DCsv2 sizes series - Azure Virtual Machines. Retrieved in Jan, 2025 from https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/dcsv2-series?tabs=sizebasic.

[11] libsodium. Retrieved in Aug, 2024 from https://doc.libsodium.org/.

[12] OpenSSL. Retrieved in May, 2024 from https://www.openssl.org/.

[13] OpenSSL Documentation: openssl-enc. Retrieved in Sep, 2024 from https://docs.openssl.org/master/man1/openssl-enc/#options.

[14] OWASP Password Storage Cheat Sheet. Retrieved in June, 2024 from https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[15] Prolexic DDoS Protection: Reference Architecture. Retrieved in May, 2024 from https://www.akamai.com/resources/reference-architecture/prolexic-ddos-protection.

[16] Puppeteer. Retrieved in Dec, 2024 from https://www.brightdata.com.

[17] Sogou c++ workflow. Retrieved in May, 2024 from https://github.com/sogou/workflow.

[18] What Is Credential Stuffing? Retreived in Dec, 2024 from https://www.akamai.com/glossary/what-is-credential-stuffing.

[19] What Is Credential Stuffing? Retreived in Dec, 2024 from https://www.cloudflare.com/learning/bots/what-is-credential-stuffing.

[20] wrk2. Retrieved in May, 2024 from https://github.com/giltene/wrk2.

[21] OpenID Authentication 2.0 - Final. https://openid.net/specs/openid-authentication-2_0.html, 2007.

[22] CVE-2020-28873. https://www.cve.org/CVERecord?id=CVE-2020-28873, 2021.

[23] CVE-2022-29700. https://www.cve.org/CVERecord?id=CVE-2022-29700, 2022.

[24] 2024 Data Breach Investigations Report. Technical report, Verizon, 2024.

[25] The Malicious Bot Playbook: Early Warning Signs, and What to Do about Them. Technical report, Cloudflare, 2024.

[26] Millions of Accounts Vulnerable due to Google's OAuth Flaw. https://trufflesecurity.com/blog/millions-at-risk-due-to-google-s-oauth-flaw, 2025.

[27] Rufaida Ahmed, Zirak Zaheer, Richard Li, and Robert Ricci. Harpocrates: Giving Out Your Secrets and Keeping Them Too. In *Proceedings of IEEE/ACM Symposium on Edge Computing (SEC'18)*, pages 103–114. IEEE, 2018.

[28] Suood Al Roomi and Frank Li. A Large-Scale Measurement of Website Login Policies. In *Proceedings of USENIX Security Symposium 2023*, pages 2061–2078. USENIX, 2023.

[29] José Becerra, Peter YA Ryan, Petra Šala, and Marjan Škrobot. An Offline Dictionary Attack against zkPAKE Protocol. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec'18)*, pages 291–292. ACM, 2018.

[30] Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. A Formal Treatment of Accountable Proxying

over TLS. In *Proceedings of S&P'18*, pages 799–816. IEEE, 2018.

[31] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *Proceedings of EuroS&P'16*, pages 292–302. IEEE, 2016.

[32] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications. RFC 9106, Internet Engineering Task Force (IETF), 2021.

[33] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *Proceedings of S&P'12*, pages 553–567. IEEE, 2012.

[34] Ran Canetti, Shai Halevi, and Michael Steiner. Mitigating Dictionary Attacks on Password-Protected Local Storage. In *Proceedings of CRYPTO'06*, pages 160–179. Springer, 2006.

[35] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. Measurement and Analysis of Private Key Sharing in the HTTPS Ecosystem. In *Proceedings of CCS'16*, pages 628–640. ACM, 2016.

[36] Anthony D Carter and Richard A Johnson. Slow Hashing Speed as a Protection for Weak Passwords. *International Journal of Advanced Engineering and Science*, 2020.

[37] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. SoK: Oblivious Pseudorandom Functions. In *Proceedings of EuroS&P'22*, pages 625–646. IEEE, 2022.

[38] Julio Casal. 1.4 Billion Clear Text Credentials Discovered in a Single Database. https://medium.com/4iqdelvedeep/1-4-billion-clear-text-credentials-discovered-in-a-single-database-3131d0a1ae14, 2017.

[39] Stéphane Ciolino, Simon Parkin, and Paul Dunphy. Of Two Minds About Two-Factor: Understanding Everyday FIDO U2F Usability through Device Comparison and Experience Sampling. In *Proceedings of SOUPS'19*, pages 339–356. USENIX, 2019.

[40] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[41] Scott A Crosby and Dan S Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of USENIX Security Symposium 2003*. USENIX, 2003.

[42] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The Tangled Web of Password Reuse. In *Proceedings of NDSS'14*, pages 23–26. ISOC, 2014.

[43] James C Davis, Eric R Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of USENIX Security Symposium 2018*, pages 343–359. USENIX, 2018.

[44] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with Finelame. In *Proceedings of ATC'19*, pages 693–708. USENIX, 2019.

[45] Bruno Freitas Dos Santos, Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. Asymmetric PAKE with Low Computation and Communication. In *Proceedings of EUROCRYPT'22*, pages 127–156. Springer, 2022.

[46] Huayi Duan, Rubén Fischer, Jie Lou, Si Liu, David Basin, and Adrian Perrig. RHINE: Robust and High-performance Internet Naming with E2E Authenticity. In *Proceedings of NSDI'23*, pages 531–553. USENIX, 2023.

[47] Christos A Fidas, Artemios G Voyiatzis, and Nikolaos M Avouris. On the Necessity of User-Friendly CAPTCHA. In *Proceedings of CHI'11*, pages 2623–2626. ACM, 2011.

[48] Sanam Ghorbani Lyastani, Sven Bugiel, and Michael Backes. A Systematic Study of the Consistency of Two-Factor Authentication User Journeys on Top-Ranked Websites. In *Proceedings of NDSS'23*. ISOC, 2023.

[49] Omer Gil. Web Cache Deception Attack. https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html, 2017.

[50] Yossi Gilad, Amir Herzberg, Michael Sudkovitch, and Michael Goberman. CDN-on-Demand: An Affordable DDoS Defense via Untrusted Clouds. In *Proceedings of NDSS'16*. ISOC, 2016.

[51] Paul A Grassi, James L Fenton, Elaine M Newton, Ray A Perlner, Andrew R Regenscheid, William E Burr, Justin P Richer, Naomi B Lefkovitz, Jamie M Danker, Y Choong, et al. NIST Special Publication 800-63B: Digital Identity Guidelines. *National Institute of Standards and Technology (NIST)*, 27, 2016.

[52] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. KHAPE: Asymmetric PAKE from Key-Hiding Key Exchange. In *Proceedings of CRYPTO'21*, pages 701–730. Springer, 2021.

[53] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, Internet Engineering Task Force (IETF), 2012.

[54] Stephen Herwig, Christina Garman, and Dave Levin. Achieving Keyless CDNs with Conclaves. In *Proceedings of USENIX Security Symposium 2020*, pages 735–751. USENIX, 2020.

[55] Scott Hollier, Jania Sajka, Jason White, and Michael Cooper. Inaccessibility of CAPTCHA. *W3C Recommendation*, 2021.

[56] Zonghao Huang, Lujo Bauer, and Michael K Reiter. The Impact of Exposed Passwords on Honeyword Efficacy. In *Proceedings of USENIX Security Symposium 2024*. USENIX, 2024.

[57] Philip J. Nesser II, Mike Straw, Craig Metz, and Neil M. Haller. A One-Time Password System. RFC 2289, Internet Engineering Task Force (IETF), 1998.

[58] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. A Survey on Locality Sensitive Hashing Algorithms and Their Applications. *arXiv preprint arXiv:2102.08942*, 2021.

[59] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model. Cryptology ePrint Archive, Paper 2014/650, 2014.

[60] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks. In *Proceedings of EUROCRYPT'18*, pages 456–486. Springer, 2018.

[61] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E Gonzalez, Raluca Ada Popa, Ion Stoica, and David A Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. Technical report, University of California, Berkeley, 2019.

[62] Matt Klein. Introduction to Modern Network Load Balancing and Proxying. https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236, 2017.

[63] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Internet Engineering Task Force (IETF), 1997.

[64] Kat Krol, Eleni Philippou, Emiliano De Cristofaro, and M Angela Sasse. "They brought in the horrible key ring thing!" Analysing the Usability of Two-Factor Authentication in UK Online Banking. In *Proceedings of NDSS Workshop on Usable Security*. ISOC, 2015.

[65] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to Make TLS Middlebox-aware? In *Proceedings of NDSS'19*. ISOC, 2019.

[66] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *Proceedings of USENIX Security Symposium 2021*, pages 3847–3864. USENIX, 2021.

[67] Zhi Li, Hai Jin, Deqing Zou, and Bin Yuan. Exploring New Opportunities to Defeat Low-Rate DDoS attack in Container-based Cloud Environment. *IEEE Transactions on Parallel and Distributed Systems (TPDS'19)*, 31(3):695–706, 2019.

[68] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. When HTTPS meets CDN: A Case of Authentication in Delegated Service. In *Proceedings of S&P'14*, pages 67–82. IEEE, 2014.

[69] Shihan Lin, Rui Xin, Aayush Goel, and Xiaowei Yang. InviCloak: An End-to-End Approach to Privacy and Performance in Web Content Distribution. In *Proceedings of CCS'22*, pages 1947–1961. ACM, 2022.

[70] Xin Liu, Xiaowei Yang, and Yong Xia. NetFence: Preventing Internet Denial of Service from Inside out. In *Proceedings of SIGCOMM'10*, pages 255–266. ACM, 2010.

[71] Zhuotao Liu, Hao Jin, Yih-Chun Hu, and Michael Bailey. MiddlePolice: Toward Enforcing Destination-Defined Policies in the Middle of the Internet. In *Proceedings of CCS'16*, pages 1268–1279. ACM, 2016.

[72] Guillaume Marçais, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. Locality-Sensitive Hashing for the Edit Distance. *Bioinformatics*, 35(14):i127–i135, 2019.

[73] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks. In *Proceedings of USENIX Security Symposium 2018*, pages 393–410. USENIX, 2018.

[74] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford university, Stanford, CA, 1979.

[75] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and Confused: Web Cache Deception in the Wild. In *Proceedings of USENIX Security Symposium 2020*, pages 665–682. USENIX, 2020.

[76] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. Web Cache Deception Escalates! In *Proceedings of USENIX Security Symposium 2022*, pages 179–196. USENIX, 2022.

[77] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS# 5: Password-Based Cryptography Specification Version 2.1. RFC 8108, Internet Engineering Task Force (IETF), 2017.

[78] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And then There were More: Secure Communication for More than Two Parties. In *Proceedings of CoNEXT'17*, pages 88–100. ACM, 2017.

[79] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of SIGCOMM'15*, pages 199–212. ACM, 2015.

[80] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password Similarity Models Using Neural Networks. In *Proceedings of S&P'19*, pages 417–434. IEEE, 2019.

[81] Colin Percival and Simon Josefsson. The scrypt Password-Based Key Derivation Function. RFC 7914, Internet Engineering Task Force (IETF), 2016.

[82] Benny Pinkas and Tomas Sander. Securing Passwords Against Dictionary Attacks. In *Proceedings of CCS'02*, pages 161–170. ACM, 2002.

[83] Will Reese. NGINX: the High-Performance Web Server and Reverse Proxy. *Linux Journal*, 2008(173):2, 2008.

[84] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. Popularity is Everything: A New Approach to Protecting Passwords from Statistical-Guessing Attacks. In *Proceedings of HotSec'10*. USENIX, 2010.

[85] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking inside a Single Enclave of Intel SGX. In *Proceedings of ASPLOS'20*, pages 955–970. ACM, 2020.

[86] Jared M Smith and Max Schuchard. Routing Around Congestion: Defeating DDoS Attacks and Adverse Network Conditions via Reactive BGP Routing. In *Proceedings of S&P'18*, pages 599–617. IEEE, 2018.

[87] Robert Snelick, Umut Uludag, Alan Mink, Mike Indovina, and Anil Jain. Large-scale Evaluation of Multimodal Biometric Authentication Using State-of-the-art Systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI'05)*, 27(3):450–455, 2005.

[88] Nick Sullivan. Keyless SSL: The Nitty Gritty Technical Details. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/, 2014.

[89] San-Tsai Sun, Eric Pospisil, Ildar Muslukhov, Nuray Dindar, Kirstie Hawkey, and Konstantin Beznosov. What makes Users Refuse Web Single Sign-On? An Empirical Investigation of OpenID. In *Proceedings of SOUPS'11*, pages 1–20. USENIX, 2011.

[90] Rajat Tandon, Haoda Wang, Nicolaas Weideman, Shushan Arakelyan, Genevieve Bartlett, Christophe Hauser, and Jelena Mirkovic. Leader: Defense Against Exploit-Based Denial-of-Service Attacks on Web Applications. In *Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses (RAID'23)*, pages 744–758. ACM, 2023.

[91] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted Online Password guessing: An Underestimated Threat, 2016.

[92] Ke Coby Wang and Michael K Reiter. How to End Password Reuse on the Web. In *Proceedings of NDSS'19*. ISOC, 2019.

[93] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding Password Choices: How Frequently Entered Passwords are Re-Used Across Websites. In *Proceedings of SOUPS'16*, pages 175–188. USENIX, 2016.

[94] Mark Watson. Web Cryptography API. *W3C Recommendation*, 2017.

[95] Changzheng Wei, Jian Li, Weigang Li, Ping Yu, and Haibing Guan. STYX: A Trusted and Accelerated Hierarchical SSL Key Management and Distribution System for Cloud Based CDN Application. In *Proceedings of SoCC'17*, pages 201–213. ACM, 2017.

[96] Wei Wu, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. A Review for Weighted Minhash Algorithms. *IEEE Transactions on Knowledge and Data Engineering (TKDE'20)*, 34(6):2553–2573, 2020.

[97] Marc Wyss and Adrian Perrig. Zero-setup Intermediate-rate Communication Guarantees in a Global Internet. In *Proceedings of USENIX Security Symposium 2024*, pages 253–270. USENIX, 2024.

[98] Yunming Xiao, Yanqi Gu, Yibo Zhao, Sen Lin, and Aleksandar Kuzmanovic. Enabling Anonymous Online Streaming Analytics at the Network Edge. *ACM Transactions on Computer Systems (TOCS'25)*, 2025.

[99] Yunming Xiao and Matteo Varvello. FIAT: Frictionless Authentication of IoT Traffic. In *Proceedings of CoNEXT'22*, pages 156–170. ACM, 2022.

[100] Yunming Xiao, Yibo Zhao, Sen Lin, and Aleksandar Kuzmanovic. Snatch: Online Streaming Analytics at the Network Edge. In *Proceedings of EuroSys'24*, pages 349–369. ACM, 2024.

[101] Rui Xin, Shihan Lin, and Xiaowei Yang. Quantifying User Password Exposure to Third-Party CDNs. In *Proceedings of PAM'23*, pages 652–668. Springer, 2023.

[102] Ming Xu, Jitao Yu, Xinyi Zhang, Chuanwang Wang, Shenghao Zhang, Haoqi Wu, and Weili Han. Improving Real-world Password Guessing Attacks via Bi-Directional Transformers. In *Proceedings of USENIX Security Symposium 2023*, pages 1001–1018. USENIX, 2023.

[103] Xiaowei Yang, David Wetherall, and Thomas Anderson. A DoS-Limiting Network Architecture. In *Proceedings of SIGCOMM'05*, pages 241–252. ACM, 2005.

[104] Zhuolin Yang, Yuxin Chen, Zain Sarwar, Hadleigh Schwartz, Ben Y Zhao, and Haitao Zheng. Towards a General Video-based Keystroke Inference Attack. In *Proceedings of USENIX Security Symposium 2023*, pages 141–158. USENIX, 2023.

[105] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. OS-Level Side Channels without Procfs: Exploring Cross-App Information Leakage on IOS. In *Proceedings of NDSS'18*. ISOC, 2018.

[106] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G Andersen. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In *Proceedings of S&P'11*, pages 212–227. IEEE, 2011.

[107] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud. In *Proceedings of ASPLOS'24*, pages 582–600. ACM, 2024.

[108] Shengbao Zheng and Xiaowei Yang. DynaShield: Reducing the Cost of DDoS Defense using Cloud Services. In *Proceedings of ATC Workshop on Hot Topics in Cloud Computing*. USENIX, 2019.

A PreAcher Registration Protocol

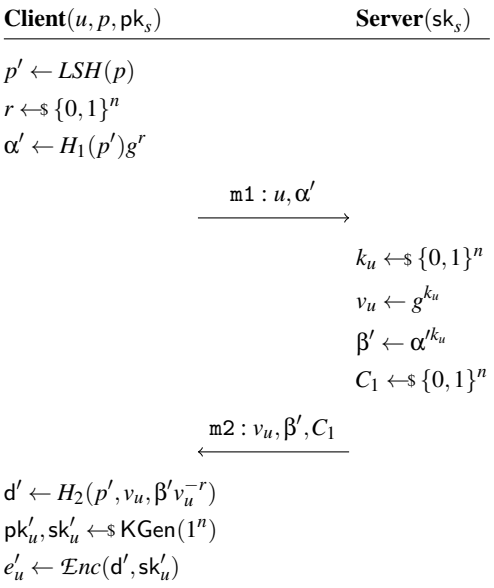


Figure 8: **First round of PreAcher registration.**

In the registration, the client communicates to the server through a CDN. The CDN simply forwards the messages and is not involved in the protocol. As we discuss in § 2.3, the registration can be considered as ADoS-resist in reality, but the CDN can observe the message content. At the end of the registration, the server will send some registration information without exposing user passwords to the CDN so that the CDN can use such information for pre-authentication. The registration protocol works as follows:

Initialization. A user inputs the username u and password p to the client. As discussed in 3.1, the client obtains the server's public key pk_s from the HTML of the login page, and the server keeps the corresponding private key sk_s.

Generating a symmetric key from the password. The client and server compute a symmetric key d' together through OPRF. This step combines the OPRF with LSH as shown in Figure 8.

Specifically, the client uses LSH to map p to p'. Then it generates a random nonce r and computes α' used by OPRF. The client sends u and α' to the server for registration (`m1`). The server generates a user-specific k_u and computes v_u, β' from α' and k_u. The server also generates a one-time random challenge C_1 to verify the client's identity in the next round of communication. Then v_u, β', and C_1 is returned to the client (`m2`). Thus, the client computes a secret symmetric key d' through H_2 and d' is hidden from the server. Note that it can be proved that d' is independent from the choice of r.

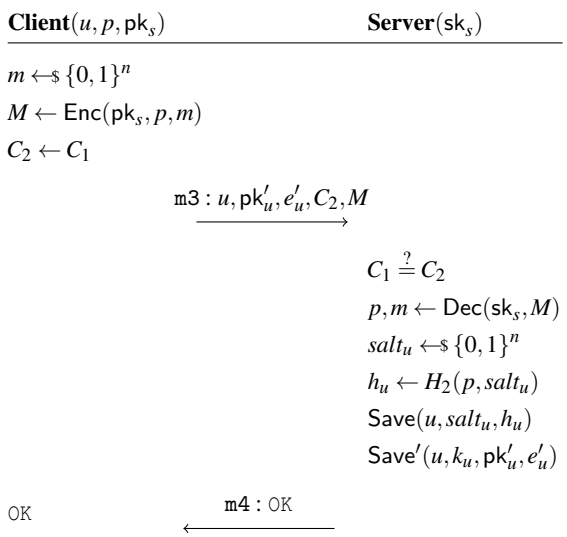


Figure 9: **Second round of PreAcher registration.**

Enclosing a new private key with the symmetric key. The client generates a new key pair $\langle \mathsf{sk}'_u, \mathsf{pk}'_u \rangle$ for the user. Then sk'_u is enclosed into an envelope e'_u by encryption with d'. This new key pair and e'_u will be used for pre-authentication by the CDN in the login.

Registing with the server for full authentication. Finally, the client and the origin server register the user through the traditional password hashing for compatibility with existing authentication in websites. Note that since we trust the origin server (§ 2.3), we can expose the password to the server. Nevertheless, the password sent to the server is encrypted by the server's public key to avoid exposure to the CDN.

As shown in Figure 9, the client uses pk_s to encapsulate p with a random nonce m. The client copies received C_1 as C_2 and sends it back to the server with pk'_u, e'_u, and M (m3). The server first needs to check C_2 against C_1 to identify the client as the one who initiated the registration in the previous round. The server obtains p by decrypting M with its own sk_s. Then it computes the corresponding salted hash value h_u from p. The server saves $salt_u$ and h_u in a user-specific record for future full authentication. Besides, the server saves $k_u, \mathsf{pk}'_u, e_u, e'_u$ and sends them to the CDN for the future pre-authentication ($\mathsf{Save}'(u, k_u, \mathsf{pk}'_u, e'_u)$). Then the server returns the indication of a successful registration (m4) to the client.

B Attack Attempts and Defense

PreAcher can defend against a passive attacker inside the CDN. To make this tangible, we provide two examples of attack attempts to demonstrate the effectiveness of PreAcher's defense.

For instance, an attacker inside the CDN can observe M in $m4$ (Figure 3) during a valid user login. The attacker may attempt to guess the password offline by computing a M value with each guessed password and validating it against the observed M. PreAcher defends against this attack because M is generated by the client using not only the password but also a random nonce m, forcing the attacker to guess m from a vast space in order to obtain a M value that matches the observed one.

Moreover, an attacker may replay the collected M from a valid login and send this M to the server for full authentication. However, the server can prevent such replay attacks by recording the recently received M and thus reject duplicate ones, which is already adopted by prior work [69].

C DuoHash Protocol

We use the same notations described in § 4.1. Figure 10a illustrates the protocol of registration by following steps:

1. The user inputs the username u and password p. Then the client encrypts the password with the server's public key pk_s and sends the username u and encrypted messages M to the server.
2. The origin server decrypts M with its private key sk_s to obtain the password. The server stores a salted hash value h_u for future authentication. Besides, the server calculates another hash value h'_u, which will be used by the CDN for pre-authentication. To avoid offline dictionary attacks by the CDN, the server first uses LSH to map the p to a value p'. Finally, the server uses two layers of hashing H_2 and H_1 to map p' into h'_u, and it is an indication of successful registration to the client.

After the registration, the server sends h'_u and $salt'_u$ to the CDN (not shown in the figure). The CDN stores h'_u and $salt'_u$ for future pre-authentication in the login phase.

The protocol in Figure 10b shows the login phase by following steps:

1. Similar to the registration, the client has u and p and creates the encrypted password M with pk_s. In addition, the client uses LSH to generate p' and the corresponding first-layer hash value h'_1, which is sent to the CDN with u and M.
2. The CDN retrieves $salt'_u$ and h'_u from the user-specific record according to username u. It then conducts the authentication by checking whether $H_2(h', salt'_u)$ is equal to h'_u. If they are equal, the CDN considers the password pre-authentication succeeds and then sends u and M to the origin server for further authentication. If the pre-authentication fails, the CDN directly returns an indication of failed login to the client (not shown in the figure), and thus filters out most failed login requests for the origin server.
3. When the server receives an authentication request from the CDN, it decrypts M with the private key to extract the password p. The server authenticates the user by comparing h_u with the computed $H_2(p, salt_u)$. The server returns the final authentication results to the CDN.
4. The CDN directly forwards back the authentication result to the client.

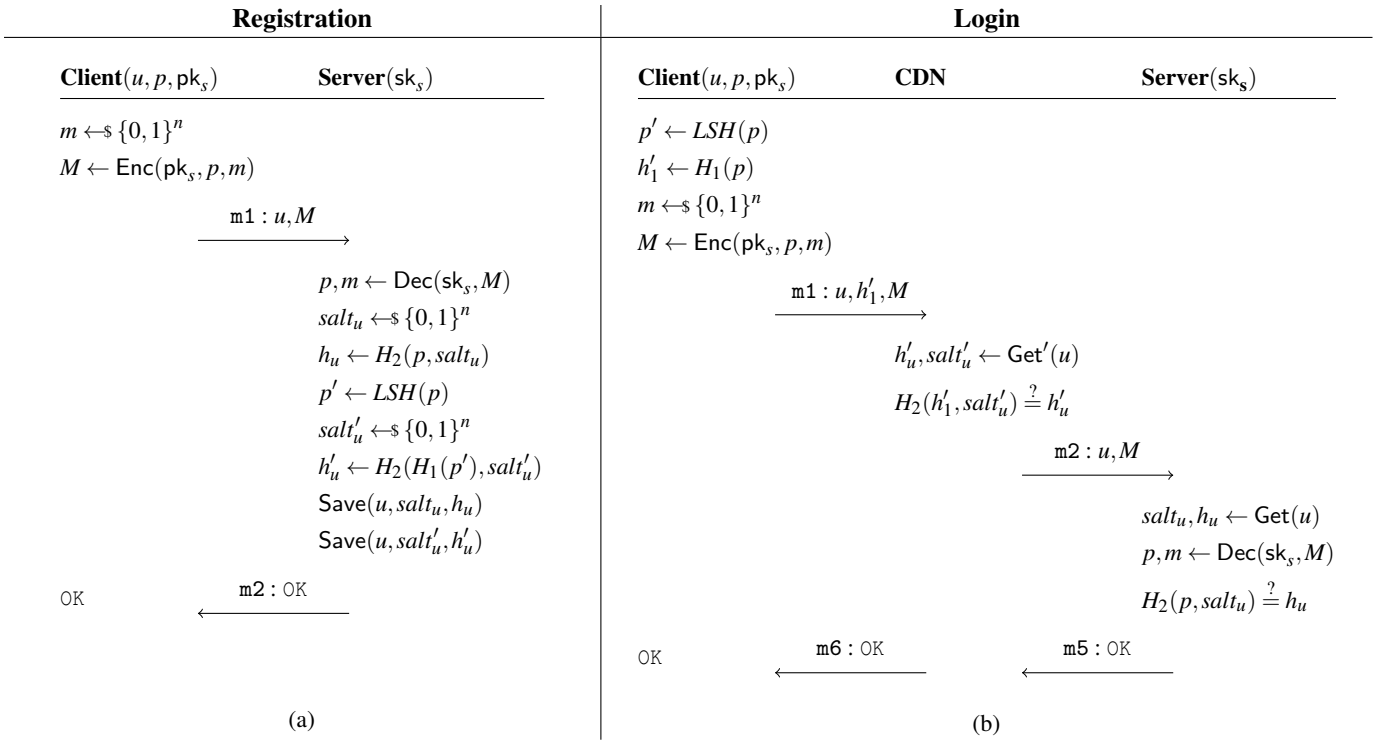


Figure 10: **DuoHash Protocol**

D Security Proof Sketch

In this section, we present a proof sketch to show the security of our protocol. The security requirements are discussed in Section 6. For clarity, we show the complete flow of PreAcher authentication protocol in Figure 11. We use the following probability $\varepsilon_{\mathsf{oprf.random}}$, $\varepsilon_{\mathsf{CPA}}$, $\varepsilon_{\mathsf{CCA}}$, $\varepsilon_{\mathsf{EU-CMA}}$ to denote adversarial advantage of winning the pseudorandomness game of OPRF, the CPA security game, CCA security game, unforgeability game of signature scheme, respectively, which are all negligible in terms of security parameter n.

Theorem 1. *PreAcher is secure against an external attacker, assuming a secure OPRF, a CPA-secure encryption Enc, a CCA-secure encryption* Enc*, and an EU-CMA-secure signature.*

Proof. First, we consider a passive external attacker, and if this attacker provides a wrong password and obtains a p' to authenticate with CDN, there can be two cases:

(1) p' is incorrect. In this case, the attacker gets a random d'^* which is independent from the correct d'. The adversarial advantage of distinguishing this d'^* from d' can be reduced to the adversarial advantage of winning the pseudorandomness game of OPRF[2] and thus is bounded by $\varepsilon_{\mathsf{oprf.random}}$. And since the attacker doesn't know the correct d', the adversarial advantage of distinguishing sk'_u is bounded by the adversarial advantage of winning the semantic security game of the encryption scheme, i.e. $\varepsilon_{\mathsf{CPA}}$. Without knowing the correct sk'_u, the probability of the attacker issuing/forging a valid signature can be reduced to the probability of the attacker winning the unforgeability game of the signature scheme, which is bounded by $\varepsilon_{\mathsf{EU-CMA}}$.

(2) p' is actually correct. This happens because the password guess is close to the correct one and thus gets mapped into the correct p' by LSH with probability $\varepsilon_{\mathsf{LSH}}$. Since p' is correct, the attacker can successfully authenticate with CDN. However, to authenticate with the origin server, the attacker needs to construct a valid adversarial ciphertext M^* from the honest M. This case can be reduced to the CCA security game and thus bounded by ε_{CCA}.

To summarize, the probability of an external adversary breaking the security of PreAcher is negligible. □

Theorem 2. *PreAcher is secure against a passive CDN attacker, assuming a secure OPRF, and a CCA-secure encryption* Enc*.*

Proof. Second, we consider the case of a malicious CDN. Here we no longer consider adversarial authentication to CDN and mainly focus on authentication to the origin server.

Because this malicious CDN holds OPRF key k_u, it can attempt to launch an offline dictionary attack by running OPRF locally multiple times and computing a dictionary D of d's corresponding to its password guesses, and try to decrypt

[2]Our protocol implements a secure OPRF assuming H_2 is a secure hash function, as proved by OPAQUE [60]

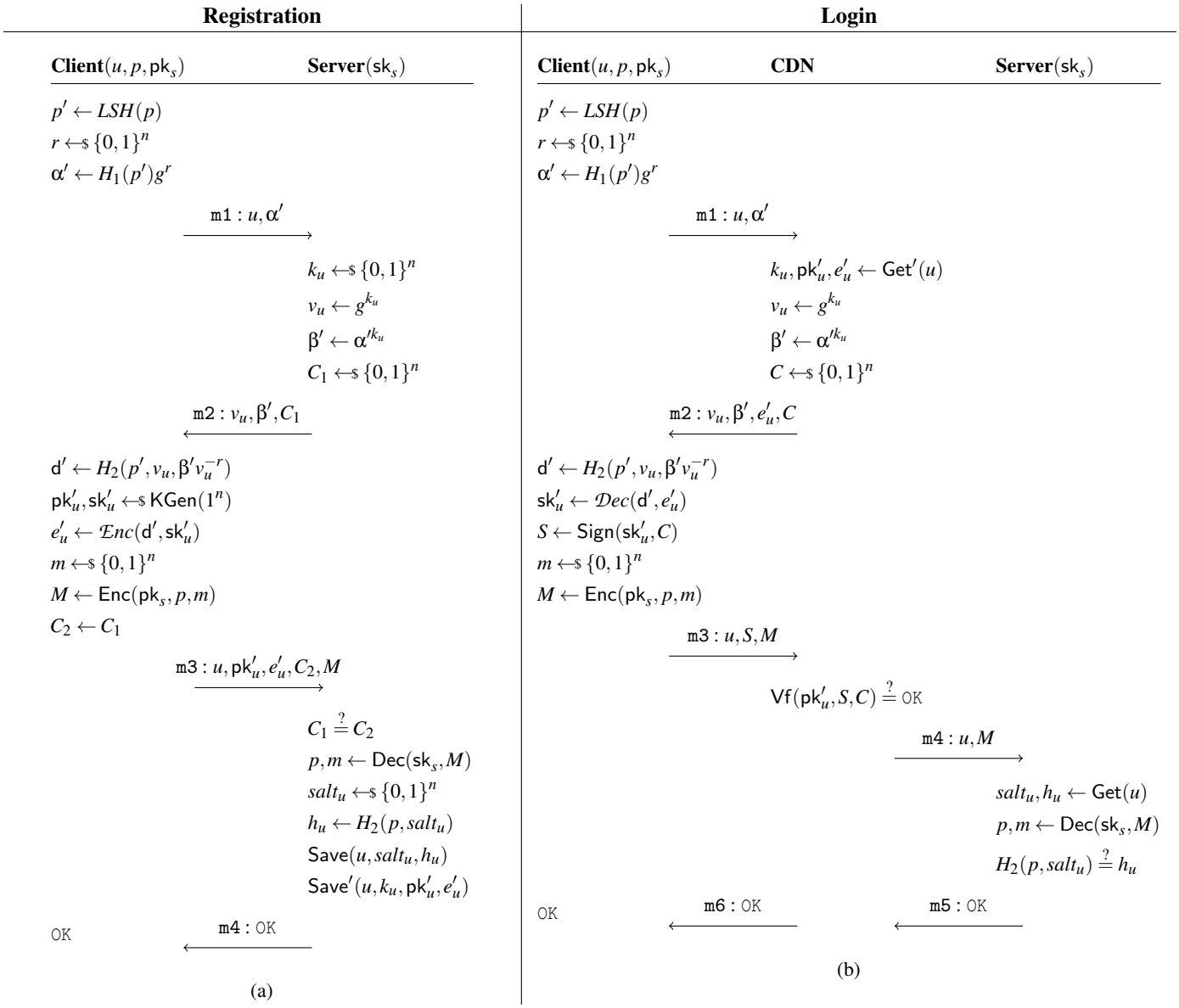


Figure 11: **PreAcher Protocol**

e_u with these d's. However, by the decryption randomness of $\mathcal{D}ec$, any wrong decryption key can only decrypt e_u to a random bitstring, which in the attacker's view is indistinguishable from the correct sk'_u, and the attacker cannot tell from this decryption result whether it uses the correct password or not. The probability of successfully launching such an offline dictionary attack is thus bounded by $\frac{1}{|2^n|} \cdot \frac{1}{|D|}$.

To summarize, the probability of a passive CDN adversary breaking the security of PreAcher is negligible. □

ClubHeap: A High-Speed and Scalable Priority Queue for Programmable Packet Scheduling

Zhikang Chen
Tsinghua University

Haoyu Song
Futurewei Technologies

Zhiyu Zhang
Fudan University

Yang Xu
Fudan University

Bin Liu
Tsinghua University

Abstract

While PIFO is a powerful priority queue abstraction to support programmable packet scheduling in network devices, the efficient implementation of PIFO faces multiple challenges in performance and scalability. The existing solutions all fall short of certain requirements. In this paper, we propose ClubHeap to address the problem. On the one hand, we develop a novel hardware-friendly heap data structure to support faster PIFO queue operations that can schedule a flow in every clock cycle, reaching the theoretical lower bound; on the other hand, the optimized hardware architecture reduces the circuit complexity and thus enables a higher clock frequency. The end result is the best scheduling performance in its class. Combined with its inherently better scalability and flexibility, ClubHeap is an ideal solution to be built in programmable switches and SmartNICs to support various scheduling algorithms. We build an FPGA-based hardware prototype and conduct a thorough evaluation by comparing ClubHeap with the other state-of-the-art solutions. ClubHeap also allows graceful trade-offs between throughput and resource consumption through parameter adjustments, making it adaptable on different target devices.

1 Introduction

In a network switch, packet scheduling, which determines the packet delivery order and time, is an essential function in Traffic Manager (TM). To satisfy the diverse application requirements, a number of packet scheduling algorithms have been proposed [6,7,15,21,22,25,31,35,45,63,65], e.g., Weighted Fair Queueing (WFQ) [15], Hierarchical Packet Fair Queueing (HPFQ) [6], Least Slack Time First (LSTF) [31], etc. Fine-grained priority scheduling [3,34,46] based on specific fields, e.g., Shortest Remaining Processing Time (SRPT) [46], is also widely used.

As the programmable network devices play a more important role in Software-Defined Networking (SDN), the programmable schedulers in those devices can support different scheduling algorithms on generic hardware. A popular queue abstraction, Push-In First-Out (PIFO) [51], has been accepted as a foundation to realize a wide range of algorithms. A PIFO is a Priority Queue (PQ) which allows elements to be pushed to an arbitrary location, but can only pop elements from the head. A series of PIFO blocks with mesh interconnections can be configured into a tree to support multi-level hierarchical scheduling. Each PIFO block contains a physical PQ which can be partitioned into multiple logical PQs (named logical PIFOs) at the same level of the tree. In a PIFO block, a flow is inserted in a queue location based on its *rank* [51], the calculated scheduling order or time (e.g., the remaining flow size in SRPT [46]), and popped when it reaches the queue head.

A PIFO queue needs to support two primitives: *push* and *pop*. In a PIFO-based scheduler, a popped flow is immediately pushed back into the queue if it still has unscheduled packets [51], forming a *pop-push* pair. We introduce a *replace* primitive to represent the pair. During scheduling, all operations for a flow are *replace* except for the first *push* and the last *pop*. Therefore, we can use Cycles-Per-Replace (CPR) as a metric to evaluate the throughput performance under the same clock frequency. In pipelinable implementations, we consider the delay between two consecutive *replace* operations as the equivalent CPR because the operations may overlap.

Reducing CPR is an effective means to increase throughput, because the clock frequency is limited in silicon chips. The original PIFO implementation [51] uses shift registers (referred to as SR-PIFO henceforth), in which both *push* and *pop* operations require 2 cycles to complete (i.e., CPR$=4$). Some PIFO queue implementations support parallel *push* and *pop* operations [5,8,26,28,62], enhancing native *replace* support. For example, RPU-BMW [62] allows a *push* every 2 cycles, a *pop* every 3 cycles, but a *replace* every 3 cycles (CPR$=3$) due to its pipeline design. Nevertheless, thus far no scalable PIFO implementation can achieve CPR$=1$ which is the lower limit of a PIFO-based packet scheduler.

In this paper, we propose a novel data structure called

ClubHeap (short for Clustered Binary Heap)[1] to support PIFO queue implementation, and a high-speed and scalable programmable scheduler solution based on it. Our major contributions are summarized as follows.

1. We design ClubHeap, a new PQ data structure supporting the PIFO abstraction. ClubHeap leverages the scalability of the heap-based data structures, and overcomes the challenge of inter-operational data dependencies troubling the other heap-based data structures [8, 26, 28, 62]. ClubHeap is the first heap-based design which supports a fully pipelined implementation with the theoretical lower bound of CPR= 1.

2. To validate the design, we synthesize ClubHeap with a 45nm ASIC toolchain [17, 23, 55] by Design Compiler [53] and implement it on a Xilinx Alveo U280 Data Center Accelerator Card [60]. The prototype supports up to 2^{17} elements in up to 2^8 logical PIFOs, with each element having a priority precision of 32 bits. The prototype achieves a throughput of approximately 200Mpps on FPGA, ensuring 100Gbps line-rate processing in the worst case. We show ClubHeap is the only solution which is scalable in terms of elements, priority levels, and logical PIFOs at the same time.

The rest of the paper is organized as follows. In Sec. 2, we discuss the gap between the requirements of a programmable packet scheduler and the existing hardware implementations using priority queues. In Sec. 3 we introduce the ClubHeap data structure. We present its hardware design in Sec. 4 and its implementation in Sec. 5. A detailed evaluation including comparisons with existing implementations is presented in Sec. 6. We summarize the related works in Sec. 7 and conclude the paper in Sec. 8.

2 Background

2.1 Requirements for PIFO Implementations

The PIFO abstraction can be applied to both programmable switches [10, 50] and SmartNICs [33, 52]. Recent research shows that the priority queues in these two scenarios share the similar requirements as follows [4].

Throughput. Due to the rapid growth of port speed and count on network devices, a single scheduler instance cannot schedule all packets at line rate. For example, a 64x400GE NVIDIA Spectrum SN5400 switch [41] supports up to 33.3 billion packets per second, exceeding any known scheduler capabilities. Therefore, a mesh of PIFO queues is needed for high aggregated scheduling throughput. Fewer queues are needed if each PIFO queue has higher throughput. SmartNICs have a lower peak packet rate than switches, allowing the use of a single PIFO queue. Some recent PIFO queue implementations (e.g., BBQ [4]) can meet the demands of FPGA-based SmartNICs operating for 100GE (148.8Mpps). However, increasing NIC throughput also puts more pressure on the packet scheduler.

Scalability. Scalability is measured in two dimensions: the number of priority levels (P) and the number of elements (N) allowed in the priority queue. In a PIFO queue, the rank of a flow might be a timestamp (e.g., in LSTF [31] or EDF [34]), which often requires high precision. A large P also extends the time span for data retention in the scheduler under non-work-conserving algorithms [21, 29, 57, 67]. On the other hand, today's multi-tenant networks may require the scheduler to handle hundreds of thousands of flows [18, 19, 43], which pose significant scalability challenges for N.

Logical Partitioning. Logical Partitioning (LP) refers to partitioning a physical PIFO block into multiple logical PIFOs. In hierarchical scheduling algorithms, queues at the same level of the scheduling tree can be considered as different logical PIFOs within the same PIFO block. The same applies to the packet queues for different output ports.With LP, a k-port switch requires only $2k$ physical PIFOs, compared to k^2+k PIFOs without LP [4], resulting in a 32.5x reduction for k=64. The significant difference in resource consumption makes LP crucial for the implementation of PIFO-based schedulers.

2.2 Existing Implementations

The requirement for high-performance PQs to support packet scheduling precedes the PIFO abstraction. Numerous PQ implementations exist, but none satisfies all three requirements in Sec. 2.1. We brief three typical implementations below, which can all be used in a PIFO-based scheduler and programmed using Domino [37, 50, 51].

Implementations based on linear data structures. Implementations based on shift registers [9, 13, 42, 51, 54] (e.g., SR-PIFO [51]) and systolic arrays [5, 30, 38] (e.g., OPQ and APQ [5]) fall into this category. For an operation, a shift register performs all comparisons within one cycle, whereas a systolic array breaks it down into multiple sub-operations, and propagates them linearly through the entire data structure in a systolic manner, thereby simplifying the combinational logic for each cycle and improving the clock frequency. APQ, with the highest throughput among them, organizes elements into groups, and leverages the Single-Instruction-Multiple-Data (SIMD) technique to achieve CPR= 1. However, these implementations need $O(N)$ comparators to compare the elements[2] in the PQ in parallel, resulting in significant hardware overhead, and a decrease in clock frequency as N increases. Therefore, these implementations cannot meet the scalability requirements for N. For example, SR-PIFO requires approximately 900K LUTs on an FPGA (∼70% on Xilinx Alveo

[1] Open source available at https://github.com/ClubHeap/ClubHeap.

[2] Each queue element includes a rank and some metadata.

PQ Implementation	Type	CPR	S.P	S.N	LP
SR-PIFO [51]	linear	4	✓	×	✓
OPQ [5]	linear	2	✓	×	×
APQ [5]	linear	1	✓	×	×
BBQ [4]	bucket	2	×	✓	✓
P-Heap [8]	heap	6	✓	✓	×
R-BMW [62]	heap	2	✓	×	×
RPU-BMW [62]	heap	3	✓	✓	×
Pipelined Heap [28]	heap	2	✓	×	✓
H-PQ [26]	hybrid	1∼4	✓	✓	×
ClubHeap	heap	1	✓	✓	✓

Table 1: Comparison of different PQ implementations. (**S.**P and **S.**N represent the scalability for P and N, respectively)

U280 [60]) when $N = 4,096$, and can only operate at a frequency of 40MHz [62]. With CPR= 4, this translates to a throughput of only 10Mpps.

Implementations based on buckets. These implementations assign a bucket to each priority level and group flows with the same priority into the same bucket. BBQ [4] introduces Hierarchical Find-First Set (HFFS) [44, 58] to manage buckets and store elements in SRAMs, achieving scalability with respect to N. However, dedicated hardware for each priority level limits scalability for P. For example, BBQ provides 2^{15} buckets in total [4]. Considering that a PIFO block supports 256 logical PQs (a typical number in [51]), each logical PQ can only support P=2^{15}/256=128, which is unacceptable for many algorithms [6, 22, 31].

Implementations based on heaps. A heap is a tree in which the rank of each node is not greater than that of its child nodes. The operation on a heap-based structure resembles a systolic array but positions sub-operations at distinct tree levels. This design requires only $O(\log N)$ comparators, each with $O(\log\log P)$ latency. P-Heap [8] is the first to exploit the heap structure, achieving scalability for both P and N. BMW-Tree [62] introduces an insertion-balanced multi-way heap with better pipeline support to reduce CPR from 6 to 3. However, both P-Heap and BMW-Tree do not support logical partitioning. Pipelined Heap [28] implements a complete binary heap which can support logical partitioning. However, the global bus required to connect all tree levels becomes a bottleneck in the pipeline, limiting the frequency and making it less scalable with respect to N.

Another challenge for heap-based implementations comes from the inter-operational data dependency problem (detailed in Sec. 2.3), which prevents a fully pipelined implementation with CPR= 1. H-PQ [26] is a hybrid structure of systolic array and a series of heaps, which attempts to eliminate this impact by distributing operations across different heaps. Unfortunately, real traffic may access the same heap continuously, causing the performance instability of the packet scheduler. ClubHeap solves the inter-operational data dependency problem through a novel data structure design.

The characteristics of the aforementioned implementations are summarized in Table 1. We use CPR as the performance indicator for simplicity[3][4]. In summary, implementations based on linear data structures have poor scalability for N, while implementations based on buckets have poor scalability for P. ClubHeap, as a heap-based implementation, achieves a high performance with CPR= 1, presents good scalability for both N and P, and possesses the logical partitioning capability.

2.3 Inter-Operational Data Dependency

Pipelining the queue operation is critical for the scheduling throughput. However, the inter-operational data dependency is a hurdle that prevents the heap-based PIFOs from applying the pipeline technique efficiently. Its solution is the key factor for a high-performance implementation of ClubHeap.

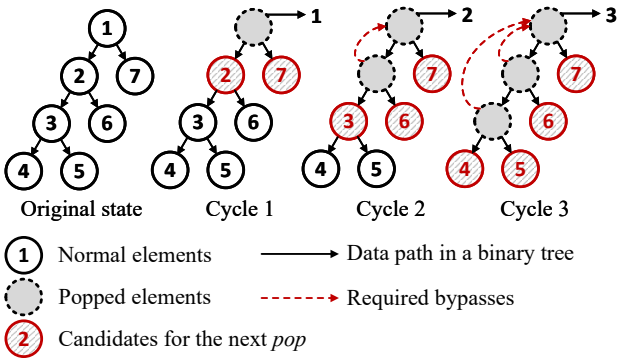


Figure 1: The inter-operational data dependency on a traditional binary heap.

Since it is impossible to predict the rank of the subsequent elements pushed into the PIFO when pushing a current element, the distribution of elements in the heap can be unbalanced. As shown in Fig. 1, in a binary heap, the distribution of the smallest ranked elements may form a dependency chain: 1 is the parent of 2, 2 is the parent of 3, and so on.

Consider consecutive *pop* operations under this circumstance. In Cycle 1, the element 1 at Level 1 (i.e., the root node) is popped, leaving a temporarily empty root, which means both elements stored in its child nodes can be the new minimum element in the heap. Therefore, a comparison must be made between the element 2 and the element 7 at Level 2. In Cycle 2, the second *pop* arrives, which needs to pop the element 2, the smaller one between 2 and 7. However,

[3] Besides CPR, the scheduling throughput is also affected by clock frequency, which may vary for different implementations and different scale specifications (i.e., P and N).

[4] P-Heap claims CPR=2 [8], but it allocates three dependent operations (read, compare, and write) in one cycle. For a fair comparison, we convert one P-Heap cycle to three physical clock cycles as suggested in [28].

the newly arriving instruction just reaches the root node and cannot operate on the element 2 which is still located at Level 2. In order to allow the element 2 to be popped immediately, a bypass from Level 2 to Level 1 (as depicted in the dashed line in Fig. 1) is required. After the element 2 is popped, its two children at Level 3 become the new candidates as well as the element 7. The comparison to elect the next minimum must be made among the three candidates. Similarly, the next *pop* operation arriving in Cycle 3 requires another bypass from Level 3 to Level 1 and increases the number of candidates to four. If the heap has L levels, $L-1$ bypasses are needed from the other levels to Level 1, and a comparison between L candidates needs to be made after $L-1$ consecutive *pop* operations.

In the case of $L = 16$ (i.e., supporting $N = 2^{16} - 1$) and $P = 2^{16}$, the operation to compare 16 16-bit element values and select the minimum one requires a complex combinational circuit, which can severely affect the achievable clock frequency. The wire delay introduced by the bypasses from Level 2, 3, ..., L to Level 1 worsens the problem. In other words, to guarantee a certain performance level, the heap depth is constrained, resulting in poor scalability for N. Because of this, the existing heap-based priority queue implementations [8, 28, 62] are difficult to reconcile with performance and scalability simultaneously.

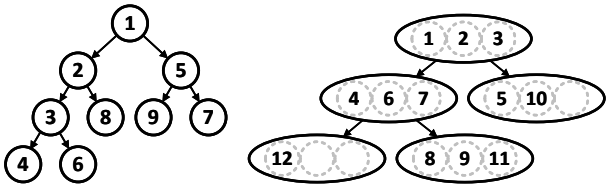
3 ClubHeap

In a traditional heap [8, 28, 62], an operation proceeds along the path from the root to a certain node in the tree structure, resembling a systolic array. Inspired by the SIMD technique where APQ [5] achieves CPR= 1, we organize elements into *clusters* (different from *groups* in APQ, elements in a cluster are rank ordered), so that consecutive operations may only access different elements within a cluster, thereby eliminating the inter-operational data dependency problem.

3.1 Definition and Corollaries

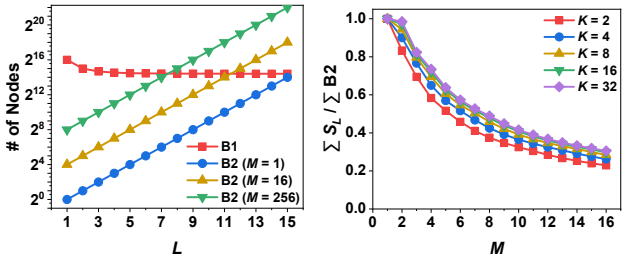
As shown in Fig. 2(a), a traditional binary heap (used in P-Heap [8] and Pipelined Heap [28]) stores a PQ element in each node of the binary tree. Denote the rank of the element on node x as $E(x)$, and consider the rank of an empty node as $+\infty$. Then for any child y of any node x, it holds that $E(x) \le E(y)$.

The key difference between ClubHeap and a traditional binary heap is that each binary tree node in ClubHeap holds more than one element which forms a *cluster*. Each node can store up to K sorted elements. A node that contains K elements is called *full*. For a *non-full* node containing T ($T < K$) elements, the $K-T$ free slots are considered to have the $+\infty$ rank value. We denote the rank array of node x as $E(x)$. For nodes x and y, if $\max(E(x)) \le \min(E(y))$, i.e., the rank of any element in node x is less than or equal to any element in node y, we denote that $E(x) \le E(y)$.



(a) Traditional binary heap. (b) ClubHeap (K = 3).

Figure 2: Comparison of a traditional binary heap and a ClubHeap.



(a) **B1** vs **B2**. (b) S_L vs **B2**.

Figure 3: Comparison of **B1** given by Corollary 2 and **B2** given by the binary tree structure.

Definition. A binary tree is a ClubHeap if and only if: for any child y of a node x, it holds that $E(x) \le E(y)$.

Corollary 1. The definition leads to a corollary: for any child node y of a node x in a ClubHeap, if x is not full, then y must be empty. This is because, for a non-full x, $\max(E(x)) = +\infty$; according to the ClubHeap definition, $\max(E(x)) \le \min(E(y))$, so $\min(E(y)) = +\infty$, meaning that all items in $E(y)$ have the value of $+\infty$ and thus y must be empty.

The corollary implies that the elements in a ClubHeap are always compactly located in the nodes of the upper levels, and the minimum ranked elements must be located at the root node. The corollary below quantitatively describes the concentration of elements in the upper levels of ClubHeap.

Corollary 2. For a ClubHeap containing N elements, the number of non-empty nodes at the L-th level is no more than $\frac{N}{K\left(1-\frac{1}{2^{L-1}}\right)+1}$. The reason is as follows. Assume there are N_L non-empty nodes at the L-th level, so there are at least N_L elements at this level. Meanwhile, due to Corollary 1, there are at least $\frac{1}{2^i} \cdot N_L$ full nodes at the $(L-i)$-th level (i=1,2,...,L-1). Therefore, these nodes contain at least $K \cdot \left(1-\frac{1}{2^{L-1}}\right) \cdot N_L$ elements. Summarizing the two parts of elements, we have $N \ge N_L + K \cdot \left(1-\frac{1}{2^{L-1}}\right) \cdot N_L$, and thus $N_L \le \frac{N}{K\left(1-\frac{1}{2^{L-1}}\right)+1}$.

The concentration of elements is a key factor for ClubHeap to support LP. Consider a ClubHeap forest consisting of M logical PIFOs, with a total of N elements. According to Corol-

lary 2, the total number of nodes at the L-th level across all M logical PIFOs has an upper bound $\frac{N}{K\left(1-\frac{1}{2^{L-1}}\right)+1}$ (**B1**), which may be tighter than the upper bound given by M independent binary trees, i.e., $M \times 2^{L-1}$ (**B2**). Fig. 3(a) presents a comparison of **B1** and **B2** when $K = 2$ and $N = 2^{16} - 2$ (i.e., $L = 15$). It can be observed that **B1** effectively controls the number of nodes at deeper levels.

The memory space required by ClubHeap at the L-th level can be set as $S_L = \min(\mathbf{B1}, \mathbf{B2})$. Fig. 3(b) shows the ratio of the space required by ClubHeap to the space of M physical PIFOs, namely $\sum S_L / \sum \mathbf{B2}$, for different values of K and $N = 2^{16} - K$ (i.e., $L = 16 - \log K$). When $M = 16$, $\sum S_L$ is only 22.9%∼30.5% of $\sum \mathbf{B2}$, indicating the memory efficiency of **B1**. A smaller K leads to a smaller ratio. In summary, **B1** ensures that the memory required by ClubHeap to store nodes does not increase linearly with M, providing better support for LP than BBQ [4].

3.2 Operations

The *push* and *pop* operations on ClubHeap are similar to those on the other heap-based PIFO queues [8, 62]. For the *push* operation, if the root node r is not full, the element is inserted in it directly. Otherwise, the maximum ranked element in r (also including the newly inserted element) is removed and inserted into a child node in the next level. The recursive process continues until the element is settled. For the *pop* operation, the minimum ranked element in the root node r is popped, and then the minimum ranked element among the two children (if non-empty) is raised to fill the node r. The process is recursive at the deeper levels until no more elements are available to promote.

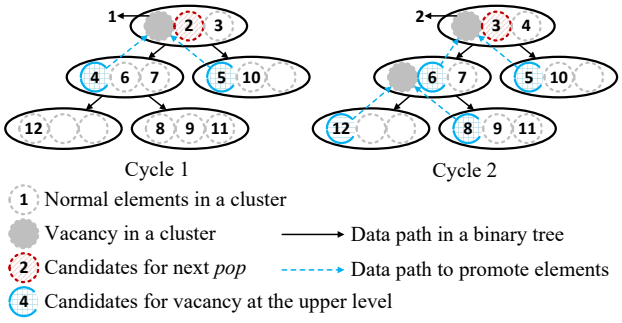


Figure 4: The consecutive *pop* operations on a ClubHeap with no inter-operational data dependency problem.

ClubHeap solves the inter-operational data dependency problem by decomposing the recursion into a sequence of sub-operations without data dependencies. As shown in Fig. 4, we perform consecutive *pop* operations on the ClubHeap in Fig. 2(b). In Cycle 1, element 1 is popped, leaving a vacancy in the root node. Elements 4 and 5 on Level 2 are candidates for promotion. According to ClubHeap's definition, the rank of the promoted element is not less than the remaining elements, so the *pop* operation in Cycle 2 does not depend on the promoted element. In Cycle 2, the element 2 is popped, the element 4 is promoted to Level 1, and elements 12 and 8 on Level 3 compete for the vacancy left on Level 2. Recursively, ClubHeap ensures that each node stores at least $K-1$ smallest ranked elements in its subtree. As long as $K \geq 2$, it can eliminate inter-operational data dependencies between sub-operations on different levels, and ensure that only data paths between adjacent levels are used, thereby reducing pipeline complexity and wire latency.

4 Architectural Design

In the architectural design of ClubHeap, our goal is to achieve CPR= 1 and high clock frequency. We design a pipeline to support CPR= 1, and employ a series of techniques to strive for simplifying the combinational logic circuit in each clock cycle in order to enhance the clock frequency.

In this section, we first present the pipeline architecture in Sec. 4.1 and the technique for efficient cluster storage in Sec. 4.2. We then discuss how the three primitives, i.e., *push*, *pop*, and *replace*, fit into the design in Sec. 4.3. Finally, we discuss the dynamic memory allocation optimization to support the flexible logical partitioning in Sec. 4.4.

4.1 Pipeline Overview

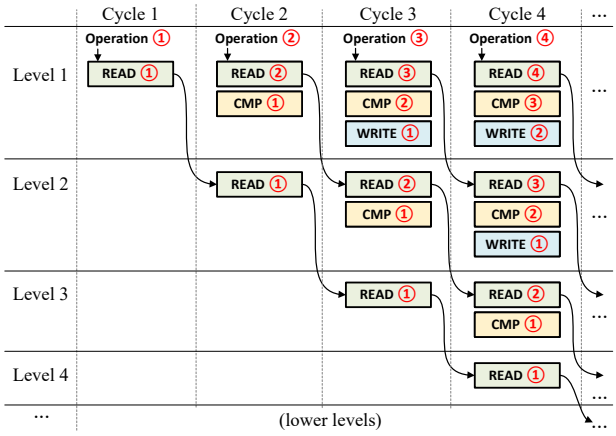


Figure 5: Overview of the ClubHeap pipeline.

In ClubHeap, each primitive, *push*, *pop*, or *replace*, undergoes three stages at each level of the heap: READ, CMP (short for Compare), and WRITE. These stages are executed in different clock cycles to boost clock frequency. Fig. 5 shows the pipeline design of ClubHeap, where four arbitrary primitive operations ①∼④ enters the pipeline sequentially from Cycle 1 to Cycle 4.

When a signal for an operation arrives at a level, it goes through the three stages in sequence. For example, ① arrives at Level 1 in Cycle 1, so its READ stage is performed in Cycle 1, reading the cluster at the root node. In Cycle 2, ① proceeds to the CMP stage at Level 1, comparing the elements within the read cluster. If ① carries an element (*push* or *replace*), it also compares it with the elements in the cluster. In Cycle 3, ① enters the WRITE stage at Level 1, writing the updated cluster back to Level 1 based on CMP results.

For consecutive operations, different stages of different operations overlap. For instance, at Level 1, the CMP stage of ① and the READ stage of ② are performed in the same cycle (Cycle 2), and the WRITE stage of ①, the CMP stage of ②, and the READ stage of ③ are performed in the same cycle (Cycle 3). Each level can handle up to three parallel operations at different stages. Since no two operations can be in the same stage at the same level, each level only requires one hardware module for each stage, eliminating the concern of resource contention.

When a signal for an operation arrives at a level, it continues to propagate downward until it reaches the bottom level. For example, if ① arrives at Level 1 in Cycle 1, it will propagate to Level 2 in Cycle 2 and Level 3 in Cycle 3, and so on. Although in Cycle 3, ① is not yet aware of the comparison results of the CMP stage at Level 2, the heap structure guarantees that the node operated on at Level 3 must be one of the two child nodes of the node operated on at Level 2. Therefore, while the CMP stage is being performed at Level 2, the READ stage can be initiated at Level 3 simultaneously, reading the two child nodes of the current node.

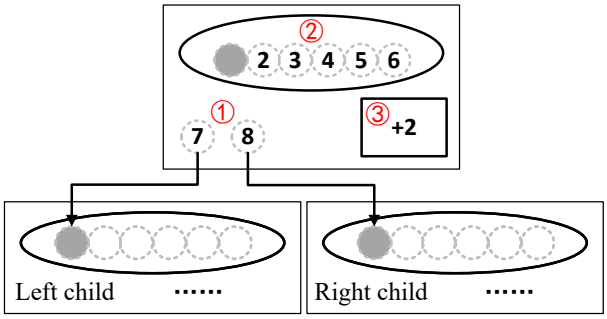
An operation may complete its task at an intermediate level. For example, when ClubHeap is empty, if ① is a *push* primitive, it only needs to insert the element it carries into the root node at Level 1. After this, the operation becomes a no-op (equivalent to pushing a $+\infty$), continuing to propagate downward and perform READ, CMP, and WRITE stages at each level without any real effect.

4.2 Node Data Structure

As explained in Sec. 4.1, because the CMP stage on the previous level has not yet produced the result when the operation enters the READ stage at the current level, both child nodes of the node being processed at the previous level need to be read simultaneously. This implies that the sibling nodes are better to be stored together to enable a single memory access to acquire both nodes. In addition, as shown in Fig. 6, the storage of the clusters adheres to three characteristics.

(1) The minimum ranked element in a cluster for a non-root node is stored in its parent node.

To illustrate the necessity, we assume that all the elements of a cluster are only stored on its corresponding node. According to Corollary 1 in Sec. 3.1, when a *pop* operation is performed on a node x, if x is full and its two child nodes, y and z, are not empty, we must choose the smaller value between $\min(E(y))$ and $\min(E(z))$ and promote the corresponding element to x to fill the vacancy left by the *pop* operation.



① The minimum ranked element in left and right child
② Elements in the cluster (except the minimum ranked)
③ The difference field

Figure 6: The node data structure in ClubHeap.

Assume x is at Level 1, and y and z are at Level 2. Since both elements with $\min(E(y))$ and $\min(E(z))$ are at Level 2, their comparison results can only be obtained after the CMP stage of Level 2 is completed. However, at the moment, the WRITE stage of Level 1 needs to determine the new $E(x)$ based on the comparison result and write it to memory. The data dependency prohibits the pipeline described in Sec. 4.1 from working properly. To fix the problem, it is necessary to store on node x the two elements with the ranks of $\min E(y)$ and $\min E(z)$ in y and z's clusters, so their comparison can be done during the CMP stage of Level 1.

(2) The elements in the cluster at each node are rank ordered.

If the cluster elements are unordered, comparators are needed to determine the minimum ranked element, which is resource consuming and requires complex combinational logic that can affect the achievable clock frequency. With the ordered elements in a cluster, the CMP stage only needs to compare the carried element of the operation with the first element in the cluster to determine the new minimum ranked element.

(3) At each non-leaf node, a field is used to indicate the difference in the number of elements between the two subtrees.

The `difference` field is used by the *push* operation in ClubHeap to achieve insertion-balance of the element distribution, which ensures free location can be found for the new elements before the heap memory is exhausted. Incoming elements are always inserted into the subtree with fewer elements. In case of a tie, the left subtree is chosen. Some existing heap-based implementations [8, 62] use two `counters` to record the number of elements in both subtrees. However, a field only recording the difference is sufficient to achieve the same effect by saving $\log N - 1$ bits per node.

4.3 ClubHeap Processor

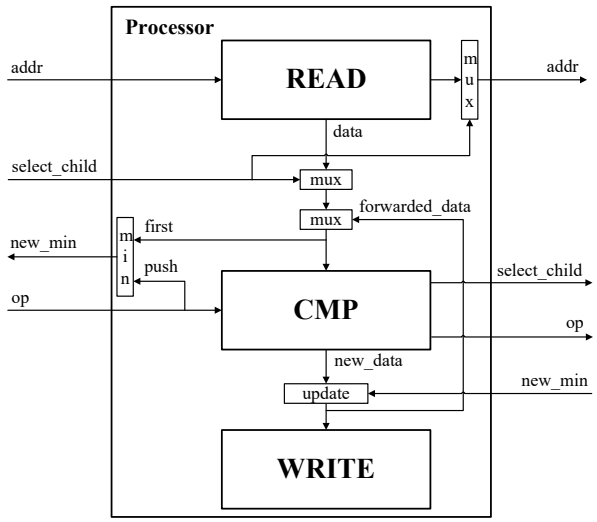


Figure 7: The structure of a processor in ClubHeap.

Each level of ClubHeap has an operation processor and the associated memory. As described in Sec. 4.1, a processor may simultaneously handle the WRITE stage of ①, the CMP stage of ②, and the READ stage of ③. Therefore, a processor in ClubHeap can be divided into three modules with each handling the task for one stage, as shown in Fig. 7. The arrows in the vertical direction represent the data path between the three modules in a single processor. A processor also exchanges data with the upper-level processor and the lower-level processor via the signals in the horizontal direction.

Overall, the READ module reads out a list of rank ordered elements, the CMP module performs parallel comparisons to find the position for writing (*push* or *replace*) or simply removes the first element (*pop*), and the WRITE module writes the result back. The two cycles for the CMP and WRITE stages are similar to the two cycles required for writing into a shift register [51]. Next, we describe the operation of the three modules in detail.

The READ module has only one input signal from the previous level: `addr`, which represents the address line for the nodes to read. As mentioned in Sec. 4.2, sibling nodes are stored together, so the READ module can read a pair of sibling nodes with a single address, and output the data via the `data` signal. Each node contains the data as shown in Fig. 6. Simultaneously, the READ module also obtains the addresses of the respective child nodes of this pair of sibling nodes. The method of storing these addresses is discussed in Sec. 4.4.

The CMP module is the core for executing *push*, *pop*, and *replace* primitives. Firstly, the data from sibling nodes acquired by the READ module passes through a data selector, which selects a branch based on the `select_child` signal generated during the CMP stage of the previous level. Assume the selected node is x. The address of x's child nodes is passed to the READ module of the next level via the `addr` signal, while x's data is sent to the CMP module for comparison. The CMP module determines the current operation type, *push*, *pop*, or *replace*, based on the `op` input signal. It outputs processed node data on the `new_data` signal for writing, and generates a new `op` signal to guide the CMP module at the next level to execute the operation. The `new_data` signal is forwarded as `forwarded_data` when processing the same node in consecutive cycles.

For the *push* operation, we assume the element to be pushed is e. If x is not full, e is directly inserted into the cluster as `new_data` and a new `op` signal, $push(+\infty)$, is generated, which is essentially a no-op. If x is full, e is inserted into the cluster if $rank(e) < \max(E(x))$. In this case, the original element with the $\max(E(x))$ rank value becomes the new e to be pushed to the next level. Otherwise, if $rank(e) \geq \max(E(x))$, the cluster on x is unchanged, and e is to be pushed to the next level. The `difference` field determines the child node for the next level insertion. If $rank(e)$ is less than the minimum rank value of the selected child node (note that the corresponding element e' is stored in node x), e replaces e', and e' is passed to the next level via the output `op` signal; otherwise, the *push* operation with the element e is passed to the next level.

For the *pop* operation, when the CMP module receives the operation from the signal `op`, because the element with $\min(E(x))$ is stored at the previous level, the *pop* operation on the cluster is actually done a cycle earlier. Therefore, the CMP module only needs to select a new minimum to fill the vacancy caused by popping the minimum ranked element, and pass it through `new_min`. Since the elements in a cluster are stored in order, the minimum ranked element can be determined without comparison, i.e., it must be the first element stored at x, or the second smallest ranked element in the cluster of x. Besides, the CMP module compares $\min(E(y))$ and $\min(E(z))$ stored at x, selects the smaller ranked element to be appended to the cluster of x, and passes the *pop* operation via `op` to the corresponding child node at the next level.

For the *replace* operation, since the minimum ranked element is stored at the previous level, the *pop* operation on x is actually performed a cycle earlier in the CMP stage at the previous level, and in the current stage, only two actions are needed: (1) select the new minimum ranked element to fill the vacancy via `new_min`; (2) push a new element e to the cluster, and e can also be selected as `new_min` if its rank value is smaller than that of any other element stored in the node. If $rank(e) < \min(\min(E(y)), \min(E(z)))$, e is directly inserted into the cluster of x, and the new minimum ranked element of x is popped out and returned to the upper level via the `new_min` signal. Otherwise, if $rank(e) \geq \min(\min(E(y)), \min(E(z)))$, the smaller ranked element with the rank value $\min(E(y))$ or $\min(E(z))$ fills the vacancy in the cluster of x, and the $replace(e)$ operation is passed via the `op` signal to the corresponding child node at the next level.

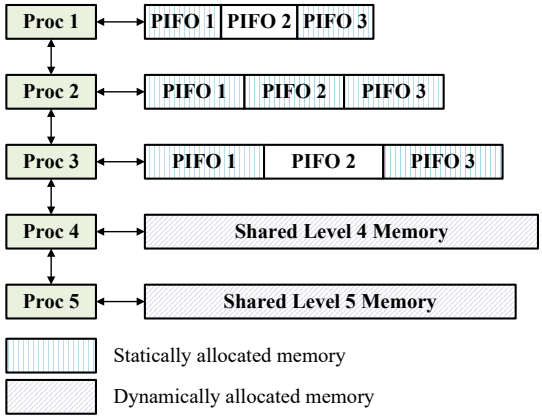


Figure 8: The ClubHeap pipeline structure.

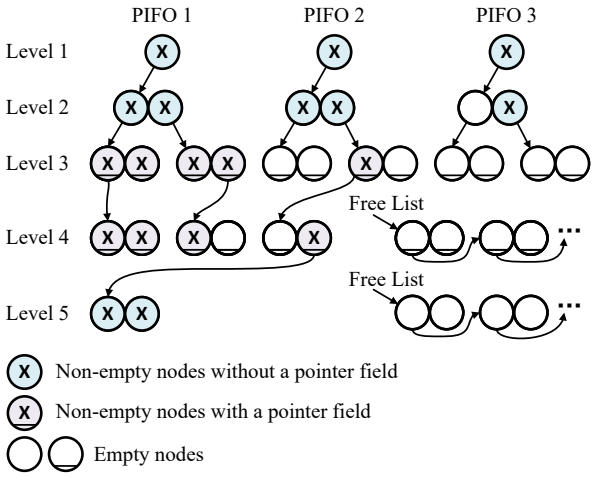


Figure 9: The logical structure of a ClubHeap partitioned into three logical PIFOs.

Finally, the `new_data` output signal from the CMP module is updated by the `new_min` output signal from the data selector in front of the CMP module at the next level, which represents the new minimum ranked element selected by the next level and should be stored at the current level as described in Sec. 4.2. The updated node data is sent to the WRITE module and written to the memory.

4.4 ClubHeap Memory Organization

ClubHeap maintains the memory alongside a processor at each level. The memory is used to support the logical PIFO queues. Since the minimum ranked element of each cluster is actually stored at the upper level, ClubHeap guarantees that the data required in a CMP stage is available at the same level, and thus data dependencies between levels are alleviated. Each processor only accesses the memory at the same level. As shown in Fig. 8, a processor only reads data from memory during the READ stage and writes it back during the WRITE stage, so a dual-port SRAM is sufficient to meet the needs.

Furthermore, as shown in Fig. 8, ClubHeap supports two different forms of memory allocation methods. The static allocation method, similar to P-Heap [8] and BMW-Tree [62], allows for direct child-node address calculation from the current node's address, without the need for storing the child-node's address in memory. Thus, one pointer per node can be saved. However, the drawback is that such allocation implies a complete binary tree, meaning the L-th level of each logical queue requires storage space for 2^{L-1} nodes. With M logical queues provisioned, the memory size required at deeper levels can be prohibitively large.

The discussion in Sec. 3.1 illustrates that the actual storage space required by ClubHeap at the L-th level, S_L, may be significantly less than $\mathbf{B2} = M \times 2^{L-1}$. It is worthwhile to trade a pointer field by reducing **B2** to **B1**. Therefore, as shown in Fig. 8, we employ static memory allocation in shallow levels where **B2**<**B1** and dynamic memory allocation in deep levels where **B1**<**B2** to achieve an optimal space management strategy for different values of M.

As shown in Fig. 9, to achieve dynamic memory allocation for the L-th level, we need to enable the pointer fields of this level and the upper level. During initialization, the pointers at the $(L-1)$-th level are set to *null*, and the pointers at the L-th level are linked into a unidirectional linked list called *free list*. For each pair of sibling nodes in the free list, the left child's pointer field points to the next node, while the right child's pointer is null. If a node is not in the free list, its pointer field either points to its child nodes at the next level if it has non-empty children, or is set to null otherwise. Leaf nodes at the bottom level do not need the pointer field. The `head` address of the free list is maintained by the READ module.

We take dynamic memory allocation at $L = 4$ as an example. The pointer field read from Level 3 is passed as `addr` to Level 4. If `addr` is null (the initial value), the READ module replaces `addr` with `head`, updates `head` with the pointer field of the read node, and resets this pointer field to null. If, at the beginning of the CMP stage at Level 4, it is found that the current node is about to become empty[5] and its sibling node is already empty, a signal is sent to Level 3 to reset the corresponding pointer field to null. Subsequently, during the WRITE stage at Level 4, the pointer field of this empty node is updated to `head`, and the new `head` becomes the address of this empty node. In short, the free list is a stack composed of free nodes, which pops nodes when allocating space and pushes nodes when recycling space. If allocating and recycling occur in the same cycle, the recycled space can be immediately allocated to the new pair of nodes.

[5]The criterion for determining that a node is about to become empty is (1) *push*(+∞) on an empty node, or (2) *pop* on a node with no more than one element. This circuit is only concerned with the valid bits of the elements. The signal to update the pointer field is passed along with `new_min` to the WRITE module of the upper level.

5 Implementation

We implement ClubHeap using 919 lines of Chisel [14] code, and evaluate it on a Xilinx Alveo U280 Data Center Accelerator Card [60] with an FPGA containing 1,303k LUTs, 2,607k Flip-flops (FF), 2,016 Block RAMs (BRAM), and 960 UltraRAMs. The implementation is parameterized, allowing users to specify the parameter K, the maximum number of elements (N), priority levels (P) and logical PIFOs that a ClubHeap can be partitioned into (M). We also implement the existing state-of-the-art PIFO queues BMW-Tree [62] and BBQ [4] on the same FPGA for comparison.

We also analyze the ASIC implementations for some typical specifications with Design Compiler [53] on an open-source 45nm toolchain [17, 23, 40, 55] at a frequency of 800MHz[6]. The reported area is listed in Table 2.

Design	K	N	P	M	Area (mm^2)
BBQ	N/A	2^{17}	2^{16}	1	27.23
ClubHeap	2	2^{17}	2^{16}	1	4.83
ClubHeap	16	2^{17}	2^{16}	1	6.15
ClubHeap	2	2^{17}	2^{32}	2^8	61.00
ClubHeap	16	2^{17}	2^{32}	2^8	63.60

Table 2: ASIC area comparison.

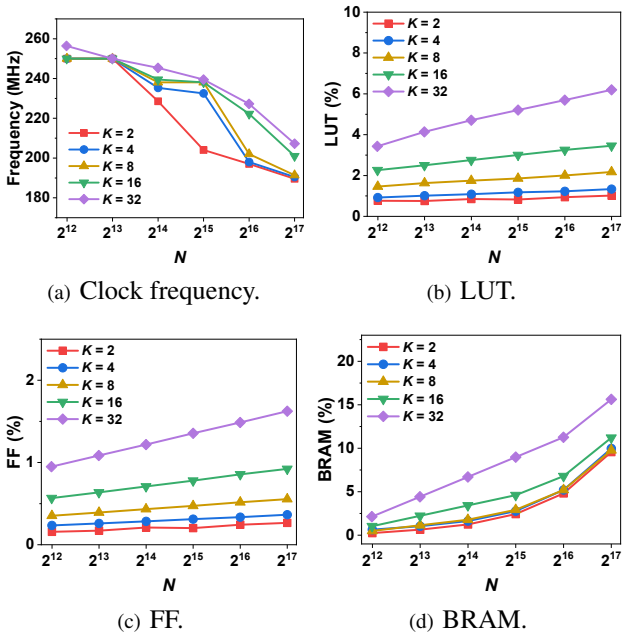
Table 2 indicates ClubHeap consumes only 17.7% (K=2) or 22.6% (K=16) of area used by BBQ for the same specification[7]. BMW-Tree is not listed here because the open-source 4-way implementation can not support N=2^{17}. We conduct a more detailed comparison between ClubHeap, BMW-Tree, and BBQ on FPGA in Sec. 6.

6 Evaluation

We evaluate ClubHeap through experiments and analysis to show that ClubHeap can meet the demands mentioned in Sec. 2.1, and is the most performant PQ implementation for a PIFO scheduler so far. We first introduce the experimental methodology in Sec. 6.1, then discuss the selection of parameter K in Sec. 6.2, and compare ClubHeap with the state-of-the-art implementations, BMW-Tree and BBQ, in Sec. 6.3. In Sec. 6.4, we show that a clustered multi-way tree is inferior in terms of throughput and resource consumption, which explains why we choose a binary heap rather than a multi-way one.

[6]BBQ claims their *logic* can achieve 3.1GHz in a 7nm process [4, 59], but the bottleneck in large SRAM access is ignored.

[7]Since a ClubHeap with L levels can contain a maximum of $N = K(2^L - 1) = 2^{K+L} - K$ elements, there may be slight variations in N for different values of K. For example, when K=2 and L=15, N=65,534, while for K=4 and L=14, N=65,532. Since these minor differences have almost negligible practical effects, both cases are denoted as $N = 2^{16}$ in our evaluation and considered as the same data point.



(a) Clock frequency. (b) LUT. (c) FF. (d) BRAM.

Figure 10: Frequency and resource consumption of ClubHeap with different K.

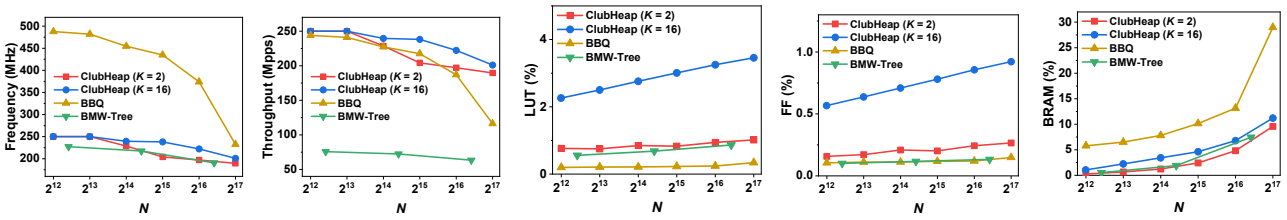
6.1 Methodology

We evaluate the FPGA implementations of ClubHeap, BMW-Tree, and BBQ for comparison. Clock frequency and resource consumption are reported by Xilinx Vivado, which may differ from existing papers [4, 62] due to different FPGAs. We measure the achievable clock frequency by testing different cycles (step size: 0.025ns) and selecting the minimum allowed. The throughput is obtained via FPGA simulation, as it is independent of traffic characteristics or scheduling algorithms for these structures. For resource consumption, we focus on LUTs, FFs, and BRAMs; UltraRAMs are converted to BRAMs with equivalent storage capacity in results.

Unless otherwise specified, we assume each scheduler element includes a 16-bit rank value (P=2^{16}) and 32-bit metadata (for flow ID, packet ID, etc.), sufficient for most PIFO-based scheduling algorithms [51, 62]. For a fair comparison, we focus on scenarios with one logical PIFO (M=1), as ClubHeap better supports logical partitioning than BMW-Tree and BBQ.

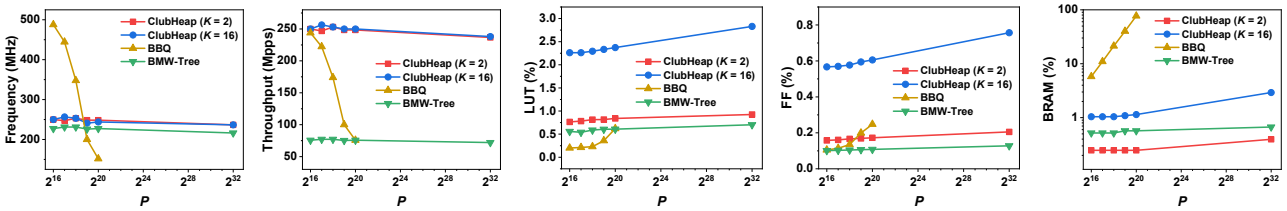
6.2 Effects of Different K Values

We first evaluate the clock frequency and resource consumption of ClubHeap with different K values. As shown in Fig. 10(a), when N=2^{12} and N=2^{13}, clock frequencies are approximately 250MHz regardless of K. As N increases, the clock frequency decreases, and differences in clock frequency begin to emerge for different K. Experiment results indicate that for the same N value, a larger K generally corresponds to a higher clock frequency. At N=2^{17}, the frequency for K=32 is



(a) Clock frequency. (b) Throughput. (c) LUT. (d) FF. (e) BRAM.

Figure 11: Frequency, throughput and resource consumption for ClubHeap, BBQ and BMW-Tree at different N.



(a) Clock frequency. (b) Throughput. (c) LUT. (d) FF. (e) BRAM.

Figure 12: Frequency, throughput, and resource consumption for ClubHeap, BBQ, and BMW-Tree at different P.

207.25MHz, which is approximately 10% higher than that for K=2 (189.57MHz). This is because a larger K results in fewer levels in ClubHeap, reducing the place and route complexity.

However, a larger K requires more resources on the FPGA, as shown in Fig. 10(b), 10(c), and 10(d). At N=2^{17}, a ClubHeap with K=32 requires relatively 6.2% LUTs and 1.6% FFs, which are approximately 6x those for K=2. Additionally, the growth rate of LUT and FF overhead with increasing N is also higher for a larger K. Comparing N=2^{12} and N=2^{17}, for a 32x increase in N, ClubHeap with K=2 experiences a 33% increase in LUTs and a 68% increase in FFs, while ClubHeap with K=32 experiences an 80% increase in LUTs and a 71% increase in FFs. Overall, ClubHeap's demand for LUTs and FFs is approximately $O(\log N)$, ensuring its good scalability.

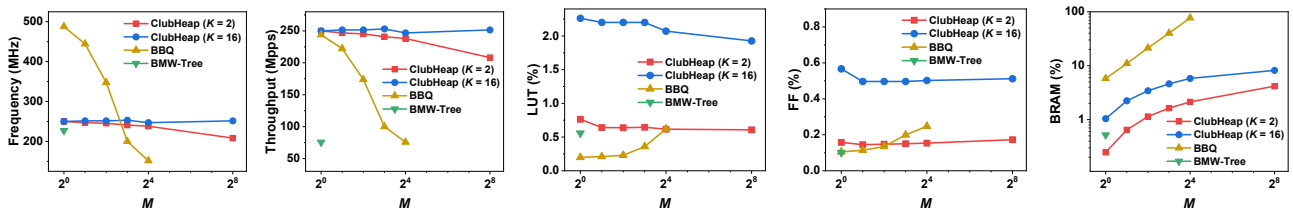
For BRAM, although the same number of elements requires the same space, a larger K implies more wasted space and thus requires more BRAMs. Since sibling nodes are stored at the same address, there are only 2^{L-2} addresses for the 2^{L-1} nodes on the L-th level. When L is small, only a small portion of a BRAM block is utilized, leaving the unused space wasted. At N=2^{12}, the number of BRAM blocks required by ClubHeap with K=32 is 8.6x that for K=2. At deeper levels, a BRAM block can be fully utilized, and the number of BRAM blocks is mainly determined by the number of storage bits. A smaller K indicates each node stores fewer elements, leading to more nodes and a greater demand for the difference field and the pointer for dynamic memory allocation. Therefore for a larger N, a smaller K results in more space requirements, reducing the relative savings compared to a larger K. At N=2^{17}, the BRAM consumption for K=32 is only 1.63x that for K=2.

In summary, the analysis indicates that for the same N, ClubHeap with a larger K exhibits a higher clock frequency but also demands more resources. When applying ClubHeap in practice, K can be selected based on the specific requirements as a tradeoff between performance and resource consumption. A proper K value could be 2, 4, 8, or 16.

6.3 Comparison with BMW-Tree and BBQ

We compare ClubHeap with K=2 and K=16 to RPU-BMW, which is the scalable version of BMW-Tree [62], and BBQ [4]. The comparison focuses on three parameters: the number of elements (N), the number of priority levels (P) and the number of logical PIFOs (M).

Evaluation on N. In Fig. 11 we fix P=2^{16} and M=1 to evaluate the scalability of the implementations on N. We observe that for all three PQ implementations, the critical path affecting clock frequency is the combinational logic when N is small, whereas it shifts to SRAM access when N is large, because they all access SRAM in one clock cycle as part of the pipeline design. ClubHeap and BMW-Tree have similar comparator-dominated combinational logic latency; thus, they have similar frequencies. ClubHeap achieves a throughput 3x higher than BMW-Tree with CPR= 3. On the other hand, BBQ's HFFS uses the least combinational logic and has the highest frequency for small N, but the frequency drops rapidly as N increases due to the memory access constraints. As shown in Fig. 11(a), when N=2^{17}, BBQ's clock frequency is comparable to ClubHeap and BMW-Tree, making CPR a more decisive factor for throughput. Fig. 11(b) indicates that ClubHeap's throughput is approximately equal to BBQ for small N, but 63% (K=2) or 72% (K=16) higher when N=2^{17},



(a) Clock frequency. (b) Throughput. (c) LUT. (d) FF. (e) BRAM.

Figure 13: Frequency, throughput, and resource consumption for ClubHeap, BBQ, and BMW-Tree at different M.

showing better scalability for N than BBQ.

As shown in Fig. 11(c) and 11(d), ClubHeap processors at each level require more LUT and FF resources than BMW-Tree and BBQ. As shown in Fig. 11(e), ClubHeap consumes the similar BRAMs as BMW-Tree, but much fewer than BBQ. At N=2^{17}, ClubHeap consumes 33% (K=2) or 39% (K=16) fewer BRAMs than BBQ. Note that the proportion of LUTs and FFs used by these structures to the total FPGA resources is significantly less than that of BRAMs. Therefore, it is worthwhile for ClubHeap to consume more LUTs and FFs to save BRAMs compared to BBQ.

The 45nm ASIC analysis in Sec. 5 supports the same conclusion. For N=2^{17}, P=2^{16}, and M=1, the clock frequency bottleneck for both ClubHeap and BBQ is SRAM access. BBQ requires larger SRAM which causes higher access latency, while ClubHeap's memory is distributed across levels which has lower access latency. Additionally, ClubHeap consumes less chip area than BBQ. The following evaluations indicate that ClubHeap's advantages become more pronounced as P and M increase. We believe that ClubHeap will remain a better choice at more advanced process nodes.

Evaluation on P. We evaluate the implementations for different P with N=2^{12} and M=1 in Fig. 12 (for BMW-Tree, we select the nearest N=5,460). When the range of ranks is 2x wider, the only changes needed in ClubHeap are an additional bit for the rank field and an additional 1-bit width of the comparators.When P increases from 2^{16} to 2^{20}, the clock frequency of ClubHeap only drops less than 3%. As a heap-based PQ implementation, BMW-Tree exhibits similar scalability for P. However, BBQ needs to expand the number of buckets to support a larger range of P, resulting in poorer scalability. As shown in Fig. 12(a), the large number of buckets poses challenges in routing, causing its clock frequency to drop lower than that of ClubHeap when P=2^{19}. Considering CPR, ClubHeap achieves a throughput 3.28x higher than BBQ when P=2^{20} as shown in Fig. 12(b). BBQ with a larger P fails to be synthesized on our FPGA because of the exhausted BRAMs, while ClubHeap can scale up to P=2^{32} with a cost of only 5.5% frequency reduction.

As shown in Fig. 12(c), 12(d), and 12(e), with the increase of P, BBQ requires more levels or a larger bitmap width for HFFS, resulting in higher consumption of LUTs, FFs, and BRAMs following an $O(M)$ trend. Among them, BRAM consumption grows the fastest, reaching 70x more than ClubHeap (K=2) when P=2^{20}. In contrast, the LUT and FF consumption for ClubHeap grows slowly. When P increases from 2^{16} to 2^{20}, ClubHeap consumes only about 10% additional LUTs and FFs, and the BRAM consumption remains constant when the additional bits do not cause the node size to exceed the width provided by BRAMs at P=2^{16}.

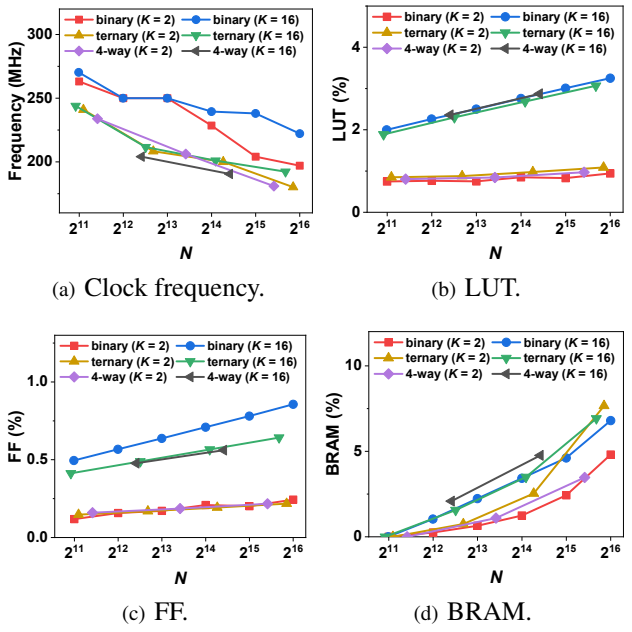
Evaluation on M. Support for logical partitioning is an advantage of ClubHeap over BMW-Tree, so BMW-Tree is represented by only one point (M=1) in Fig. 13. BBQ supports logical partitioning, but it requires evenly dividing buckets among various logical PIFOs. With a constant N and P, M logical PIFOs in BBQ use M times more buckets, which is equivalent to a M times larger P. Fig. 13 shows the frequency, throughput, and resource consumption when a physical queue with N=2^{12} and P=2^{16} is divided into M logical PIFOs. We draw a similar conclusion as for Fig. 12. The frequency of ClubHeap with K=2 decreases by only 16.7% when M scales to 2^8, while at K=16 there is no frequency reduction.

As shown in Fig. 13(c) and 13(d), the LUT and FF consumption decreases when M gets larger, because more memory is synthesized as BRAMs rather than LUTMEMs. It is worth mentioning that the memory space required for ClubHeap has an upper limit independent of M, bounded by **B1**.

In summary, although BBQ achieves a high throughput for small N, it lacks scalability on P and M. BMW-Tree, on the other hand, offers scalability on both N and P, but has lower throughput and does not support logical partitioning.

6.4 Binary Heap vs Multi-way Heap

We compare ClubHeap with clustered ternary and 4-way heaps to explain the rationale for choosing a binary heap rather than a multi-way one. As shown in Fig. 14(a), the frequency of a clustered multi-way heap is always lower than that of ClubHeap. The frequency degradation is because the clustered ternary heap requires more comparisons and selections during the process of selecting `new_min`, resulting in a longer critical path. Meanwhile, as shown in Fig. 14(b), 14(c), and 14(d), the clustered multi-way heap does not reduce the resource consumption as well.



(a) Clock frequency. (b) LUT.

(c) FF. (d) BRAM.

Figure 14: Frequency and resource consumption comparison of ClubHeap and a clustered ternary / 4-way heap.

7 Related Works

PIFO [51] is a classical programmable packet scheduling abstraction that can express a wide range of algorithms, but it faces two limitations. (1) PIFO's programmability is achieved through a mesh composed of multiple PQs, but its original implementation , SR-PIFO [51], encounters significant challenges in terms of throughput and scalability. (2) PIFO assumes that the rank of a packet is already calculated when it enters the scheduler and is unchanged thereafter. The assumption makes the abstraction unable to express algorithms with dynamic ranks, e.g. pFabric [3], which introduces a starvation avoidance mechanism to SRPT [46].

Optimization of PIFO. Replacing SR-PIFO with other PQs can help improve the throughput and scalability of PIFO. OPQ and APQ [5] introduce the Single-Instruction-Multiple-Data (SIMD) technique on systolic arrays to enhance throughput, but they lack scalability on the number of elements. Calendar Queues [11], CPI [58], and BBQ [4] implement PQs using buckets, but the number of buckets limits the number of priorities. P-Heap [8], Pipelined Heap [28], H-PQ [26], and BMW-Tree [62, 68] distribute the storage and computation of a PQ across levels using heaps, achieving good scalability. However, the inter-operational data dependency problem discussed in Sec. 2.3 restricts their performance. ClubHeap, proposed in this paper, is the first high-speed and scalable PIFO queue implementation, which can help PIFO overcome the first limitation.

Approximation of PIFO. Another direction to overcome the first limitation is to approximate PIFO by eliminating the need for using a mesh of PQs. UPS [36] employs LSTF as an approximate universal packet scheduling algorithm, enabling programmable packet scheduling with a single PQ. SP-PIFO [1] utilizes a group of FIFOs, clustering flows with similar ranks into the same FIFO to achieve scheduling approximation. FDPA [12], AFQ [47], PCQ [48], PUPD [56], GearBox [20], QCluster [61], and PR-AQM [32] also use multiple FIFOs for approximate scheduling. AIFO [64], on the other hand, only uses a single FIFO, which limits its ability to approximate scheduling order but ensures approximation for whether a packet is admitted or dropped. FAIFO [69] extends AIFO by considering information freshness to better support time-sensitive applications. RIFO [39] improves AIFO's admission control using min-max linear normalization. PACKS [2] is a hybrid structure of SP-PIFO and AIFO, which achieves approximation of PIFO in terms of both scheduling and admission.

Extension of PIFO. New abstractions are proposed to overcome the second limitation. PIEO [49] allows the PQs to pop elements from any location rather than the head only, enabling scheduling with eligibility predicates. PIPO [66] implements an approximated PIEO with multiple PIFO queues for bounded-delay scheduling algorithms. PCSQ [27] extends PIPO to large-scale deterministic networks with long-distance links. CIPO [24] proposes an approximation of PIEO with two-dimensional predicates. DR-PIFO [16] combines a PIFO queue with a group of FIFOs to support dynamic ranks. Eiffel [44] introduces a queue structure based on Circular FFS to provide on-dequeue scheduling. ClubHeap does not support these extended abstractions, but it can be incorporated as a PIFO queue in their implementations.

8 Conclusion

ClubHeap is a novel heap-based PIFO queue implementation. It overcomes the performance and scalability challenges of PIFO's implementations to achieve line-rate scheduling for a large number of flows with a wide range of priority levels. It is beneficial to any programmable schedulers implemented with the PIFO abstraction on switches or SmartNICs based on either FPGA or ASIC.

Acknowledgement. We thank the anonymous reviewers and the paper shepherd Stefan Schmid for their insightful comments and suggestions, which help improve this paper. We thank Chuwen Zhang (Tsinghua University), Yongzheng Zhang (ShanghaiTech University), Yinuo Jia (Wuhan University), and Haodong Wang (Northwestern Polytechnical University) for helping in conducting experiments. This work is supported by National Key Research and Development Program of China (2024YFE0203900), NSFC (62032013, 62272258, 62172108), NSFC-RGC (62061160489). The corresponding author is Bin Liu (lmyujie@gmail.com).

References

[1] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, Santa Clara, CA, 2020. USENIX Association.

[2] Albert Gran Alcoz, Balazs Vass, Pooria Namyar, Behnaz Arzani, Gabor Retvari, and Laurent Vanbever. Everything matters in programmable packet scheduling. In *NSDI*. USENIX, 2025.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, 2013.

[4] Nirav Atre, Hugo Sadok, and Justine Sherry. BBQ: A fast and scalable integer priority queue for hardware packet scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 455–475, Santa Clara, CA, 2024. USENIX Association.

[5] Imad Benacer, François-Raymond Boyer, and Yvon Savaria. A fast, single-instruction–multiple-data, scalable priority queue. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(10):1939–1952, 2018.

[6] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *SIGCOMM Comput. Commun. Rev.*, 26(4):143–156, 1996.

[7] Jon C. R. Bennett and Hui Zhang. WF2Q: worst-case fair weighted fair queueing. In *Proceedings of the Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies Conference on The Conference on Computer Communications (INFOCOM) - Volume 1*, INFOCOM'96, page 120–128, USA, 1996. IEEE Computer Society.

[8] Ranjita Bhagwan and Bill Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 538–547, 2000.

[9] Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha. Shared hardware data structures for hard real-time systems. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, page 133–142, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, 2013.

[11] Randy Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[12] Carmelo Cascone, Nicola Bonelli, Luca Bianchi, Antonio Capone, and Brunilde Sansò. Towards approximate fair bandwidth sharing via dynamic priority queuing. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, 2017.

[13] Ravikesh Chandra and Oliver Sinnen. Improving application performance with hardware data structures. In *2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–4, 2010.

[14] ChipsAlliance. Chisel. https://www.chisel-lang.org/, 2024.

[15] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, 1989.

[16] Mostafa Elbediwy, Bill Pontikakis, Alireza Ghaffari, Jean-Pierre David, and Yvon Savaria. DR-PIFO: A dynamic ranking packet scheduler using a push-in-first-out queue. *IEEE Transactions on Network and Service Management*, 21(1):355–371, 2024.

[17] Hewlett Packard Enterprise. CACTI. https://github.com/HewlettPackard/cacti, 2017.

[18] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, 2017. USENIX Association.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg.

Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[20] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. Gearbox: A hierarchical packet scheduler for approximate weighted fair queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 551–565, Renton, WA, 2022. USENIX Association.

[21] S. Jamaloddin Golestani. A stop-and-go queueing framework for congestion management. *SIGCOMM Comput. Commun. Rev.*, 20(4):8–18, 1990.

[22] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997.

[23] Bespoke Silicon Group. BSG Block-box SRAM Generator. https://github.com/bespoke-silicon-group/bsg_fakeram, 2022.

[24] Feng Guo, Shidong Sun, Junjie Hu, Ning Zhang, and Zhiqiang Lv. CIPO: Efficient, lightweight and programmable packet scheduling. *Computer Networks*, 245(C), 2024.

[25] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-Queue fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, Renton, WA, 2019. USENIX Association.

[26] Muhuan Huang, Kevin Lim, and Jason Cong. A scalable, high-performance customized priority queue. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2014.

[27] Yudong Huang, Shuo Wang, Shiyin Zhu, Guoyu Peng, Xinyuan Zhang, Tian Pan, Tao Huang, Lei Zhou, Zuopin Cheng, Daorong Guo, Hui Lin, Lianqing Zhang, Juyan Lei, Liangzhang Xu, Wei Wang, Xinmin Liu, Xuejun You, and Yunjie Liu. Poster: Programmable cycle-specified queue for deterministic networking. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 1132–1134, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Aggelos Ioannou and Manolis G. H. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking*, 15(2):450–461, 2007.

[29] C.R. Kalmanek, H. Kanakia, and S. Keshav. Rate controlled servers for very high-speed networks. In *[Proceedings] GLOBECOM '90: IEEE Global Telecommunications Conference and Exhibition*, pages 12–20 vol.1, 1990.

[30] Pierre Lavoie, David Haccoun, and Yvon Savaria. A systolic architecture for fast stack sequential decoders. *IEEE Transactions on Communications*, 42(234):324–335, 1994.

[31] Joseph Y. T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1–4):209–219, 1989.

[32] Ziyong Li, Yuxiang Hu, Le Tian, and Zhao Lv. Packet rank-aware active queue management for programmable flow scheduling. *Computer Networks*, 225:109632, 2023.

[33] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, 2020.

[34] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[35] Paul E. McKenney. Stochastic fairness queueing. In *Proceedings. IEEE INFOCOM '90: Ninth Annual Joint Conference of the IEEE Computer and Communications*, pages 733–740 vol.2, 1990.

[36] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 501–521, Santa Clara, CA, 2016. USENIX Association.

[37] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. Formal abstractions for packet scheduling. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[38] Sung-Whan Moon, Kang G. Shin, and Jennifer Rexford. Scalable hardware priority queue architectures for high-speed packet switches. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium (RTTAS)*, pages 203–212, 1997.

[39] Habib Mostafaei, Maciej Pacut, and Stefan Schmid. RIFO: Pushing the efficiency of programmable packet schedulers, 2024.

[40] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14, 2007.

[41] NVIDIA. NVIDIA Spectrum SN5000 Series Switches. https://www.nvidia.com/en-us/networking/ethernet-switching/, 2023.

[42] Dan Picker and Ronald D. Fellman. A VLSI priority packet queue with inheritance and overwrite. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(2):245–253, 1995.

[43] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, SIGCOMM '17, page 404–417, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 17–32, Boston, MA, 2019. USENIX Association.

[45] Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. SCED: a generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM Transactions on Networking*, 7(5):669–684, 1999.

[46] Linus E. Schrage and Louis W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[47] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 1–16, USA, 2018. USENIX Association.

[48] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 685–700, USA, 2020. USENIX Association.

[49] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 367–379, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery.

[51] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[52] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, Boston, MA, 2019. USENIX Association.

[53] Synopsys. Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, 2023.

[54] Kenji Toda, Kenji Nishida, Eiichi Takahashi, Nick Michell, and Yoshinori Yamaguchi. Implementation of a priority forwarding router chip for real-time interconnection networks. In *Second Workshop on Parallel and Distributed Real-Time Systems*, pages 166–175, 1994.

[55] NC State University. NCSU FreePDK45. https://eda.ncsu.edu/freepdk/freepdk45/, 2023.

[56] Balázs Vass, Csaba Sarkadi, and Gábor Rétvári. Programmable packet scheduling with sp-pifo: Theory, algorithms and evaluation. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2022.

[57] D.C. Verma, H. Zhang, and D. Ferrari. Delay jitter control for real-time communication in a packet switching network. In *Proceedings of TRICOMM '91: IEEE Conference on Communications Software: Communications for Distributed Applications and Systems*, pages 35–43, 1991.

[58] Hao Wang and Bill Lin. Per-flow queue management with succinct priority indexing structures for high speed packet scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1380–1389, 2013.

[59] Shien-Yang Wu, C.Y. Lin, M.C. Chiang, J.J. Liaw, J.Y. Cheng, S.H. Yang, C.H. Tsai, P.N. Chen, T. Miyashita, C.H. Chang, V.S. Chang, K.H. Pan, J.H. Chen, Y.S. Mor, K.T. Lai, C.S. Liang, H.F. Chen, S.Y. Chang, C.J. Lin, C.H. Hsieh, R.F. Tsui, C.H. Yao, C.C. Chen, R. Chen, C.H. Lee, H.J. Lin, C.W. Chang, K.W. Chen, M.H. Tsai, K.S. Chen, Y. Ku, and S. M. Jang. A 7nm cmos platform technology featuring 4th generation finfet transistors with a 0.027um2 high density 6-t sram cell for mobile soc applications. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 2.6.1–2.6.4, 2016.

[60] Xilinx. Alveo U280 Data Center Accelerator. https://www.avnet.com/opasdata/d120001/medias/docus/196/XLX-A-U280-A32G-DEV-G-Datasheet.pdf, 2019.

[61] Tong Yang, Jizhou Li, Yikai Zhao, Kaicheng Yang, Hao Wang, Jie Jiang, Yinda Zhang, and Nicholas Zhang. Qcluster: Clustering packets for flow scheduling. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1752–1763, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. BMW tree: Large-scale, high-throughput and modular PIFO implementation using balanced multi-way sorting tree. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 208–219, New York, NY, USA, 2023. Association for Computing Machinery.

[63] Xin Yu, Wei Chen, and Ye Tian. OWFQ: Reducing packet drops for approximate weighted fair queueing with calendar queues. In *2023 9th International Conference on Computer and Communications (ICCC)*, pages 540–544, 2023.

[64] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 179–193, New York, NY, USA, 2021. Association for Computing Machinery.

[65] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty years after: Hierarchical Core-Stateless fair queueing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 29–45. USENIX Association, 2021.

[66] Chuwen Zhang, Zhikang Chen, Haoyu Song, Ruyi Yao, Yang Xu, Yi Wang, Ji Miao, and Bin Liu. PIPO: Efficient programmable scheduling for time sensitive networking. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11, 2021.

[67] Hui Zhang and Domenico Ferrari. Rate-controlled service disciplines. *J. High Speed Netw.*, 3(4):389–412, October 1994.

[68] Zhiyu Zhang, Shili Chen, Ruyi Yao, Ruoshi Sun, Hao Mei, Hao Wang, Zixuan Chen, Gaojian Fang, Yibo Fan, Wanxin Shi, Sen Liu, and Yang Xu. vpifo: Virtualized packet scheduler for programmable hierarchical scheduling in high-speed networks. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 983–999, New York, NY, USA, 2024. Association for Computing Machinery.

[69] Meng-yuan Zhu, Ke-fan Chen, Zhuo Chen, and Na Lv. FAIFO: UAV-assisted IoT programmable packet scheduling considering freshness. *Ad Hoc Networks*, 134(C), 2022.

Self-Clocked Round-Robin Packet Scheduling

Erfan Sharafzadeh[1,2], Raymond Matson[3], Jean Tourrilhes[2], Puneet Sharma[2] and Soudeh Ghorbani[1,4]

[1]*Johns Hopkins University,* [2]*Hewlett Packard Labs,* [3]*University of California Riverside,* [4]*Meta*

Abstract. Deficit Round Robin (DRR) is the de facto fair packet scheduler in the Internet due to its superior fairness and scalability. We show that DRR can perform poorly due to its assumptions about packet size distributions and traffic bursts. Concretely, DRR performs best if (1) packet size distributions are known in advance; its optimal performance depends on tuning a parameter based on the largest packet, and (2) all bursts are long and create backlogged queues. We show that neither of these assumptions holds in today's Internet: packet size distributions are varied and dynamic, complicating the tuning of DRR. Plus, Internet traffic consists of many short, latency-sensitive flows, creating small bursts. These flows can experience high latency under DRR as it serves a potentially large number of flows in a round-robin fashion.

To address these shortcomings while retaining the fairness and scalability of DRR, we introduce Self-Clocked Round-Robin Scheduling (SCRR), a parameter-less, low-latency, and scalable packet scheduler that boosts short latency-sensitive flows through careful adjustments to their virtual times without violating their fair share guarantees. We evaluate SCRR using theoretical models and a Linux implementation on a physical testbed. Our results demonstrate that while performing on an equal footing with DRR on achieving flow fairness, SCRR reduces the average CPU overhead by 23% compared to DRR with a small quantum while improving the application latency by 71% compared to DRR with a large quantum.

1 Introduction

The Internet is continuously evolving as a multi-party environment. To meet service performance requirements, enforcing isolation at various levels, from tenants to applications and individual flows is crucial [40, 43, 58, 72, 80, 83]. Packet scheduling offers fine-grained control over network bandwidth at both the edge and core of the network. Despite the abundance of scheduling techniques relying on explicit labeling, such as strict priority queues [28, 77, 89] or time wheels [72, 78], obtaining flow priorities or timestamps from Internet transports is usually not feasible. Instead, Fair Queuing provides fine-grained fairness without flow information, and has been deeply studied from fairness, latency bounds, and computational complexity standpoints [21, 32, 41, 42, 64, 68]. Since Fair Queuing schedulers suffer from non-linear computational complexity that hinders deployment [21, 41, 42], a middle-ground alternative is to use Deficit Round-Robin (DRR) scheduling due to its low overhead and scalability [56, 69, 81, 90]. Indeed, DRR is used everywhere from hardware traffic management pipelines to software middleboxes, host stacks, and network interface cards [6, 7, 12, 14, 15, 82, 83].

Regrettably, Internet traffic has evolved since Deficit Round-Robin was proposed, and its drawbacks are increasingly problematic. Workloads like real-time video conferencing and voice communication, streaming, cloud gaming, and instant messaging are now widespread among more traditional web and file transfer [27, 35, 52, 59, 60, 65, 67, 86, 87]. Such workloads feature diverse properties concerning flow counts, flow arrival rates, flow sizes, packet lengths, packet inter-arrivals, and burstiness [22, 23, 27, 39, 67, 85, 86]. First, concerning burstiness, Internet traffic traces show bursts of packets with diverse lengths and inter-arrivals [1, 4, 20, 22, 29]. Long bursts are prone to creating backlogged queues at the packet scheduler, while, broadly, short bursts from latency-sensitive applications such as web and streaming [61, 74] exhibit periodic on-off arrivals. For example, the median response size for web workloads does not exceed 10 kB, while Variable Bit-Rate (VBR) streaming workloads such as Zoom have a median frame length of 10 kB with large idle gaps between frames [61].

Our experiments with DRR reveal that under a backlogged scheduler, a bursty latency-sensitive flow has to wait for up to a full scheduling round, i.e., until all flows have been serviced at least once, before it can send its traffic. This can be detrimental to the performance of such flows, causing much higher application latency compared to Fair Queueing Schedulers. Even with recent DRR enhancements, such as Sparse Flow Optimization (SFO) [47], which prioritize new incoming flows over backlogged ones to improve responsive-

ness, bursts with more than one packet are prone to being quickly demoted to lower-priority and forced to compete with backlogged flows, thus experiencing long delays.

Secondly, packet lengths are diverse and skewed [1,4,16,20, 22,29] and packet length distribution for wide-area network traffic is difficult to generalize. Many modern applications use small packets to reduce latency and the impact of packet loss [30,79]. Network Interface hardware acceleration [11] may produce very large packets (64 kB). DRR relies on a fixed, user-specified quantum equal to (or larger than) the maximum packet size in the network [81], making it hard to tune. A large quantum can be detrimental to intermediate buffers as it allows large bursts of packets to be sent when servicing each flow [56,69], ultimately increasing response times for latency-sensitive applications. *No packet scheduler today offers low resource utilization, low latency, and adaptability to packet sizes without explicit flow information feedback.*

Our work makes several novel contributions to fair scheduling. First, we show that DRR's drawbacks, i.e., high latency for bursty flows and vulnerability to quantum choice, are measurable at the application level, even on high-speed networks. We further identify the trade-off between latency and CPU utilization to be intrinsic to DRR's design and argue that there's no correct answer to configuring DRR (§2).

Second, we outline the design of Self-Clocked Round-Robin (SCRR), a novel multi-queue round robin scheduler. SCRR borrows virtual clocking from Fair Queuing [41,42] to eliminate the quantum and enforce fairness, and services sub-queues like DRR in strict sequence for simplicity and scalability. SCRR is parameter-less, light, scalable, and adapts to bursts and packet length variations. We design various SCRR enhancements to efficiently prioritize bursty flows and reduce application latency. When an idle flow becomes active, the scheduler decides its bandwidth allowance via its virtual clock setting. Backed by strong theoretical virtual clocking principles, SCRR can efficiently adjust the virtual clocks for short, bursty flows, allowing them to quickly make progress without violating fairness bounds (§3). Third, we provide rigorous proofs of flow burstiness and fairness for SCRR, guaranteeing that SCRR has the same fairness as DRR with a low maximum burstiness bound (§4).

Fourth, we evaluate SCRR and competing schedulers on a physical testbed under Cubic and BBRv3 using novel experiments. Application-level fairness is tested with thousands of active flows while application latency is measured for request-response flows and VBR traffic. Our results demonstrate that employing SCRR in a software switch reduces CPU utilization by 23% when compared to DRR with 1500B quantum. Our results further reveal that SCRR offers 93×, 71%, and 45% lower response times over tail-drop, DRR with a small quantum, and DRR with a large quantum, respectively. (§5)

SCRR is a step toward more dynamic packet scheduling for multi-party wide-area deployments. It is the first true alternative to DRR, offering the low latency of Start Time Fair Queuing with the low complexity of DRR. Our ultimate goal is for SCRR to replace existing hardware and software packet schedulers because of its plug-and-play deployment, low resource consumption, low latency, and desirable fairness.[1]

2 Packet Scheduling for Modern Internet

Internet workloads incorporate a mix of heavy, throughput-intensive flows, as well as bursty latency-sensitive flows with diverse arrival patterns. Can Deficit Round Robin [81], as the de facto candidate for fair scheduling cope with modern traffic characteristics?

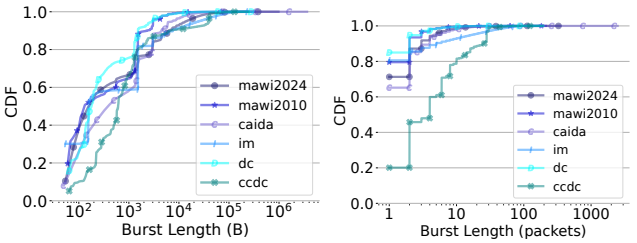
2.1 A brief history of packet scheduling

Historically, Packet-by-packet Generalized Processor Sharing (PGPS) [68] was among the first scheduling paradigms that target fair bandwidth allocation among tenants. Tenants or flows are classified into separate FIFO queues that are then processed by the packet scheduler. Subsequently, numerous Fair Queuing schedulers [32,41,42,45,66,81,90] were introduced with distinct bounds and computational complexities. Classical Fair Queuing algorithms such as Start-Time Fair Queuing (STFQ) [42] have the downside of expensive per-packet sorting operations as they need to constantly track and update the bandwidth allocation for each flow and choose the most eligible sub-queue, i.e., the sub-queue with the least progress. Further, computing requirement scales up with the number of sub-queues.

Deficit Round Robin (DRR) scheduling addresses this shortcoming by allowing constant complexity dequeue operations irrespective of scale [81]. Instead of maintaining a sorted data structure to keep track of flow progress, it visits each sub-queue in a round-robin fashion and allocates a fixed amount of bytes, called *quantum*, to each queue at each round. The part of the quantum that is too small to send the next packet is accumulated as a deficit for the queue. Other variants of DRR, including Weighted Round-Robin (*WRR*) and Modified Deficit Round Robin (*MDRR*) [12] can operate on a limited number of FIFO queues strictly reserved for different flow priorities [6]. These variants have a coarser fairness granularity (per priority group, instead of per-flow) and need to be configured with a deficit value that determines the amount of bytes each priority group transmits on each pass.

Due to its constant computational complexity and low memory footprint, *DRR* is one of the most widely deployed packet schedulers in both software and hardware [6, 7, 12, 14, 15, 81–83]. Even with the emergence of novel queuing abstractions such as *PIFO* [82] and calendar queues [78] that offer advanced traffic classification and shaping, *DRR* endures as a robust and widely used option for flow scheduling, particularly in scenarios where intricate packet

[1] SCRR artifacts are available at https://github.com/jean2/scrr.



(a) *Burst size in bytes* (b) *Packets in a burst*

Figure 1: *Diversity in burst length distributions in the wild.*

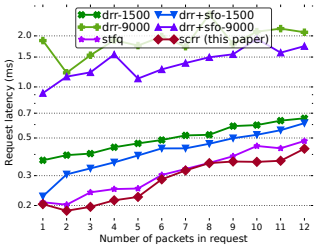


Figure 2: *DRR offers poor latency under different burst sizes.*

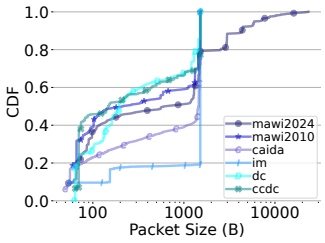


Figure 3: *Skewed packet length distributions in the wild.*

classification is unnecessary [26, 33, 37, 91, 93] and standard TCP support is required.[2]

2.2 Flow burstiness in modern traffic

Burstiness is a well-known phenomenon for Internet traffic due to its multitude of root causes [20, 22, 50, 51, 76, 92]. The emergence of interactive applications such as cloud gaming, video streaming, and video conferencing further underscores a type of traffic that experiences long idle times due to user behavior, network congestion, and variable bit-rate enforcement by the application [20, 44, 61, 75, 87]. Traffic studies further report that many latency-sensitive workloads primarily produce short flows. For example, the median response size for web requests entering Facebook data centers is reported to be less than 10 kilobytes [74], and frames in Zoom video traffic do not exceed 10 kB, with large gaps in between [61].

Fig. 1 presents the CDF of burst length distributions for various Internet traces, ranging from backbone internet (*caida*) [4], campus networks (*dc*) [22], and ISP gateways (*mawi*) [29], to instant messaging (*im*) [20] and anonymized traces (*ccdc*) [1] using an Inter-Packet Gap (IPG) threshold of 50 μs (distributions with other IPG thresholds are similar and reported in Appendix C). The diversity in burst sizes suggests that not all Internet flows tend to remain active for a long time as they arrive at a bottleneck. Notably, many flows exhibit small bursts of packets followed by large idle periods. But we also see a number of very large bursts belonging to heavy flows likely to congest the packet scheduler.

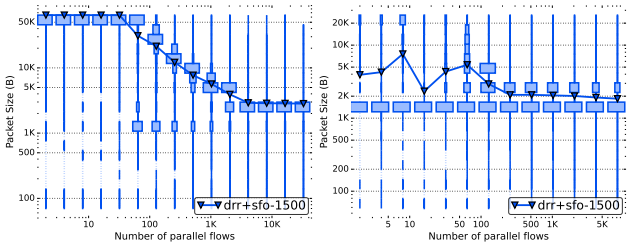
DRR has effectively the same theoretical bounds as Fair Queuing schedulers (§2.1): it is fair and has a maximum service deviation of 2× the quantum [81]. The fairness, burstiness and latency guarantees of *DRR* and Fair Queuing schedulers are based on the assumption that all flows arriving at the scheduler are always backlogged [41, 42, 81]. Alas, the above evidence suggests that this is rarely the case for modern Internet traffic! In practice, DRR usually has higher latency for those short bursty flows which are not backlogged at the scheduler, and this is seldom studied. With gaps in arrivals, short flows are prone to get stuck behind backlogged flows in the scheduler. Fig. 2 compares the response times for a request-response workload under two software implementations of *DRR*, the original design presented in [81], and the state-of-the-art implementation of *DRR* which implements *Sparse Flow Optimization (SFO)* [47], a technique that prioritizes interactive traffic (§3.3.1). We compare *DRR* with two different quantum settings: 1500B and 9000B, and with *STFQ* [42]. Two senders generate a total of 16 parallel streams of requests composed of bursts of 150B MSS packets, while the burst length increases on the x axis. Two other senders generate 64 heavy flows each to ensure the scheduler is backlogged. The full experiment setup can be found in §5.5. Latency gaps as large as 4× between *DRR* and *STFQ* suggest that *DRR* degrades the application performance due to poor scheduling decisions regardless of its quantum setting, as consecutive packets in a burst have to wait for full scheduling rounds before they are transmitted. *DRR+SFO* is initially able to bridge this gap by boosting short flows through the scheduler, but as the burst size increases, consecutive packets of the burst do not benefit from prioritization and have to wait.

2.3 The amplifying impact of packet sizes

Packet lengths are diverse in today's networks. Recent studies report a wide variety of packet length distributions observed in the network [4, 20, 22, 71, 92]. Fig. 3 presents the packet size distribution for a few Internet traces. According to these studies, heavier workloads such as instant messaging (*im*) tend to transmit MTU-sized packets while mixed workloads such as *mawi* send a wider range of packet sizes with a considerable number of short packets in the mix. The available backbone traffic packet traces [4] further highlight that packet length distribution constantly alternates in short periods (e.g., between bursts of 550B, 1500B, and 60B segments). Many modern interactive applications use small packets to reduce latency and to minimize the impact of packet loss, while some even adapt their packet sizes based on performance feedback [30, 79]. The quantum for DRR needs to be set based on the packet size [81], which makes it even harder for operators to tune their fixed quantum scheduler since these variations in packet lengths are not only workload-dependent but also volatile in the short term.

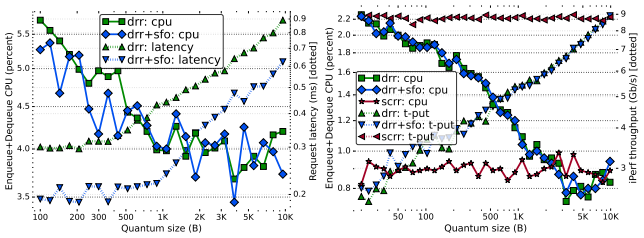
Hardware offloads and large packet buffers make it difficult to configure quanta. Software network stacks rely on

[2] We later compare the performance of state-of-the-art implementations of *PIFO*, such as AIFO [89] and *SP-PIFO* [17], with *DRR* in §5. Our results suggest that *AIFO* suffers from severe throughput loss at large RTTs, while *SP-PIFO* causes heavy packet re-ordering.



(a) *TSO on TX machine* (b) *LRO on RX machine*

Figure 4: *SKB length distribution under NIC offloading functions.*



(a) *CPU vs scheduling latency* (b) *CPU vs app. throughput*

Figure 5: *Trade-offs in DRR's quantum configuration.*

offloading support in the network interface (NIC) to keep up with their increasing bandwidth [25]. On the transmit side, *Transmit Segmentation Offload* (TSO) breaks large Socket Buffers (SKBs) into MTU-sized packets in the NIC [11]. On the receiving side, *Large Receive Offload* (LRO) aggregates small segments into larger frames [11]. In Linux, by default, the SKB size can run as high as 64 kB and with proposals to greatly increase this maximum [34]. The reduction in SKB rate provided by TSO and LRO improves software performance by reducing the overhead of context switches, cache misses, and metadata handling. However, TSO and LRO force any software packet scheduler to deal with a wide and unpredictable range of packet sizes.

For packet schedulers deployed as part of virtual switches in the datacenter or cloud, the packet size distribution will be dictated by *TSO*. Fig. 4a presents the packet size distribution of experiment §5.3, in the form of parallel histograms (wider bins indicate higher occurrence and the straight horizontal line presents the average packet size over the number of parallel flows) as received by the packet scheduler for different numbers of concurrent *Iperf* flows generated on that server. With more flows, TSO tends to generate smaller packets, as each flow has a smaller bandwidth and TSO tries to match the smaller bandwidth delay product.

For packet schedulers that are part of home routers, SD-WAN gateways, access points or middleboxes, the packet size distribution will be dictated by *LRO*. Fig. 4b presents the packet sizes generated by LRO for experiment §5.3. LRO causes less aggregation than TSO, but is also less predictable, likely due to the interplay of LRO with actual packet arrivals at the NIC and NIC interrupt coalescence. TSO and LRO make it harder to set the quantum optimally, and a quantum greater than the largest packet size, 64 kB, is much too large in most cases.

Mis-configuration with Jumbo Ethernet. Many network devices may not be optimally configured, or are configured conservatively. Most end hosts are configured for standard Ethernet, in case a network device does not support Jumbo packets. If a network device enables support for Jumbo Ethernet [57], e.g., in case some end-hosts need it, then DRR on this device would use a 9 kB quantum when the MTU is effectively 1500 B.

2.4 DRR quantum configuration trade-offs

DRR's quantum is usually set to the maximum packet size in the network (MTU) [81]. Such configuration impacts application performance and resource utilization. With a large quantum, *DRR* produces large bursts of packets that belong to a single flow and a latency-sensitive flow has to wait longer to be allocated its share, increasing the application latency (Fig. 2). Conversely, with a quantum smaller than MTU, DRR may need to spend a few scheduling rounds accumulating enough deficit to transmit a large packet, such missed visits ultimately result in wasted compute resources.[3] Fig. 5a presents the results of an experiment with the *DRR* scheduler mixing latency-sensitive 500B requests traffic (a two packet burst) with heavy background traffic using 1500B packets as an attempt to mimic the mixed and diverse nature of Internet bursts arriving at a bottleneck (the experiment is fully described in §5.5). Increasing the quantum to 10 kB increases the latency of the small requests experienced at the application level (by 140% compared to the best-case latency for *DRR+SFO*). Reducing the quantum to 100B raises the CPU utilization of the scheduler to nearly 7% of the system (around 75% increase), in this case 93% of its scheduling visits are missed due to insufficient deficit.

CPU utilization is a good proxy for the overall resource requirements of the packet scheduler. Some software implementations of packet schedulers in Internet gateways result in reduced network throughput due to their high CPU usage [2, 3]. Fig. 5b shows an incast of heavy flows sharing a bottleneck link serviced by *DRR* as we increase the CPU utilization by reducing the quantum. We use *perf* [9] to measure the CPU consumption of the scheduler's qdisc module. At higher quanta, the lower per-packet processing time is balanced by the increasing packet rate, and CPU usage is constant. A small 20B quantum makes *DRR* the bottleneck and reduces aggregate throughput to 3 Gb/s, which is a third of the capacity. During packet processing many parts of the Linux network stack must be exercised (§D.4). The CPU usage for *DRR* is less than 2.5%, however this increases CPU contention, cache misses and overall pressure on the rest of the networking stack, causing this slowdown.

[3] Some hardware implementations of *DRR* use negative deficits, however after sending a large packet, missed visits will be needed to recover the deficit and avoid unfairness. The only way to eliminate missed visits is either to accept unfairness or to restrict the quantum to be greater than MTU.

3 Self Clocked Round Robin

An ideal packet scheduler should adapt to workload dynamics, such as varying packet sizes and burstiness, both at the flow and packet level, without the need for explicit user input. Conversely, DRR requires quantum configuration based on packet lengths and flow types. In addition, a widespread deployment requires scalability, resource efficiency, and tight bounds on both performance and fairness, which cannot be satisfied altogether by Fair Queuing schedulers. In this section, we introduce *Self-Clocked Round-Robin Scheduling (SCRR)* to address those needs. We design SCRR with the following goals in mind:

- Per-flow or per-class QoS scheduling.
- Fair bandwidth allocation.
- Low scheduling latency for bursty flows.
- Low resource consumption at high loads.
- Scalability to large number of sub-queues.
- Compatibility to any transport, including TCP.
- Parameter-less and agnostic to flow information.
- Plug and play replacement for DRR and Fair Queuing.

To envision SCRR, we find that virtual clocking proves to be a great asset in realizing dynamic scheduling paradigms that adapt to workload changes. Unlike conventional deficit round-robin that sends a predetermined burst on each sub-queue visit, SCRR aims to track virtual times for sub-queues and transmit as much as a flow sub-queue's virtual time allows in order to achieve fair scheduling without relying on any flow information. In order to eliminate the expensive ordered list maintenance as in classical fair scheduling [41, 42], SCRR visits each flow sub-queue in a round-robin fashion. SCRR can be used in both low-load and high-load environments as a parameter-less improvement over DRR. (§3.1)

SCRR is flexible and is augmented by various customizations and enhancements. SCRR can optionally use weight for the sub-queues (§3.1) [32]. It can use the *virtual finish time*, like SCFQ [41], or *virtual start time*, like STFQ [42]. *No Packet Metadata* (§3.2) eliminates the need to tag packets with their virtual time while enqueued, reducing resource requirements. The combination of *Sparse Flow Optimization* (§3.3.1), *Initial Advance* (§3.3.2), and *No Empty* (§3.3.3) use the properties of virtual time to decrease the latency of bursty flows beyond the state of the art, without violating the fairness and burstiness guarantees of SCRR. With these enhancements, when both heavy and short flows are present at the scheduler, bursty flows will not have to wait for multiple scheduling rounds to make progress and enjoy lower scheduling latency.

3.1 Virtual Clocking Semantics

We start by laying out the formal foundations of the SCRR scheduler. All mathematical notations used are summarized in Table 1. The SCRR scheduler maintains a *global virtual clock*, which is updated to the maximum virtual time of dequeued packets at the end of each round. Each sub-queue features a head virtual time which is determined by the virtual time of the packet at the head of the queue. Depending on the head virtual time, sub-queues can be either *lagging* or *leading* compared to the global virtual clock. During each sub-queue visit, the scheduler allows a lagging sub-queue to catch up with the global virtual scheduling time, while a leading sub-queue advances the global virtual clock. During a period that is longer than a scheduling round, sub-queues may change their state from *lagging* to *leading* and vice versa.

Definition 1. A *scheduling round* is the period of time when all active sub-queues have been visited once, and only once. A sub-queue is active when it contains outstanding packets.

Definition 2. A sub-queue is *lagging* (catching up) as long as its head virtual time is smaller than the global virtual clock.

Definition 3. A sub-queue is *leading* as long as its head virtual time is larger or equal to the global virtual.

During each round, the scheduler visits sub-queues in round-robin and forwards all packets that have a virtual time that is older than the global virtual clock. If a sub-queue does not feature a lagging packet, the head packet is transmitted, and the scheduler moves to the next sub-queue. This ensures that no processing cycle is wasted when visiting each sub-queue, while still ensuring that fairness can be achieved via virtual clocking. SCRR effectively implements round-robin scheduling with byte-level fairness granularity and an adaptive quantum equal to the largest serviced packet in a round.

Enqueue Process. Since SCRR is a multi-queue scheduler, the arrived packets undergo a classifier to determine their target sub-queues. In implementations that target per-flow fairness, each flow is stochastically assigned to a separate sub-queue using a flow classifier (e.g., *fq* in Linux). If the sub-queue is backlogged, the virtual time of the tail packet plus its size is used as the virtual time for the new packet. However, if the sub-queue had been idle, the current global clock is used as the packet virtual time. This is to ensure that an idle sub-queue does not receive a large (and consequently, unfair) scheduling share due to inactivity for a long period.

Formally, at scheduling round k, the virtual time of packet p_f^j, i.e., the j'th packet of sub-queue f, is defined as:

$$S(p_i^j) = max\{F(p_i^{j-1}), c(k)\} \quad (1)$$

where $F(p_i^j)$ is the virtual finish time of the previous packet and $c(k)$ denotes the current virtual time of the scheduler. The virtual finish time for packet p_i^j is defined as:

$$F(p_i^j) = S(p_i^j) + \frac{l(p_f^j)}{r_f} \quad (2)$$

Table 1: *Description of mathematical notations.*

Notation	Description
p_f^j	Packet j of flow f
$l(p_f^j)$	Length of packet p_f^j
$c(k)$	Global virtual clock at scheduling round k
$v(p_f^j)$	Virtual time of packet (p_f^j)
r_f	Rate (weight) of flow f
N	Number of packets in the scheduler
Q	Vector of active sub-queues in the scheduler
n	Number of active sub-queues in the scheduler
M	Maximum length among all packets

where $l(p_f^j)$ denotes the length of the packet and r_f is the weight of the sub-queue f. Algorithm 1 in Appendix A presents the enqueue procedure.

Dequeue Process. The majority of virtual timekeeping processes take place during the packet dequeue. SCRR starts with the first non-empty sub-queue and schedules one packet. The scheduler then loops through that sub-queue and transmits packets as long as the virtual time of the head packet is older than the global virtual clock. This approach accomplishes two key objectives. First, it guarantees the transmission of at least one packet from each sub-queue, preventing missed schedule opportunities that lead to wasted compute cycles. Second, it helps the lagging sub-queues to catch up with the leading sub-queues by sending multiple packets. Upon moving to the next sub-queue, the scheduler records the start time of the last dequeued packet and cycles through the remaining active sub-queues. When a scheduling round is finished, SCRR updates the global virtual clock to the maximum packet virtual time that was transmitted during that round. Therefore, an update of the global clock only occurs once during each round to ensure the correct operation of the scheduler. The dequeue procedure is outlined in Algorithm 2 in Appendix A.

The self-clocking semantics in SCRR uses Start-Time Fair Queuing (STFQ) [42] as its basis. Alternatively, as a design choice, it is possible to schedule packets using their virtual finish times as in [41]. In §4, we perform all the correctness proofs for the *start time* version, and in Appendix B.1, we do the same for the *finish-time* version. The two self-clocking paradigms offer identical fairness and burstiness bounds for SCRR and differ slightly in the state-keeping required for the scheduler's implementation. It is also worth noting that all virtual time calculations in SCRR support sub-queue weights to implement QoS, i.e., the packet's length is divided by the weight of its corresponding sub-queue.

3.2 Eliminating per-packet metadata

Both SCFQ [41] and STFQ [42] need to store the virtual time of the packet while the packet is enqueued. Since this is usually done in the packet metadata, it has the downsides of added complexity and increased memory operations. **No Packet Metadata** (NPM) is an enhancement to the SCRR algorithm that computes packet metadata after dequeuing the packet, and therefore eliminates the need to store the virtual time of the packet as metadata.

When dequeuing a packet, the first issue is to know if the sub-queue was idle or backlogged when the packet was enqueued (§3.1). With **NPM**, the scheduler saves the global virtual clock of the previous round. If the virtual time of the last dequeued packet in the sub-queue is greater than previous virtual clock, then that sub-queue was scheduled in the previous or current round, and was backlogged. When a sub-queue is backlogged, the start time of any queued packet can be derived from the finish time of its predecessor in the sub-queue. Consequently, the sub-queue must store the finish time of the last dequeued packet, instead of the finish time of the last enqueued packet. When a sub-queue was idle, NPM always sets to virtual time of the packet to the current global clock. When the sub-queue was idle, the packet virtual time is supposed to be the value of the global virtual clock at the time the packet was enqueued (Equation 1). The insight is that in SCRR, precise virtual timing does not really matter as long as the packet is included in the proper scheduling round and its virtual time does not change the computation of the next global clock. Further, in many cases the sub-queue was added to the schedule in the current round, so the packet virtual time would have been the current global clock anyway.

NPM computes the virtual time of dequeued packets based on the scheduler's previous global time and the previous packet's finish time in the same sub-queue. This simple heuristic makes SCRR as memory efficient as DRR while making the enqueue process a breeze by removing four memory operations during insertion. The theoretical proofs obtained for SCRR also apply to SCRR with NPM, using either the finish time of packets or their start times.

3.3 Flow Latency Enhancements for SCRR

Packets of *backlogged flows* need to wait for all prior packets of the sub-queue to be dequeued, so their queuing delay is the sub-queue's length times the duration of each scheduling round. The latency of such packets is dominated by the queuing delay, therefore their exact position in the scheduling round has little impact on their latency. Further, backlogged flows are usually associated with throughput-intensive applications whose performance is dictated by the fairness of the scheduler. For backlogged flows, round robin schedulers, such as DRR and SCRR, are in practice as good as Fair Queuing schedulers, such as SCFQ and STFQ.

Non-backlogged flows, also called *sparse flows*, don't accumulate packets in the sub-queue, therefore, their packets are immediately eligible to be dequeued. The position of those packets in the scheduling round makes a big difference to their latency. Further, interactive and latency-sensitive

applications are major origins of sparse flows, or flows that alternate between sparse and lightly backlogged. Our aim is to make the latency of SCRR for sparse flows as good as the Fair Queuing schedulers.

Our enhancements for SCRR reduce the latency of *sparse flows*, and as a result make the latency of backlogged flows slightly worse than DRR. As pointed above, backlogged flows are almost never sensitive to such minor differences in latency. This design choice aims to improve the latency of flows that care the most about it. This enables to balance the scheduling performance for various types of flows without explicit flow information.[4] The above enhancements involve various trade-offs between resource usage, latency, and fairness. Hence, it is at the discretion of the network operators to decide which combination of SCRR features to use. We empirically evaluate the direct impact of each of these enhancements on application performance in Appendix D.2.

3.3.1 Sparse Flow Optimization

Sparse Flow Optimization (SFO) is a technique developed for DRR that reduces the latency of bursty flows by prioritizing sparse flows [47]. **SFO** was trivially added to SCRR. In **SFO**, sub-queues that were previously inactive (i.e. sparse) are no longer added at the back of the schedule, but instead in the *new list* of sub-queues that always take precedence over the *old list* of sub-queues (that were already part of the schedule, i.e. backlogged). To prevent sparse sub-queues from starving backlogged sub-queues, newly empty sub-queues are kept in the schedule and are only potentially removed from it during the next scheduling round [19].

3.3.2 Initial Advance for new sub-queues

The trivial version of SFO in *SCRR* does not benefit the bursty flows as well as SFO for *DRR*. With *DRR*, if multiple packets have accumulated prior to a sub-queue being scheduled via SFO, those packets can be sent in the current scheduling round, up to a quantum. With basic SCRR, when a sub-queue has been idle, the virtual time of the first packet is set to the current virtual clock (same as *SCFQ/STFQ*). When the sub-queue is scheduled with SFO, the first packet can be sent, but any subsequent packet has a virtual time in the future and needs to wait for the next scheduling round.

Initial advance fixes this issue by allowing SFO to send multiple packets in the first schedule of a previously inactive sub-queue. It simply sets the virtual time of the first packet based on the previous virtual time instead of the current virtual time (Fig. 6c). The virtual time of subsequent packets is based on the virtual time of the first packet, packet size, and the sub-queue weight, therefore setting a virtual time closer

[4]The Completely Fair Scheduler is a relevant case study. While the original process scheduler design heavily depended upon the use of nice values for latency-sensitive processes, the scheduler was gradually improved to automatically favor interactive applications [24].

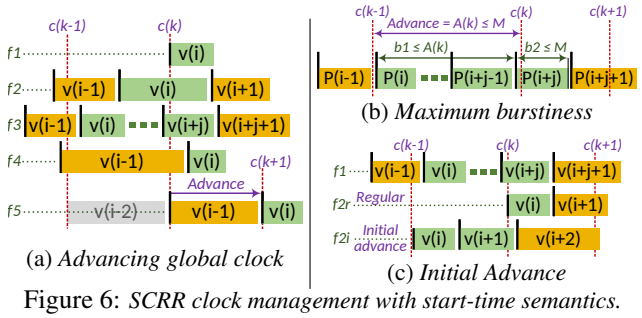


Figure 6: *SCRR clock management with start-time semantics.*

to the previous virtual clock enables potentially more subsequent packets to have a virtual time lower than the current virtual time, and consequently to send more packets in current schedule.

Initial advance will not violate the scheduler's fairness and burstiness properties. The current scheduling round processes packets from sub-queues with virtual times later than the previous virtual clock $c(k-1)$ (Fig. 6a). Therefore, setting the virtual time of the first packet to any value later than the previous virtual clock is still compatible with SCRR. To further prevent unfairness and avoid starvation, SCRR makes sure that the virtual time of a sub-queue used to tag packets can only go forward, not backward. **Initial advance** can be further tuned. To reduce the average flow burstiness, we set the virtual time of the head packet to the previous virtual clock plus the packet's size times the sub-queue weight: if all packets in the sub-queue are the same size, the burst of packets from the sub-queue scheduled by SFO is equal to the advance, instead of the advance plus an additional packet (Fig. 6b).

3.3.3 Getting rid of empty visits with No Empty

SFO in DRR keeps newly empty flows in the schedule for at least one round to prevent unfairness and starvation [19]. This means that the next time that sub-queue is scheduled, if it is still empty, the scheduler needs to remove it from the schedule and pick another sub-queue. This is problematic in software as such empty visits create extraneous cache misses. This is also problematic in hardware with fixed-delay processing pipelines, it makes the amount of work to schedule one packet unpredictable, as many successive empty sub-queues may need to be processed.

No Empty improves SFO and eliminates these empty visits by leveraging the virtual clocking semantics. With **No Empty**, idle sub-queues are never kept in the schedule. To prevent starvation, when a previously inactive sub-queue needs to be inserted into the schedule, its virtual time is checked against the current clock. If a sub-queue has transmitted more than its fair share of packets in the current schedule, its virtual time will have advanced beyond the global virtual clock. If a sub-queue has a virtual time greater than the global virtual clock, it is denied the use of SFO, and is instead added to the back of the *old list* of sub-queues, similar to the original DRR.

Otherwise, it can use SFO and is added at the back of the *new list* of sub-queues.

No Empty is especially attractive to hardware implementations as it makes the processing of SCRR with SFO completely deterministic, without loops and recursions. It also combines with **Initial advance** to benefit bursty flows. In many cases, the packets of the burst do not arrive at the same time. With the original SFO, only the packets already accumulated prior to SFO scheduling are sent in the current schedule, if a packet arrives later, the sub-queue is at the back of the schedule and the packet has to wait a full scheduling round. With **No Empty**, when this later packet arrives, the sub-queue is inactive, and can be inserted in the current scheduling round another time, as long as the virtual time of the packet is older than the current virtual clock.

4 Theoretical Analysis

In this section, we explore the theoretical properties of the scheduler and provide upper bounds for the throughput, fairness, and burstiness.

4.1 Analysis of Backlogged SCRR

Initially, we show that the movement of the global clock under SCRR is tightly bounded during each scheduling round. Next, using this insight, we establish the fairness index of the backlogged scheduler and argue that SCRR enjoys the same fairness index as DRR in the long run. Finally, we argue that the per-flow burst sizes are strictly bounded by two maximum-sized packets, similar to DRR.

Definition 4 (**Single Clock Update per Round**)**.** For the smallest time interval $[t_1, t_2]$ that includes a full round-robin scheduling round, i.e., a period when all active flows are visited at least once, SCRR global clock is updated only once according to the scheduled packet with the highest virtual time value.

Definition 5 (**SCRR Global Clock Updates**)**.** All global clock updates in SCRR occur at the end of a scheduling round, the global clock only advances forward, and the clock update marks the start of a new scheduling round.

Theorem 1 (**The Global Clock Advance Upper Bound**)**.** At each scheduling round, the global clock advancement in SCRR, $A(k)$, is bounded by the maximum packet length.

Proof. Suppose that for the SCRR scheduler with backlogged sub-queues, the global clock is set to $c(k)$ at the beginning of a scheduling round. Since SCRR guarantees the dequeue of a packet upon a visit, packet p_f^i is dequeued for flow f. The packet's virtual time (start time) is $v(p_f^i)$. The following cases can be observed (Fig. 6a):

(1) $v(p_f^i) \leq c(k)$:
The dequeued packet has a virtual time earlier than the global clock (flows *f1*, *f2*, *f3*). In this case, the flow is lagging behind the global clock and the dequeued packet will not move the global clock forward according to Definition 5. The flow will have a chance to dequeue consecutive packets as long as the above condition holds, without an impact on the global clock.

(2) $v(p_f^{i-1}) \leq c(k) < v(p_f^i)$:
The dequeued packet has a virtual time later than the global clock, therefore, this packet can move the global clock to $v(p_f^i)$ (flows *f4*, *f5*). The flow can only send a single packet, since dequeuing any packet with a virtual time after the clock will force the scheduler to switch to a new sub-queue (see §3.1). The value of $v(p_f^i) - v(p_f^{i-1})$ is the size of packet p_f^{i-1}, and is therefore always smaller than the maximum packet size. Let M denote the maximum possible packet size in the scheduler, i.e., $M = max_{f \in Q}(l_f^{max})$. In this condition, the updated virtual clock has the following bound:

$$v(p_f^i) \leq v(p_f^{i-1}) + M \leq c(k) + M$$

$$c(k+1) = max_{f \in Q}(v(p_f^i)) \leq c(k) + M.$$

No other conditions are possible, Theorem 1 follows. □

Theorem 2 (**Maximum Burstiness of SCRR**)**.** At each scheduling round, the aggregate size of a set of packets dequeued from one sub-queue is at most $(r_{\max} + 1)M$, where $r_{\max} = \max_{f \in Q} r_f$.

Proof. We denote the burst size for flow f at round k as $Burst_f^k$. If the flow has become active during the scheduling round, or if the virtual clock of the dequeued packet is later than the global clock, it is only allowed to send a single packet, according to Theorem 1.

If the virtual clock of the dequeued packet is earlier than the global clock, the flow is lagging and all packets with virtual time earlier than the global clock can be dequeued. Given that the virtual time of these packets is later than the global clock of the previous round–otherwise they would have been dequeued during the previous scheduling round–, the set of packets dequeued is constrained by:

$$c(k-1) < v(p_f^i) \leq v(p_f^{i+j}) \leq c(k)$$

The virtual time difference between the first and last packet is bounded by the clock advance, which is bounded by the maximum packet size, M:

$$v(p_f^{i+j}) - v(p_f^i) \leq c(k) - c(k-1) \leq M \tag{3}$$

In a flow, the virtual clock increase of successive packets corresponds to the size of those packets. The virtual clock is based on packet start times (see Fig. 6b for a demonstration), so the clock increase is based on the previous packet. Since $v(p_f^i) - v(p_f^{i-1}) = l(p_f^{i-1})$ and by (3),

$$v(p_f^{i+j}) - v(p_f^i) = \sum_{n=i}^{i+j-1} l(p_f^n)$$

$$Burst_f^k = \sum_{n=i}^{i+j-1} r_{\max} \cdot l(p_f^n) + l(p_f^{i+j}) \leq (r_{\max}+1)M \quad \square$$

4.2 Throughput and Fairness in SCRR

In this section, we derive theoretical bounds for SCRR.

Theorem 3 (Fairness Index for SCRR). The fairness index of SCRR, FairnessIndex_f, is 1 for any flow, f.

Proof. Suppose each flow in our system is continuously backlogged and each packet is tagged with their start-time at enqueue. Let $A(k) = c(k+1) - c(k)$, $n = |Q|$, $M = \max_{f \in Q}\left(l_f^{\max}\right)$, and $R = \sum_{f \in Q} r_f$.

By Theorem 1, $A(k) \leq M$ for each round k. If a packet has a virtual time before or equal to $c(k)$, our algorithm will send it by the end of round k. Thus, the total amount of bytes sent by flow f by the end of round k, $\text{Sent}_{f,k}$, is bounded by

$$r_f \sum_{j=1}^{k} A(j) - M \leq r_f \sum_{j=1}^{k-1} A(j) \leq \text{Sent}_{f,k} \leq r_f \sum_{j=1}^{k} A(j) + M.$$

Similarly, summing over all flows provides bounds for Sent_k, the total amount of data sent overall by round k.

$$\sum_{f \in Q}\left(r_f \sum_{j=1}^{k} A(j) - M\right) \leq \text{Sent}_k \leq \sum_{f \in Q}\left(r_f \sum_{j=1}^{k} A(j) + M\right)$$

Then combining these inequalities gives us

$$\frac{r_f \sum_{j=1}^{k} A(j) - M}{R \sum_{j=1}^{k} A(j) + n \cdot M} \leq \frac{\text{Sent}_{f,k}}{\text{Sent}_k} \leq \frac{r_f \sum_{j=1}^{k} A(j) + M}{R \sum_{j=1}^{k} A(j) - n \cdot M}.$$

As the number of rounds approaches infinity, the fairness quotient converges to r_f/R by the squeeze theorem, and thus our fairness index is

$$\text{FairnessIndex}_f = \frac{FQ_f}{\frac{r_f}{\sum_{g \in Q} r_g}} = \frac{FQ_f}{\frac{r_f}{R}} = 1. \quad \square$$

5 Testbed Evaluations

We present a scalable way to evaluate SCRR by developing a novel approach to testing packet schedulers. We use a physical testbed with synthesized traffic and perform measurements at the application level. For simplicity, we only evaluate flow scheduling, i.e., all scheduling entities (flows) have the same weight. We believe that our findings apply to QoS scheduling, where flows have different weights. Our experiments yield the following findings:

- In the presence of NIC acceleration functions, SCRR reduces the CPU utilization of the packet scheduling operations by up to 46% and 23% compared to STFQ and DRR (with 1500B quantum), respectively (§5.3).
- SCRR achieves equivalent Jain fairness to all fair scheduling schemes when facing up to 20k active flows (§5.4). E.g., SCRR maintains a Jain index of > 0.97 when scheduling up to 20k active flows.
- SCRR offers 87×, 50%, and 18% lower latency for a request-response workload, compared to tail-drop, DRR-1500 with Sparse Flow Optimization, and STFQ, respectively (§5.5).
- SCRR offers 15× and 40% lower frame latency compared to *tail-drop* and *DRR+SFO-1500*, respectively, when scheduling synthetic VBR traffic (§5.5).

A more comprehensive set of results on performance of PIFO approximations (D.1), SCRR's individual components (D.2), application performance under TCP BBRv3 congestion control (D.3), CPU utilization and throughput (D.4), flow fairness (D.5) and application performance under TCP Cubic (D.6) can be found in Appendix D.

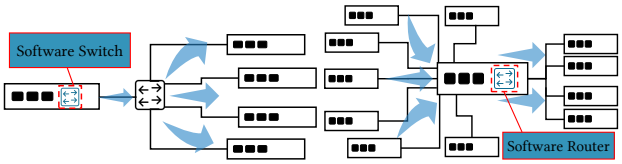
5.1 Scheduler Implementations

We implement various queuing disciplines and schedulers in Linux *tc* subsystem as a set of 'qdisc' modules for Linux version 5.10. These software implementations in the *tc* subsystem enable us to use the schedulers on a real system and employ them as part of a sender software switch, a software router, or combined with other qdiscs. Our implementations are compatible with all accelerations of the Linux TCP/IP stack, such as segmentation offloading and interrupt coalescing. Our implementations are intentionally single-threaded to remove the interference from the NIC driver's weighted round-robin scheduler among parallel packet rings [7, 84]. The evaluated systems are as follows:

Tail-drop uses the standard byte-FIFO from the Linux kernel (*bfifo*) on a single queue with the default capacity of 18 MB (15 ms of traffic) for a 10 Gbps setup. **PI2** is our own optimized version of the PI2 reference implementation [31]. As a representative of modern Active Queue Management (AQM) [73], PI2 uses a single queue and its target delay is 15 ms (the recommended default).

DRR+SFO is the Deficit Round Robin implementation from the *sch_fq* module [33]. We stripped all extraneous features from this scheduler and kept only the classifier and the scheduler. The simpler code and smaller sub-queue structure help dramatically reduce CPU usage. The classifier is based on a small hash table, and each hash bucket contains an RB-tree [10] holding the sub-queues. This implementation improves the original DRR algorithm [81] with *Sparse Flow Optimization* [47], which prioritizes newly active sub-queues to reduce the response times of latency-sensitive flows. We also include **DRR** in our results to represent the original DRR algorithm without Sparse Flow Optimization. This is the implementation widely found on hardware targets. The suffix of DRR+SFO and DRR indicates its quantum.

SCRR is our implementation of SCRR. This scheduler



(a) *Edge Switching* (b) *Software Routing*

Figure 7: *Topologies in the physical testbed.*

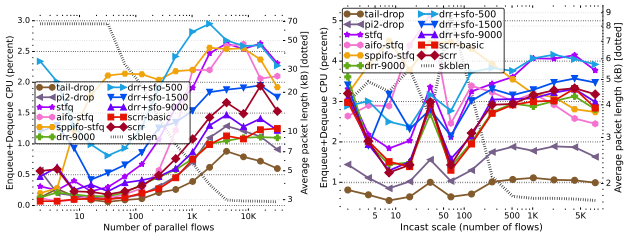
reuses the structure and classifier of the DRR scheduler, along with its sub-queue management and instrumentation. **SCRR** includes four enhancements: *No Packet Metadata* (§3.2), *Sparse Flow Optimization* (§3.3.1) *Initial Advance* (§3.3.2), and *No Empty* (§3.3.3). **SCRR-basic** refers to the SCRR algorithm without these enhancements.

STFQ is our implementation of STFQ [42]. We reuse the structure and classifier of the DRR and SCRR modules. The sub-queue pointers are stored in a sorted RB-tree [10], ordered by the virtual time of the head packets. Using an RB-tree for implementing STFQ proved to be the most efficient way to manage a dynamic sorted list– inspired by CFS [5].

AIFO and **SP-PIFO** are two approximations of Push-In First-Out queuing abstractions that use STFQ prioritization [17, 82, 89]. We implement AIFO using a single FIFO queue with default parameters set as 64 for the sampling and 10% for the burst headroom. SP-PIFO uses 8-level priority FIFOs with no restrictions on their size. We include both techniques in our results and refer the readers to Appendix D.1 for further analysis of these schemes,

5.2 Experiment Setup

We use the two topologies as depicted in Fig. 7. The edge switching topology (Fig. 7a) represents packet switching on a sender host machine, for example in a virtual switch deployed on a server in a datacenter or in the cloud. One server acts as the traffic generator and the schedulers are configured on that server to forward the traffic. This mimics a small-scale outcast traffic pattern setting. The software routing topology (Fig. 7b) is used by default and represents in-network scheduling where one software gateway routes traffic between the senders and receivers, this represents deployment on home routers, SD-WAN gateways, access points or middleboxes. The schedulers are configured on the middle server to forward the traffic between seven receiving network interfaces and one transmitting interface being the bottleneck. This topology enables us to emulate incast traffic patterns. These topologies are implemented in a testbed consisting of ten Linux servers connected via X710 10 Gbps NICs to Aruba 5406R switches. The scheduling server machine in the edge topology features 2x Intel Xeon E5-2643 CPUs totaling 24 logical cores (3.4 GHz). The scheduling server for the routing topology leverages 2x Intel Xeon E5-2680 CPUs totaling 28 cores (2.4 GHz). A subset of our experiments is done in a 25 Gbps topology consisting of four servers in



(a) *TSO* (b) *LRO*

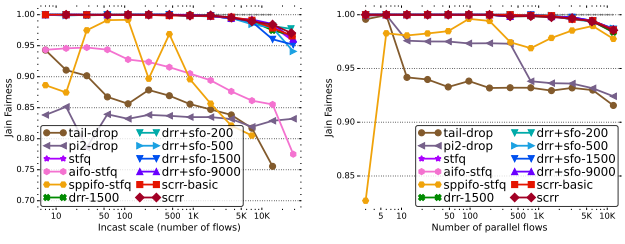
Figure 8: *Impact of increasing Incast scales on CPU utilization in the presence of NIC offloading functions.*

the edge-switching topology, connected via Intel E810 NICs and a Tofino 2 switch. By default, each experiment runs for 1 minute. The workload traffic is generated using versions 2.1.9 and 2.1.10 of *iperf* [8]. The default TCP congestion control is Cubic (refer to §D.3 for BBRv3 results). The default MSS is 1456 bytes and LRO/GRO are disabled unless specified otherwise. In some experiments, Byte Queue Limits are lowered [48], to shrink the NIC transmit queue and to reduce transmit batching, which can decrease application latency. Finally, CPU utilization is measured using Linux *perf* framework [9] by summing the usage of the enqueue and dequeue functions of each scheduler.

5.3 NIC Accelerations - TSO & LRO

We start by investigating the interaction between hardware offloading functions and the schedulers. Fig. 8a shows the impact of TCP Segmentation Offloading (TSO) on software packet schedulers as we increase the number of TCP flows in the edge switching topology (Fig. 7a). With a larger number of flows, the individual bandwidth share decreases and the average TSO packet size (dotted line) decreases from 64 kB to 1500B (full packet size distribution is in Fig. 4a), causing an increase in CPU usage in all schedulers. Decreasing the quantum of *DRR* increases CPU cycles wasted in missed visits (§2.4). The CPU utilization of *SP-PIFO* and *AIFO* is atypical and described in §D.1.1. We also observe that *SCRR-basic* scheduling automatically lowers CPU usage by adapting to the packet sizes, yielding a similar CPU overhead as *DRR-9000*. This experiment is a worst case for *SCRR*, in our other experiments it often has equal or lower CPU usage than *DRR+SFO-9000* (§5.5). Nevertheless, with 2048 flows, *SCRR* reduces the CPU utilization compared to *STFQ* and *DRR-1500* by 46% and 23%, respectively.

Fig. 8b presents the LRO packet sizes (dotted) received on the router in topology 7b, where each of the two sender machines produces half of the workload flows. The impact of LRO is a lot less predictable than TSO, and less packet aggregation is done (full packet size distribution is in Fig. 4b). *DRR* with smaller quanta cannot adapt to the dynamics of the traffic, causing CPU wastage while *SCRR* offers lower CPU overhead amongst schedulers with flow classification at high scales. Our previous results in Fig. 5b indicate that for



(a) *TCP flows and MSS bias* (b) *UDP flows and rate bias*

Figure 9: *Jain fairness index with different bias settings.*

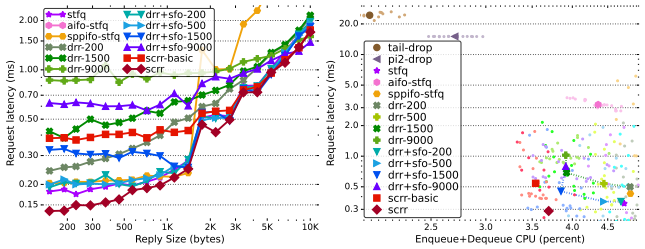
every 1% increase in the overall CPU utilization of the qdisc up to 17% reduction in throughput can be expected. In other experiments, we disable the offloading functions to prevent the schedulers from dealing with unwanted large segments.

5.4 Application Fairness

The crucial property of a packet scheduler is enforcing fairness and QoS. To better scale the experiments, we configured all sub-queues with identical weights and configured biases between flows to showcase how the scheduler overcome the bias and achieves fairness under backlogged flows.

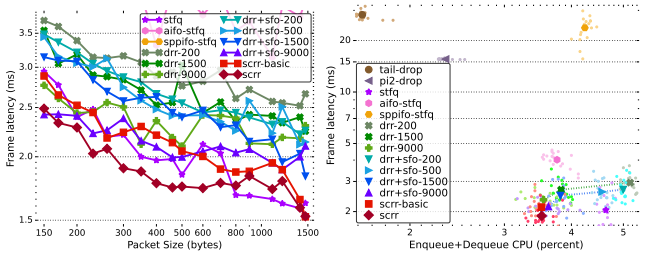
Fig. 9a uses maximum segment sizes (MSS) as the bias between TCP flows, each of the seven senders is using an MSS set to 2500, 3000, 3500, 5000, 6000, 8500, and 8956 bytes, respectively. This forces the schedulers to send more packets from flows that feature smaller packets and can verify the proper byte-level accounting of the schedulers. On each sender, the number of TCP connections varies from 1 to 4096, for a maximum of 28,672 flows. The Jain Index [49] based on the throughput of every flow shows that all Fair Queuing and Round-Robin schedulers, including *SCRR*, offer superior fairness as they enforce byte-level fairness for every flow. Since TCP congestion control is not designed to provide byte-level fairness, *tail-drop* and *PI2* show poor fairness. *AIFO* and *SP-PIFO* also show poor fairness due to their limited number of sub-queues. At a very high number of TCP flows (28k), the fairness of all schedulers is impacted by hash collisions in the classifier, a well-known caveat of using hashing in flow classifiers. Most QoS classifiers have well-defined QoS classes that do not use hashing and would not suffer from this issue. With 28k flows, some flows experience reduced performance or stalls, especially with the *tail-drop* queue.

We repeat the above experiments using UDP and flow rates as the bias in a 25 Gbps setup consisting of one sending machine and three receivers. Each connection pair is configured to offer 16.6, 12.5, and 8.3 Gbps, respectively, all bottlenecked by the sender's 25 Gbps egress bandwidth. According to Fig. 9b, a similar trend can be observed for all evaluated schedulers. The only difference is *AIFO*, which is able to throttle faster flows in order to control the queue utilization. A more comprehensive set of results on fairness using UDP and different bias settings can be found in Appendix D.5.



(a) *Response size vs latency* (b) *CPU util. vs latency*

Figure 10: *Impact of packet scheduling on request latency.*



(a) *Frame size vs latency* (b) *CPU util. vs latency*

Figure 11: *Impact of packet scheduling on VBR frame latency.*

5.5 Bursty Flow Performance

Request-response workloads. Modern networks are filled with small request-response flows [22, 62, 71]. This improved BufferBloat experiment [38] showcases how our schedulers can help those small requests when they are mixed with long-lived heavy flows. Here, two of the receiver machines generate 8 latency-sensitive flows each, the flows are a sequences of short back-to-back request-response with a request of 50B and varying response lengths. The replies are usually a short burst: the TCP ack and one or more packets for the reply. Another two senders generate 32 parallel background long-lived TCP flows each, to force the scheduler to be congested (uncongested schedulers just act as FIFOs).

Fig. 10a presents the latency of a selected subset of the schedulers. As the response size increases, they are split across more TCP segments, requiring more scheduling rounds to be forwarded, and we see a corresponding increase in latency. *SP-PIFO* has poor latency and fairness beyond one MSS, this is detailed in §D.1.3. *DRR+SFO* variants with smaller quanta are able to produce lower response times as smaller response packets experience lower queuing delays, similar to *STFQ*. *SCRR* with its latency enhancements offers superior performance compared to all other alternatives because when a sub-queue is added to the schedule, it can send multiple packets in the round instead of a single packet.

Fig. 10b averages CPU overhead and average latency over all reply sizes of fig. 10a. *Tail-drop* and *PI2* use a single queue that is quickly filled by background TCP flows, imposing a full queuing delay to the request packets. *AIFO* keeps its single queue short (§D.1.2), which only slightly helps its latency at the cost of increased CPU utilization. *SP-PIFO*'s CPU utilization is amongst the highest, and higher than *STFQ*

(§D.1.1). The performance trade-offs of *DRR* is detailed in Fig. 5a. In this experiment, 500B looks to be a sweet spot for quanta setting. Other configurations may lead to different sweet spots (see §D.6). Both *SCRR* and *SCRR-basic* adapt to packet sizes and are therefore able to offer smaller latency than the optimally configured *DRR+SFO-1500*. Specifically, *SCRR* is able to achieve lower CPU utilization than all *DRR* counterparts while improving the application latency by 87×, 1.5×, and 1.18×, compared to *tail-drop*, *DRR+SFO-1500*, and *STFQ*, respectively. A more comprehensive performance study of SCRR components can be found in Appendix D.2.

Streaming workloads. Streaming workloads are nowadays popular [55]. They commonly use *Variable Bit Rate (VBR)* encoding and reduce their packet sizes to mitigate the impact of packet losses [30, 79]. Those short bursts of small packets will alternate between sparse and backlogged, demanding low latency and high throughput performance from the scheduler. We use the *isochronous* capability in *Iperf* to emulate 64 concurrent VBR UDP flows on two servers, each targeting 1.5 Mbps and 30 Frames Per Second (FPS), which represents a typical 720p Zoom video communication bitrate. The VBR flows are competing in 7b topology with 128 long-running heavy TCP flows from two servers. Fig. 11 presents the one-way latency of each frame for different schedulers. Reducing packet size decreases performance as the header and processing overheads of small packets hurts their throughput. For *DRR*, the increased CPU utilization of short quantum hurts the throughput of the burst. *STFQ* manages low latency despite its high CPU utilization. Much like the request-response workload, the majority of VBR frames are bursty, therefore, *SCRR* which is carefully designed for bursty flow performance offers superior latency compared to all alternatives. Concretely, *SCRR* offers 15×, 1.4×, and 1.08× lower frame latency compared to *tail-drop*, *DRR+SFO-1500*, and *STFQ*, respectively. Our experiments with BBRv3 congestion control in Appendix D.3 further confirm that while congestion control has no impact on the performance of short bursty flows, *SCRR* can provide lower latency and resource utilization compared to the state-of-the-art schedulers.

6 Related Work

Starting from Nagle's [64] practical fair queuing, packet scheduling has been the focus of numerous works.

The Origin of Self-Clocking. Packet-based Generalized Processor Sharing (*PGPS*) scheduling [68] studies latency and burstiness bounds with virtual clocking and leaky bucket admission control. Self-Clocked Fair Queuing (*SCFQ*) [41] tags packets with a virtual clock, indicating their *virtual finish time* (expected end of transmission). To determine eligible queues for transmission, the scheduler maintains a sorted list based on virtual times of head packets. Upon enqueue, each packet's virtual time is calculated as the queue's current virtual time, augmented by the packet's length, proportional to the queue's assigned weight assigned. After transmission, the scheduler updates its virtual time to the transmitted packet's time.

Start-Time Fair Queuing (*STFQ*) [42] is a similar fair queuing paradigm, using virtual clocking but tags packets with their virtual start time (expected start of transmission). This virtual time equals the finish time of the tail packet if the queue is backlogged or the scheduler's global virtual time if idle. *STFQ* improves *SCFQ*'s average scheduling delays by 53% [42], but its O($log(n)$) computational complexity limits its practicality. Several works explore *STFQ*'s implementation [45, 53, 88] or attempt to approximate it in programmable hardware [17, 63, 89]. However, scalability challenges remain [45, 82], and existing approximations fail to ensure flow fairness and ideal sparse flow performance. Fair queuing schedulers have strong performance, and may outperform SCRR for some workloads or configurations. Finally, *DRR* variants [56, 69, 81] offer flow-level fairness, but using a user-specified quantum makes it difficult to argue their performance trade-offs.

Programmable and priority packet scheduling. [78] implements a programmable queuing abstraction for complex scheduling and queue management paradigms such as *Weighted Fair Queuing*. Similarly, *PIFO* [82] enables packet-level prioritization at a limited scale in hardware. As shown in Appendix D.1, existing programmable scheduling implementations suffer from under-utilization and heavy packet reordering. *AFQ* [77] similarly approximates fair queuing paradigms in programmable hardware. *Karuna* [28] presents packet scheduling for deadline flows. *SCRR* focuses on standard TCP/IP networks while allowing for future extensions.

7 Conclusions

Dating back to almost three decades ago, deficit round-robin is still one of today's widely used fair queueing paradigms. This paper identifies the diversity in packet sizes and burstiness as two critical characteristics of modern Internet traffic that make DRR's performance unpredictable and sub-optimal. We then revisit classical fair queueing and introduce the *Self-Clocked Round Robin Scheduler (SCRR)* with strong fairness bounds. We augment SCRR with enhancements that target bursty, latency-sensitive flows over the Internet. Our physical testbed evaluations, carefully crafted to probe the performance of the packet scheduler, reveal that *SCRR* offers lower resource consumption and superior latency performance compared to widely deployed scheduling and policing mechanisms.

Acknowledgments

We thank our shepherd John Sonchack and all the anonymous reviewers for their constructive feedback. We thank Hewlett Packard Labs for hosting the first two authors as interns during this work. The work is partially supported by NSF CNS NeTS grant 2313164.

References

[1] U.S. National CyberWatch Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC) Traces. https://www.netresec.com/?page=MACCDC, 2012.

[2] SQM, Cake and Piece of Cake QoS - High CPU Usage. https://forum.openwrt.org/t/sqm-cake-and-piece-of-cake-qos-high-cpu-usage/17794, 2018.

[3] SQM/QoS Can Saturate the CPU/Is This Expected or Can the Code Be Improved? https://forum.openwrt.org/t/sqm-qos-can-saturate-the-cpu-is-this-expected-or-can-the-code-be-improved/, 2019.

[4] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/catalog/datasets/passive_dataset, 2019.

[5] CFS Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html, 2023.

[6] Intel® Ethernet Controller E810 Datasheet. https://cdrdv2.intel.com/v1/dl/getContent/613875?wapkw=columbiaville%20datasheet, 2023.

[7] Intel® Ethernet Controller X710/ XXV710/XL710 Datasheet. https://cdrdv2-public.intel.com/332464/332464_710_Series_Datasheet_v_4_1.pdf, 2023.

[8] Iperf 2 Download. https://sourceforge.net/projects/iperf2/, 2023.

[9] perf: Linux Profiling With Performance Counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2023.

[10] Red-black Trees (rbtree) in Linux. https://www.kernel.org/doc/Documentation/rbtree.txt, 2023.

[11] Segmentation Offloads in Linux Kernel. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html, 2023.

[12] Understand and Configure MDRR/WRED on 12000 Series Routers. https://www.cisco.com/c/en/us/support/docs/routers/12000-series-routers/18841-mdrr-wred-18841.html, 2023.

[13] BBR. https://github.com/google/bbr, 2024.

[14] Configuring MDRR on Enhanced Queuing DPCs. https://www.juniper.net/documentation/us/en/software/junos/cos/topics/concept/cos-configuring-mdrr-on-enhanced-queuing-dpcs.html, 2024.

[15] Linux Advanced Routing & Traffic Control HOWTO. https://tldp.org/HOWTO/Adv-Routing-HOWTO, 2024.

[16] AGARWAL, S., CAI, Q., AGARWAL, R., SHMOYS, D., AND VAHDAT, A. Harmony: A Congestion-free Datacenter Architecture. In *NSDI* (2024).

[17] ALCOZ, A. G., DIETMÜLLER, A., AND VANBEVER, L. SP-PIFO: Approximating Push-in First-out Behaviors Using Strict-priority Queues. In *NSDI* (2020).

[18] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Mminimal Near-optimal Datacenter Transport. In *SIGCOMM* (2013).

[19] ARASHLOO, M. T., BECKETT, R., AND AGARWAL, R. Formal Methods for Network Performance Analysis. In *NSDI* (2023).

[20] BAHRAMALI, A., SOLTANI, R., HOUMANSADR, A., GOECKEL, D., AND TOWSLEY, D. Practical Traffic Analysis Attacks on Secure Messaging Applications. In *NDSS* (2020).

[21] BENNETT, J. C. R., AND ZHANG, H. WF2Q: Worst-case Fair Weighted Fair Queueing. In *INFOCOM* (1996).

[22] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[23] BERG, B., BERGER, D. S., MCALLISTER, S., GROSOF, I., GUNASEKAR, S., LU, J., UHLAR, M., CARRIG, J., BECKMANN, N., HARCHOL-BALTER, M., AND GANGER, G. The CacheLib Caching Engine: Design and Experiences at Scale. In *OSDI* (2020).

[24] BOURON, J., CHEVALLEY, S., LEPERS, B., ZWAENEPOEL, W., GOUICEM, R., LAWALL, J., MULLER, G., AND SOPENA, J. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *ATC* (2018).

[25] CAI, Q., CHAUDHARY, S., VUPPALAPATI, M., HWANG, J., AND AGARWAL, R. Understanding Host Network Stack Overheads. In *SIGCOMM* (2021).

[26] CAI, Q., VUPPALAPATI, M., HWANG, J., KOZYRAKIS, C., AND AGARWAL, R. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *SIGCOMM* (2022).

[27] CHANG, H., VARVELLO, M., HAO, F., AND MUKHERJEE, S. Can You See Me Now? A Measurement Study of Zoom, Webex, and Meet. In *IMC* (2021).

[28] CHEN, L., CHEN, K., BAI, W., AND ALIZADEH, M. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *SIGCOMM* (2016).

[29] CHO, K., MITSUYA, K., AND KATO, A. Traffic Data Repository at the WIDE Project, 2000.

[30] CHOI, A., KARAMOLLAHI, M., WILLIAMSON, C., AND ARLITT, M. Zoom Session Quality: A Network-Level View. In *PAM* (2022).

[31] DE SCHEPPER, K., ALBISSER, O., STEEN, H., AND TILMANS, O. sch_dualpi2: Dual Queue with Proportional Integral controller Improved with a Square. https://github.com/L4STeam, 2020.

[32] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM* (1989).

[33] DUMAZET, E. sch_fq: Fair Queue Packet Scheduler. https://lwn.net/Articles/564825/, 2013.

[34] DUMAZET, E., AND LI, C. BIG TCP. In *Netdev 0x15* (2021).

[35] FELDMANN, A., GASSER, O., LICHTBLAU, F., PUJOL, E., POESE, I., DIETZEL, C., WAGNER, D., WICHTLHUBER, M., TAPIADOR, J., VALLINA-RODRIGUEZ, N., HOHLFELD, O., AND SMARAGDAKIS, G. The Lockdown Effect: Implications of the COVID-19 Pandemic on Internet Traffic. In *IMC* (2020).

[36] FLOYD, S., HENDERSON, T., AND GURTOV, A. The NewReno Modification to TCP's Fast Recovery Algorithm. https://datatracker.ietf.org/doc/html/rfc3782, 2001.

[37] FORENCICH, A., SNOEREN, A. C., PORTER, G., AND PAPEN, G. Corundum: An Open-Source 100-Gbps NIC. In *FCCM* (2020).

[38] GETTYS, J. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing* (2011).

[39] GHABASHNEH, E., ZHAO, Y., LUMEZANU, C., SPRING, N., SUNDARESAN, S., AND RAO, S. A Microscopic View of Bursts, Buffer Contention, and Loss in Data Centers. In *IMC* (2022).

[40] GHODSI, A., SEKAR, V., ZAHARIA, M., AND STOICA, I. Multi-resource Fair Queueing for Packet Processing. In *SIGCOMM* (2012).

[41] GOLESTANI, S. J. A Self-clocked Fair Queueing Scheme for Broadband Applications. In *INFOCOM* (1994).

[42] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time Fair Queueing: a Scheduling Algorithm for Integrated Services Packet Switching Networks. *ToN* (1997).

[43] GRANT, S., YELAM, A., BLAND, M., AND SNOEREN, A. C. SmartNIC Performance Isolation with FairNIC. In *SIGCOMM* (2020).

[44] HE, J., AMMAR, M., ZEGURA, E., AND HALEPOVIC, E. QoE Metrics for Interactivity in Video Conferencing Applications: Definition and Evaluation Methodology. In *MMSys* (2024).

[45] HEDAYATI, M., SHEN, K., SCOTT, M. L., AND MARTY, M. Multi-Queue Fair Queuing. In *ATC* (2019).

[46] HEMMINGER, S. tc-netem(8): Network Emulator - Linux man Page. https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[47] HØILAND-JØRGENSEN, T. Analyzing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm. *IEEE Communications Letters* (2018).

[48] HRUBY, T. Byte Queue Limits Revisited. In *Linux Plumbers Conference* (2012).

[49] JAIN, R., CHIU, D. M., AND WR, H. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR* (1998).

[50] JIANG, H., AND DOVROLIS, C. Why Is the Internet Traffic Bursty in Short Time Scales? In *SIGMETRICS* (2005).

[51] KAPOOR, R., SNOEREN, A. C., VOELKER, G. M., AND PORTER, G. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *CoNEXT* (2013).

[52] KARAMOLLAHI, M., WILLIAMSON, C., AND ARLITT, M. Packet-Level Analysis of Zoom Performance Anomalies. In *ICPE* (2023).

[53] KORTEBI, A., MUSCARIELLO, L., OUESLATI, S., AND ROBERTS, J. Minimizing the Overhead in Implementing Flow-aware Networking. In *INCS* (2005).

[54] KUZNETSOV, A. N. tc-tbf(8): Token Bucket Filter - Linux man Page. https://man7.org/linux/man-pages/man8/tc-tbf.8.html.

[55] LEE, I., LEE, J., LEE, K., GRUNWALD, D., AND HA, S. Demystifying Commercial Video Conferencing Applications. In *MM* (2021).

[56] LENZINI, L., MINGOZZI, E., AND STEA, G. Aliquem: a Novel DRR Implementation to Achieve Better Latency and Fairness at O(1) Complexity. In *IEEE QoS* (2002).

[57] LEVY, M. Jumbo Frame Deployment at Internet Exchange Points. https://www.ietf.org/archive/id/draft-mlevy-ixp-jumboframes-00.txt, 2011.

[58] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving Resource Efficiency at Scale. In *ISCA* (2015).

[59] MACMILLAN, K., MANGLA, T., SAXON, J., AND FEAMSTER, N. Measuring the Performance and Network Utilization of Popular Video Conferencing Applications. In *IMC* (2021).

[60] MENG, Z., ATRE, N., XU, M., SHERRY, J., AND APOSTOLAKI, M. Confucius: Achieving Consistent Low Latency With Practical Queue Management for Real-time Communications. *arXiv cs.NI 2310.18030* (2023).

[61] MICHEL, O., SENGUPTA, S., KIM, H., NETRAVALI, R., AND REXFORD, J. Enabling Passive Measurement of Zoom Performance in Production Networks. In *IMC* (2022).

[62] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *SIGCOMM* (2018).

[63] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM* (2016).

[64] NAGLE, J. On Packet Switches with Infinite Storage. *IEEE ToC* (1987).

[65] NISTICÒ, A., MARKUDOVA, D., TREVISAN, M., MEO, M., AND CAROFIGLIO, G. A Comparative Study of RTC Applications. In *IS* (2020).

[66] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, Low Latency Scheduling. In *SOSP* (2013).

[67] PANG, R., ALLMAN, M., BENNETT, M., LEE, J. R., PAXSON, V., AND TIERNEY, B. A First Look at Modern Enterprise Traffic. In *IMC* (2005).

[68] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *ToN* (1993).

[69] RAMABHADRAN, S., AND PASQUALE, J. Stratified round Robin: a low complexity packet scheduler with bandwidth fairness and bounded delay. In *SIGCOMM* (2003).

[70] RAMAKRISHNAN, K., FLOYD, S., AND BLACK, D. The Addition of Explicit Congestion Notification (ECN) to IP. https://datatracker.ietf.org/doc/html/rfc3168, 2001.

[71] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In *SIGCOMM* (2015).

[72] SAEED, A., DUKKIPATI, N., VALANCIUS, V., THE LAM, V., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM* (2017).

[73] SCHEPPER, K., ALBISSER, O., TSANG, I., AND BRISCOE, B. PI2 : A Linearized AQM for both Classic and Scalable TCP. In *CoNEXT* (2016).

[74] SCHLINKER, B., CUNHA, I., CHIU, Y.-C., SUNDARESAN, S., AND KATZ-BASSETT, E. Internet Performance from Facebook's Edge. In *IMC* (2019).

[75] SENGUPTA, S., YADAV, V. K., SARAF, Y., GUPTA, H., GANGULY, N., CHAKRABORTY, S., AND DE, P. MoViDiff: Enabling Service Differentiation for Mobile Video Apps. In *IM* (2017).

[76] SHARAFZADEH, E., ABDOUS, S., AND GHORBANI, S. Understanding the Impact of Host Networking Elements on Traffic Bursts. In *NSDI* (2023).

[77] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI* (2018).

[78] SHARMA, N. K., ZHAO, C., LIU, M., KANNAN, P. G., KIM, C., KRISHNAMURTHY, A., AND SIVARAMAN, A. Programmable Calendar Queues for High-Speed Packet Scheduling. In *NSDI* (2020).

[79] SHARMA, T., MANGLA, T., GUPTA, A., JIANG, J., AND FEAMSTER, N. Estimating WebRTC Video QoE Metrics Without Using Application Headers. In *IMC* (2023).

[80] SHIEH, A., KANDULA, S., GREENBERG, A., KIM, C., AND SAHA, B. Sharing the Data Center Network. In *NSDI* (2011).

[81] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM* (1995).

[82] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *SIGCOMM* (2016).

[83] STEPHENS, B., AKELLA, A., AND SWIFT, M. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI* (2019).

[84] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *ATC* (2017).

[85] THAKKAR, P., SAXENA, R., AND PADMANABHAN, V. N. AutoSens: Inferring Latency Sensitivity of User Activity Through Natural Experiments. In *IMC* (2021).

[86] WANG, Z., LI, Z., LIU, G., CHEN, Y., WU, Q., AND CHENG, G. Examination of WAN traffic characteristics in a large-scale data center network. In *IMC* (2021).

[87] XU, X., AND CLAYPOOL, M. Measurement of Cloud-based Game Streaming System Response to Competing TCP Cubic or TCP BBR Flows. In *IMC* (2022).

[88] XU, Y., AND ZHAO, M. IBIS: Interposed Big-data I/O Scheduler. In *HPDC* (2016).

[89] YU, Z., HU, C., WU, J., SUN, X., BRAVERMAN, V., CHOWDHURY, M., LIU, Z., AND JIN, X. Programmable packet scheduling with a single queue. In *SIGCOMM* (2021).

[90] YUAN, X., AND DUAN, Z. Fair Round-Robin: A Low Complexity Packet Scheduler with Proportional and Worst-Case Fairness. *IEEE ToC* (2009).

[91] ZHANG, I., RAYBUCK, A., PATEL, P., OLYNYK, K., NELSON, J., LEIJA, O. S. N., MARTINEZ, A., LIU, J., SIMPSON, A. K., JAYAKAR, S., PENNA, P. H., DEMOULIN, M., CHOUDHURY, P., AND BADAM, A. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP* (2021).

[92] ZHANG, Q., LIU, V., ZENG, H., AND KRISHNAMURTHY, A. High-resolution Measurement of Data Center Microbursts. In *IMC* (2017).

[93] ZHANG, Y., TAN, Y., STEPHENS, B., AND CHOWDHURY, M. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI* (2022).

A Self-Clocked Round-Robin Scheduling

This section expands on the design of the Self-Clocked Round-Robin (SCRR) packet scheduler. Both *SCRR-basic* and *SCRR* with its enhancements are presented.

Algorithm 1 SCRR-basic Enqueue

```
function ENQUEUE(pkt)
    cur_queue ← classify(pkt)
    if cur_queue is inactive then
        sub_queues.tail ← cur_queue
        rounds ← rounds + 1
    end if
    pkt.virtual ← max(virtual_clock, cur_queue.clock)
    cur_queue.clock ← pkt.virt_tag + pkt.length
    cur_queue.enqueue(pkt)
end function
```

Algorithm 2 SCRR-basic Dequeue

```
Require: NP ≥ 0                                   ▷ Packet Count
function DEQUEUE                                  ▷ Packet Retrieval
    if NP == 0 then return NULL
    end if
    cur_queue ← sub_queues.head
    pkt ← cur_queue.dequeue()
    NP ← NP − 1
                                                  ▷ Global Clock Advance
    if NP == 0 then
        virtual_dequeue ← pkt.virtual + pkt.length
        virtual_clock ← virtual_dequeue
    else if pkt.virtual > virtual_dequeue then
        virtual_dequeue ← pkt.virtual
    end if
                                                  ▷ Sub-Queue Round Robin
    virtual_next ← pkt.virtual_time + pkt.length
    if virtual_next > virtual_clock or cur_queue is empty
then
        if rounds ≤ 0 then
            virtual_clock ← virtual_dequeue
            rounds ← all_queues.length
        end if
        rounds ← rounds − 1
        sub_queues.head ← cur_queue.next
        if cur_queue is not empty then
            sub_queues.tail ← cur_queue
        end if
    end if
     return pkt
end function
```

A.1 Pseudo code of SCRR-basic

SCRR-basic is the simplest of all the variants of SCRR. The pseudo-code for the **enqueue process** of SCRR-basic is shown in Algorithm 1. This is the process that is run when a new packet arrives at the scheduler. First, the arrived packet passes through the classifier to obtain its sub-queue handle, i.e. the sub-queue where it will be enqueued. When doing per-flow classification, the 5-tuple from the packet header is used to select a sub-queue (source IP, destination IP, protocol, source transport port and destination transport port). Other QoS classifiers can also be used with SCRR. The current sub-queue may be active (already in the schedule) or inactive (empty). In the case the sub-queue is inactive, it is added at the end of the schedule, the schedule is a simple linked list of sub-queues. Tagging a packet with its virtual time is implementation dependent and usually done in the metadata associated with the packet. The packet virtual time is computed here exactly like *STFQ* [42], i.e., the most recent of the global clock and the finish time of the previous packet added to the sub-queue. The virtual time of the sub-queue is updated with the finish time of the new packet by adding its length in bytes. In the weighted version of SCRR, the packet length would be multiplied by the sub-queue weight. Ultimately, the packet is added to the SKB list of the corresponding sub-queue.

The pseudo-code for the **dequeue process** of SCRR-basic is shown in Algorithm 2. This is the process that is run when a packet needs to be sent over the outgoing link. First, a packet is extracted from the current sub-queue. Then, the global clock advance is computed. While conceptually similar to *STFQ* [42], SCRR makes sure that the clock can only go forward. Then, to advance the global clock, the scheduler maintains a temporary *virtual_dequeue* clock that tracks the largest virtual time of packets dequeued during the scheduling round. After finishing a scheduling round, the global clock is set to *virtual_dequeue*. The next part is where SCRR distinguishes itself from *STFQ*. SCRR uses the virtual time of the next packet in the sub-queue to decide if it stays on that sub-queue or round robin to the next sub-queue. If the virtual time is in the future (greater than current clock), or if the sub-queue is empty, then the scheduler moves on to the next sub-queue (that will be used next time the function is called). If a number of scheduling rounds equal to the number of sub-queues has elapsed, the virtual clock is updated with *virtual_dequeue* (the largest of the packet virtual times dequeued during that scheduling round). If the sub-queue is not empty, it is put at the back of the schedule, so that its packets can be sent at the next scheduling round.

The dequeue process must properly keep track of scheduling rounds. The enqueue process may add a previously inactive sub-queue to the current scheduling round. Tracking the scheduling round is needed to ensure that the global clock is only updated between rounds. If the global clock were to be

Algorithm 3 SCRR Enqueue (with enhancements)

```
function ENQUEUE(pkt)
    cur_queue ← classify(pkt)
    if cur_queue is inactive then
                                                ▷ No Empty
        if cur_queue.clock <= virtual_clock or nq == 0
then                                  ▷ Sparse Flow Optimization
            new_queues.tail ← cur_queue
            rounds ← rounds + 1
        else
            old_queues.tail ← cur_queue
        end if
    end if
    cur_queue.enqueue(pkt)
end function
```

updated after each packet transmission, the later sub-queues might fall further behind the global clock, leading to large unwanted packet bursts.

The algorithm describes SCRR-basic using the *start-time* of the packet like STFQ [42]. SCRR can also be implemented using the *finish-time* of the packet, similar to SCFQ [41]. We chose the *start-time* because it enables us to make SCRR's implementation simpler by computing the virtual time of the next packet without any information about that packet.

A.2 Pseudo code of SCRR with enhancements

SCRR, for brevity, refers to the variant of SCRR scheduler that includes all the enhancements, *No Packet Metadata*, *Sparse Flow Optimization*, *Initial Advance* and *No Empty*. In this section, we will only highlight the differences from SCRR-basic in §A.1.

The pseudo-code for the **enqueue process** of SCRR is shown in Algorithm 3. SCRR uses the *No Packet Metadata* enhancement, therefore the packet virtual time is not computed in the enqueue process, but in the dequeue process. Consequently, the packet is not tagged with its virtual time. In case the sub-queue has been inactive, SCRR uses the *No Empty* enhancement to decide how to schedule that sub-queue. If the finish time of the last packet on this sub-queue is in the past (earlier than the current clock), then the new packet can be fit into the schedule. In this case, it is eligible to use *Sparse Flow Optimization* and the inactive sub-queue is added to the list of sub-queue for priority scheduling. Otherwise, it is added at the end of the schedule.

The pseudo-code for the **dequeue process** of SCRR-basic is shown in Algorithm 4. SCRR uses *Sparse Flow Optimization*, therefore it first looks for a sub-queue in the list of previously inactive sub-queues, and only if that list is empty, it looks in the regular schedule. With *No Packet Metadata*, packet virtual time can be computed in dequeue. If the virtual finish of the last packet on the sub-queue is older than the virtual clock of the previous scheduling round, then that sub-queue was inactive. If the queue was inactive, and *Initial*

Algorithm 4 SCRR Dequeue (with enhancements)

```
Require: NP ≥ 0                                  ▷ Packet Count
Require: nq ≥ 0                    ▷ Active Sub-queue Count
function DEQUEUE                                 ▷ Packet Retrieval
    if NP == 0 then return NULL
    end if
                                      ▷ Sparse Flow Optimization
    cur_queue ← new_queues.head()
    if cur_queue is null then
        cur_queue ← old_queues.head()
    end if
    pkt ← cur_queue.dequeue()
    NP ← NP − 1
                                                ▷ No Packet Metadata
    if cur_queue.clock < virtual_previous then
        if config.initial_advance then          ▷ Initial Advance
            virtual_pkt ← virtual_previous + pkt.length
        else
            virtual_pkt ← virtual_clock
        end if
    else
        virtual_pkt ← cur_queue.clock
    end if
    virtual_next ← virtual_pkt + pkt.length
    cur_queue.clock ← virtual_next
                                          ▷ Global Clock Advance
    if NP == 0 then
        virtual_dequeue ← pkt.virtual + pkt.length
        virtual_previous ← virtual_clock
        virtual_clock ← virtual_dequeue
    else if pkt.virtual > virtual_dequeue then
        virtual_dequeue ← pkt.virtual
    end if
                                        ▷ Sub-queue Round Robin
    if virtual_next > virtual_clock or cur_queue is empty
then
        if rounds ≤ 0 then
            virtual_previous ← virtual_clock
            virtual_clock ← virtual_dequeue
            rounds ← nq
        end if
        rounds ← rounds − 1
        rr_queue ← rr_queue.next
        if cur_queue is not empty then
            old_queues.tail ← cur_queue
        end if
    end if
     return pkt
end function
```

Advance is enabled, the packet virtual time is based on the virtual clock of the previous scheduling round, this enables the sub-queue to potentially send a few packets in the current scheduling round before its clock advances beyond the global clock. Otherwise, it uses the current virtual clock, and the sub-queue can only send one packet in the current schedule. If the sub-queue has been active, the packet virtual time is based on the virtual time of the previous packet in that sub-queue, like in *SCRR-basic* and *STFQ*. After that, the sub-queue's clock is advanced based on the packet length and optionally the sub-queue's weight. The rest of the dequeuing process is identical to *SCRR-basic*, except the need to keep track of the global clock of the previous scheduling round.

B Extended Theoretical Analysis

This section expands the theoretical properties of the scheduler, extending our findings in §4. We present the analysis of SCRR using finish-time semantics.

B.1 Analysis of Backlogged SCRR Using Finish-time Semantics

Section 4.1 studies the SCRR scheduler using start time, similar to STFQ [42]. SCRR can also use finish time, similar to SCFQ [41]. Using the finish time in SCRR does not change the way the schedule behaves and similar proofs can be obtained for SCRR's fairness.

In Theorem 1, the packet virtual time is $v(p_f^i)$ and is equal to its finish-time (Fig. 12). In case (2), The value of $v(p_f^i) - v(p_f^{i-1})$ is the size of packet p_f^i (instead of packet p_f^{i-1}), this packet is still smaller than the maximum packet size, therefore the bound for case (2) is:
$v(p_f^i) \leq v(p_f^{i-1}) + M \leq c(k) + M$.
$c(k+1) = max_{f \in Q}(v(p_f^i)) \leq c(k) + M$.

The bound for case (2) is unchanged, and the other case is unchanged, therefore Theorem 1 also applies when using finish time.

In Theorem 2, equation 3 remains unchanged. Since the virtual clock is based on packet's finish time (Fig. 13), the increase in the global clock occurs based on the current packet (instead of the previous packet) :

$$v(p_f^{i+j}) - v(p_f^i) = \sum_{n=i+1}^{i+j} l(p_f^n)$$

$$Burst_f^k = \sum_{n=i}^{i+j} l(p_f^n) = l(p_f^i) + \sum_{n=i+1}^{i+j} l(p_f^n)$$

$$Burst_f^k \leq M + M$$

Throughput Bounds and Fairness. The proof of fairness with the finish time is identical to the proof using start-time (§4.2). The only difference is that we define $A(k) = c(k) - c(k-1)$ (instead of $A(k) = c(k+1) - c(k)$). The rest of the proof is unchanged, and as the number of rounds approaches infinity, the fairness quotient converges to $\frac{1}{n}$ and the fairness index converges to 1.

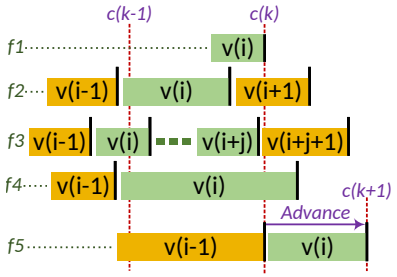


Figure 12: *Advancing the global clock in SCRR with finish-time semantics.*

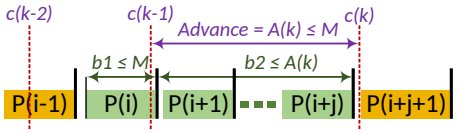


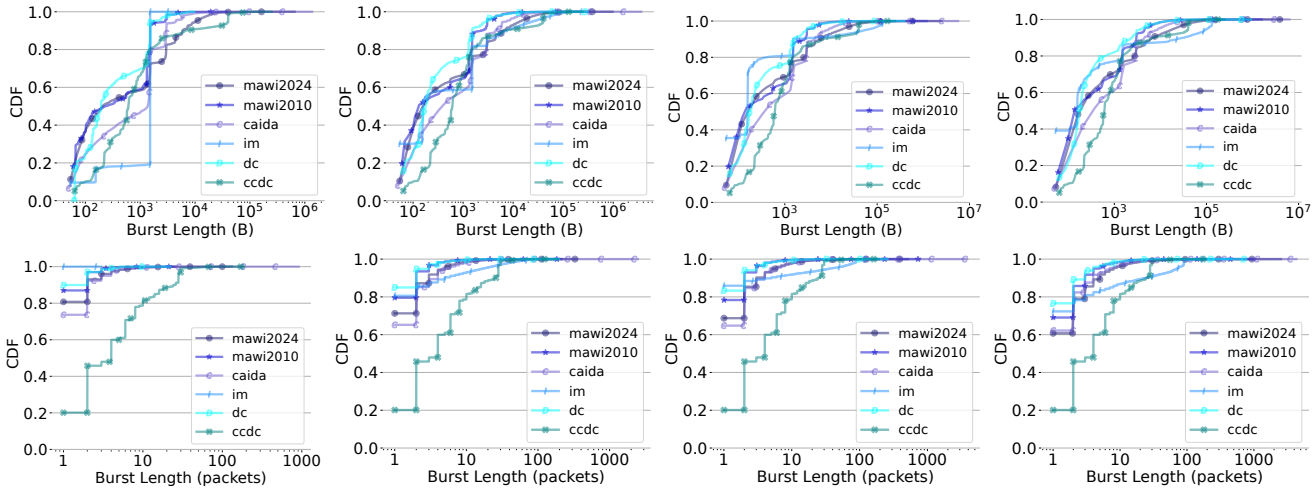
Figure 13: *Burstiness of SCRR with finish-time semantics.*

C Burstiness in Internet Traces

A burst is a train of packets with inter-packet gaps smaller than a threshold [51]. Using this definition, we plot the CDF of burst lengths for six representative Internet traces [1, 4, 20, 22, 29] in Fig. 14. Regardless of the threshold, a diverse range of burst lengths can be observed which suggests that Internet workloads feature both heavily backlogged and short flows arriving at the bottlenecks. In fact. the similarity among burst size distributions with different Inter-Packet Gap (IPG) thresholds suggest that our analysis of Internet bursts in §2 is independent of the chosen threshold. Additionally, the above figures highlight the presence of back-to-back packets with very small gaps, forming up a burst, and large inactivity gaps between bursts. This can be partly attributed to factors such as user behavior, especially for workloads like video streaming. We explore the performance of packet scheduling under VBR workloads in §5.5 and more thoroughly in Appendices D.3 and D.6.

D Extended Testbed Evaluation

This section presents an extended version of the testbed evaluation in §5, putting together our results of scheduling in the presence of NIC offloading, fairness experiments, and application latency experiments, *SCRR*'s component analysis, and a closer look at the performance of existing *PIFO* implementations.



(a) *IPG < 10µs* (b) *IPG < 50µs* (c) *IPG < 100µs* (d) *IPG < 500µs*

Figure 14: *Burst length distribution for Internet traces using various IPG thresholds.*

D.1 Performance of *PIFO* Approximations

Push-In First-Out abstraction [82] presents an interesting breakthrough in realizing priority packet scheduling, paving the way for maintaining priority lists in hardware that can realize schemes such as *STFQ* [42] and *pfabric* [18]. To overcome the practical roadblocks of deploying *PIFO*, recent works attempt to approximate *PIFO* using admission control on a single queue [89], and using strict-priority FIFOs [17].

D.1.1 CPU utilization in *AIFO* and *SP-PIFO* is high

The CPU utilization profile of *AIFO* is strikingly different from conventional packet schedulers. Fig. 15 shows the CPU utilization of the enqueue and dequeue functions separately for the experiment shown in Fig. 8a (§5.3). The CPU utilization of *AIFO* dequeue function is lower than even Tail-drop (Fig. 15b), because *AIFO* keeps the queue much shorter than other packet schedulers (§D.1.2) and reduces the overall cache footprint. On the other hand, the CPU utilization in enqueue is very high (Fig. 15a) due to the rank sampling and percentile calculation [89]. Since the rank sampling of *AIFO* can be tuned, we configured *AIFO* to sample the rank every 16 packets and to maintain up to 64 samples. We increased the maximum number of samples from the default [89] to improve *AIFO*'s fairness. In practice, the number of samples is usually much lower than 64, because only the samples still in the queue are considered, and the queue is kept short. Changes to the rank sampling rate and the maximum number of rank samples would obviously impact *AIFO*'s fairness and CPU utilization, however we haven't found a configuration giving better overall performance than *STFQ* in our experiments.

The CPU utilization profile of *SP-PIFO* is more balanced. In enqueue, there is additional overhead in choosing the proper priority sub-queue and keeping track of the rank of each priority sub-queue (Fig. 15a). In dequeue, there is additional overhead in scanning the priority sub-queues (Fig. 15b). For the majority of data points, *SP-PIFO* has higher CPU utilization than *STFQ* with higher application latency (Fig. 21, 22, 23).

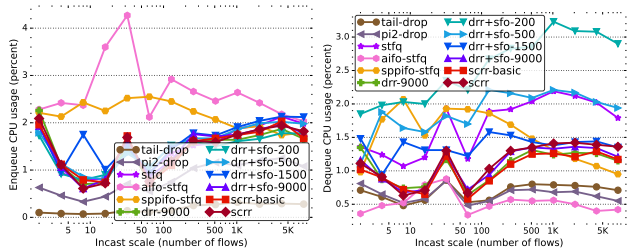
D.1.2 *AIFO* causes queue underutilization

AIFO selectively drops packets prior to admitting them in its single queue. For each packet, it computes a rank, and compares this rank to the distribution of ranks for packet already in the queue. If the percentile of the packet is lower than the current queue occupancy, the packet is dropped, otherwise it is admitted. *AIFO* implements *STFQ* by calculating the packet rank from packet virtual times.

In Fig. 16a, a single TCP flow with increasing RTT is run through a bottleneck. The bottleneck is created using Linux TBF rate limiter [54], while the RTT in increased by adding a fixed delay with Linux *NetEm* [46]. Each scheduler is configured for 15 ms queuing delay. The theory of buffer sizing [46] predicts that the 15 ms queue should be large enough for TCP Cubic traffic with RTT up to 35 ms. This is the case for all schedulers, except *AIFO*.

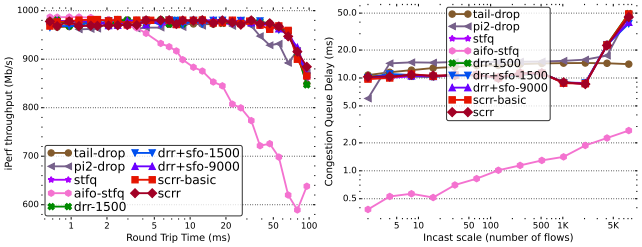
The problem is that when using *STFQ*, the statistical distribution of ranks is highly biased, and the statistical admission control of *AIFO* is no longer appropriate. With a single flow, the virtual time of incoming packets is strictly increasing, as it is always larger than any other packet in the queue, and is therefore always rejected based on its rank. *AIFO* also includes a burst mechanism: when the queue utilization is below 10%, packets are never dropped. As a result, with a single flow, *AIFO* can only use the portion on the queue configured for burst. This underutilization of the queue has direct impact on TCP performance as depicted in Fig 16a.

Fig 16b shows that even for very large number of flows



(a) *LRO - Enqueue* (b) *LRO - Dequeue*

Figure 15: *Impact of increasing Incast scales on CPU utilization in the presence of NIC offloading functions.*



(a) *Throughput as RTT increases* (b) *Queue utilization*

Figure 16: *Throughput and queue utilization of AIFO under different schedulers as RTT and flow count increase.*

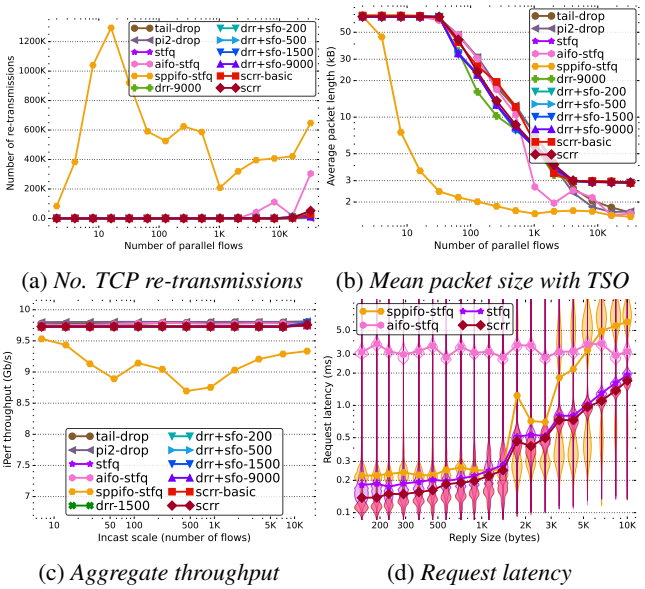
(10k), the statistics of packet virtual times are still sufficiently biased to show severe queue uderutilization with *AIFO*. The amount of queue underutilization decreases with the number of flows, therefore it is not possible to fully compensate for it.

D.1.3 *SP-PIFO* causes packet reordering

We implement *SP-PIFO* using the semantics outlined in [17] and compare its performance against our representative packet schedulers. Fig. 17a presents the TCP re-transmission count for all schemes for the TSO experiment in §5.3. Even with heavy flows congesting the scheduler, *SP-PIFO* suffers from intense packet re-ordering, resulting in unwanted TCP re-transmissions. This in turn impairs the large receive offload functions to save CPU by creating larger packets as depicted in Fig. 17b, ultimately resulting in higher CPU utilization. Hence, *SP-PIFO*'s flow migration across priorities, comes at the cost of heavy re-ordering, starvation, TCP re-transmissions and higher CPU consumption for heavy flows.

Next, we repeat the fairness experiments previously reported in Fig. 9a on a testbed with similar configuration but faster CPUs and report the results in Fig. 17c (§5.4). With faster CPUs, all schedulers except *SP-PIFO* can achieve line rate regardless of the incast scale, even in the case of resource-hungry *DRR+SFO-200*. Similar to the above case, *SP-PIFO* suffers from unwanted TCP re-transmissions that result in reduced throughput.

Fig. 17d presents the request-response workload performance under various packet schedulers; it is an extract of the experiments previously reported in 10a (§5.5). The vertical violins in Fig. 17d represent the distribution of latencies for



(a) *No. TCP re-transmissions* (b) *Mean packet size with TSO*

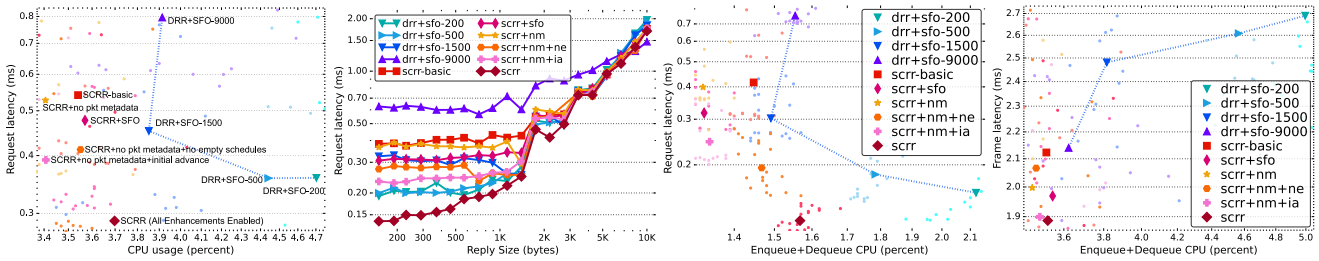
(c) *Aggregate throughput* (d) *Request latency*

Figure 17: *SP-PIFO performance with heavy flows and TSO (top row), under heavy flows (bottom left), and under request-response workloads (bottom right).*

each reply size and each scheduler. Other schedulers have a tight distribution of latency, whereas the latency distribution of *SP-PIFO* is wide and mostly bimodal when the reply size spans multiple packets. Although many requests are lucky to benefit from the highest priority band and are quickly completed with a latency even lower than *STFQ* and *SCRR*, some requests are heavily penalized and lead to poor average performance. This is because, while the first response packet will have a high chance of using the highest priority band, the consecutive packets will have a larger virtual time and will be inserted into lower bands, hence facing starvation.

D.2 *SCRR* Component Analysis

Fig. 18a compares the CPU utilization and response latency of different *SCRR* components for the request-response experiment reported in §5.5. The basic *SCRR* algorithm (without bursty flow enhancements) offers lower CPU utilization compared to DRR alternatives. The *no packet metadata* variant offers even better CPU performance due to imposing less memory pressure. The *No Empty Schedule* enhancement prevents the *Sparse Flow Optimization* in *SCRR* from performing unwanted visits into stale flow entries in the old flows list by immediately kicking the flows out of the schedule and re-inserting them back when they become active. This results in improved responsiveness by lowering the latency by 23% compared to the basic algorithm. The *initial advance* enhancement has a similar effect on sparse flows by allowing them to send a slightly larger burst without violating the fairness bounds. When both enhancements come together in *SCRR*, the best response latency results can be seen, improving the basic algorithm by 46% while performing 18% faster than



(a) *CPU util. vs latency* (b) *Response size vs latency* (c) *Edge: Varying Request Size* (d) *Router: Varying VBR pkt size*

Figure 18: *Comparing the performance of individual SCRR components for the request-response experiment in §5.5.*

DRR+SFO with 200B quantum configuration. Finally, with increasing reply lengths, the upper hand of Sparse Flow Optimization starts to fade as the latency-sensitive flows become backlogged in the scheduler and exhaust their fair share (Fig. 18b). We repeat this experiment on the Edge switching topology and report the results in Fig. 18c in which similar trends in the CPU utilization and response times can be observed for SCRR variants. Finally, Fig. 18d presents the CPU vs latency trade-off for a VBR traffic on the routing topology. SCRR-basic shows a better adaptation to the VBR traffic compared to DRR by offering 15% lower latency and 8% lower CPU utilization with respect to DRR-1500 setting. Other enhancements of SCRR further increase the latency gap by 16%.

D.3 Application Performance Under BBRv3

Request-response Flow Performance. Small request-response flows are one of the most prevalent traffic patterns of modern applications [22, 62, 71]. We already presented experiments with request-response flows using TCP Cubic in §5.5 and §D.6, this section repeats those experiments with BBRv3 and highlight the differences between using BBRv3 and TCP Cubic. We integrated BBRv3 from its official repository [13] and use a different testbed with identical configuration to Fig. 7 with Linux machines running kernel 6.10. The main difference is that on this testbed the scheduler is running on much faster CPUs (Intel Xeon W5-3435X), and therefore the packet schedulers with high CPU usage are less penalized.

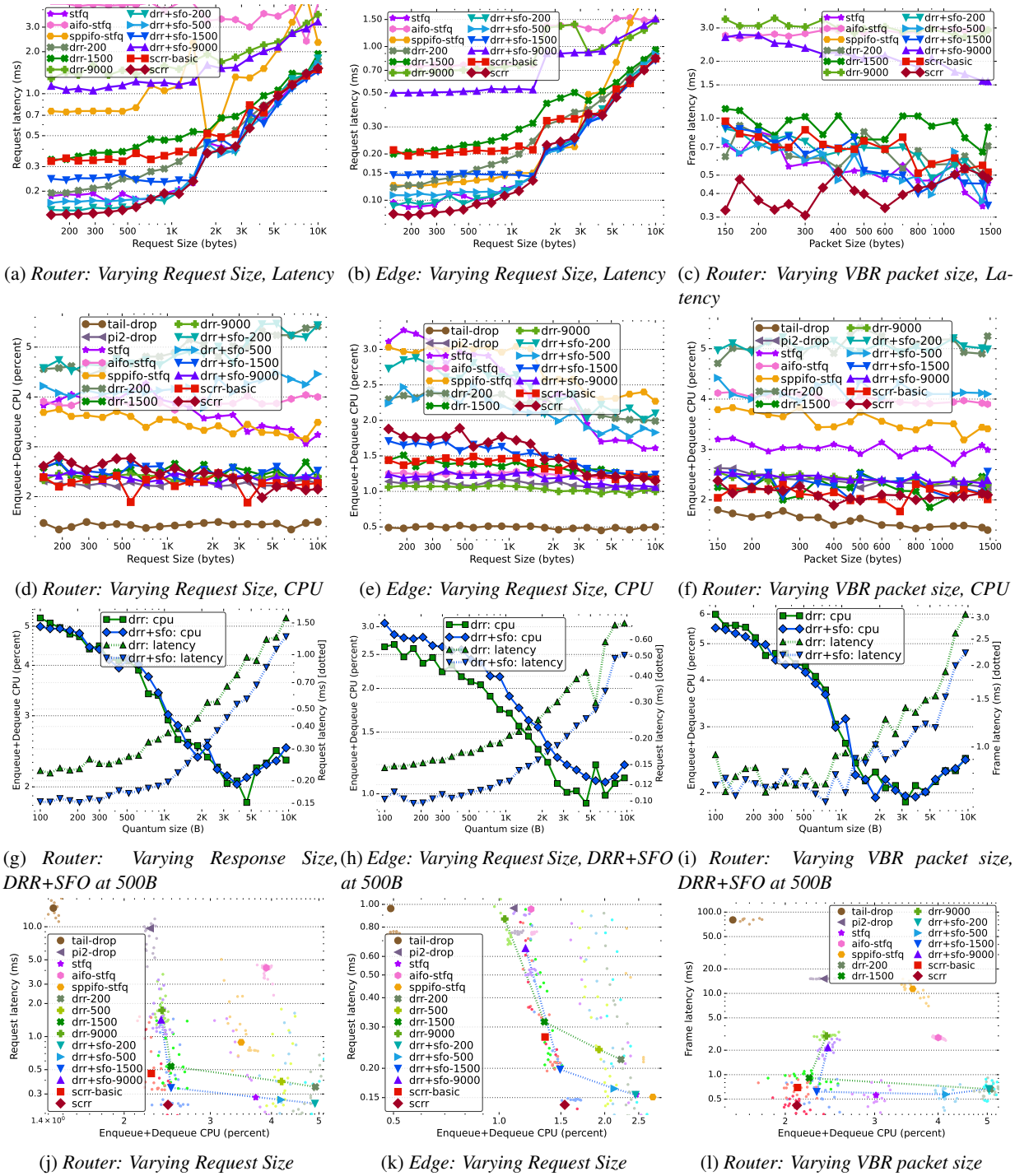
Fig. 19a depicts the application performance as we increase the size of the requests from 100B to 10 kB while the reply size is set to 100B on the router in topology 7b. For this experiment, two sender machines generate 8 latency-sensitive flows each, similar to the Cubic experiments (§5.5). Another two senders generate 64 parallel background long-lived TCP flows each. All flows are configured to use BBRv3. We intentionally opt not to enable Explicit Congestion Notification (ECN) [70] support in BBRv3 as it can arbitrarily impact scheduling performance, and the integration of ECN in *SP-PIFO* and *AIFO* is not trivial and undefined [82, 89].

Fig. 19d shows that with the faster CPU, the CPU utilization of all schedulers is reduced, and has also less jitter. Fig. 19g illustrates that for request-response workloads, the quantum configuration for DRR under BBRv3 offers the usual tradeoff between latency and CPU usage along with the benefits offered by SFO. Finally, BBRv3 offers a tighter control over queue utilization, which causes all single-queue schedulers, i.e., tail-drop, *PI2*, and *AIFO*, to have a smaller average queuing latency. Nonetheless, *SCRR* is able to offer superior latency compared to all alternatives, with low CPU usage (Fig. 19j). *SCRR* has the lowest average latency (0.24 ms), closely followed by *DRR+SFO-200* (0.25 ms), *DRR+SFO-500* (0.27 ms), *STFQ* (0.28 ms) and *DRR+SFO-1500* (0.34 ms).

Figs. 19b, 19e, 19h, and 19k repeat the request-response experiment on the edge topology (7a). Again, the same trends in both application latency and CPU utilization in the face of different request sizes can be observed. Linux's control over the transmit socket buffer size prevents the backlog to exceed 1 ms for tail-drop which significantly improves its application response time. As a result, both *PI2* and *AIFO* do not show any meaningful latency improvements over *Tail-Drop*. Those results are pretty much equal to our results with Cubic (§D.6), indicating again that the congestion control has little effect on overall performance in this setup. *SCRR* has the lowest average latency (0.14 ms), closely followed by *SP-PIFO* (0.15 ms), *DRR+SFO-200* (0.15 ms), *STFQ* (0.15 ms), *DRR+SFO-500* (0.16 ms) and *DRR+SFO-1500* (0.19 ms).

Streaming Flow Performance. The traffic of video conferencing applications is another important traffic pattern of modern applications [52, 61, 79]. We already presented experiments with Variable Bit Rate (VBR) flows using TCP Cubic in §5.5 and more extensively in §D.6, this section repeats those experiments with BBRv3 and highlight the differences between using BBRv3 and TCP Cubic.

VBR experiment results under BBRv3 are depicted in Figs. 19c, 19f, 19i, and 19l. In this experiments, 2 senders generate 32 concurrent VBR UDP flows each, and two other servers generate 64 long-running heavy TCP flows each. Here, the major difference from Cubic experiments (§5.5) is that *DRR*'s overall latency is barely affected when changing the quantum, which is attributed to the faster CPU. The smaller quanta does increase the CPU utilization, and for this experiment, the optimal quantum is around 1500 B. Furthermore, *SFO* improves the latency of *DRR* at little CPU cost. As usual, *SCRR* meaningfully reduces both the latency and the CPU uti-



(a) *Router: Varying Request Size, Latency* (b) *Edge: Varying Request Size, Latency* (c) *Router: Varying VBR packet size, Latency*

(d) *Router: Varying Request Size, CPU* (e) *Edge: Varying Request Size, CPU* (f) *Router: Varying VBR packet size, CPU*

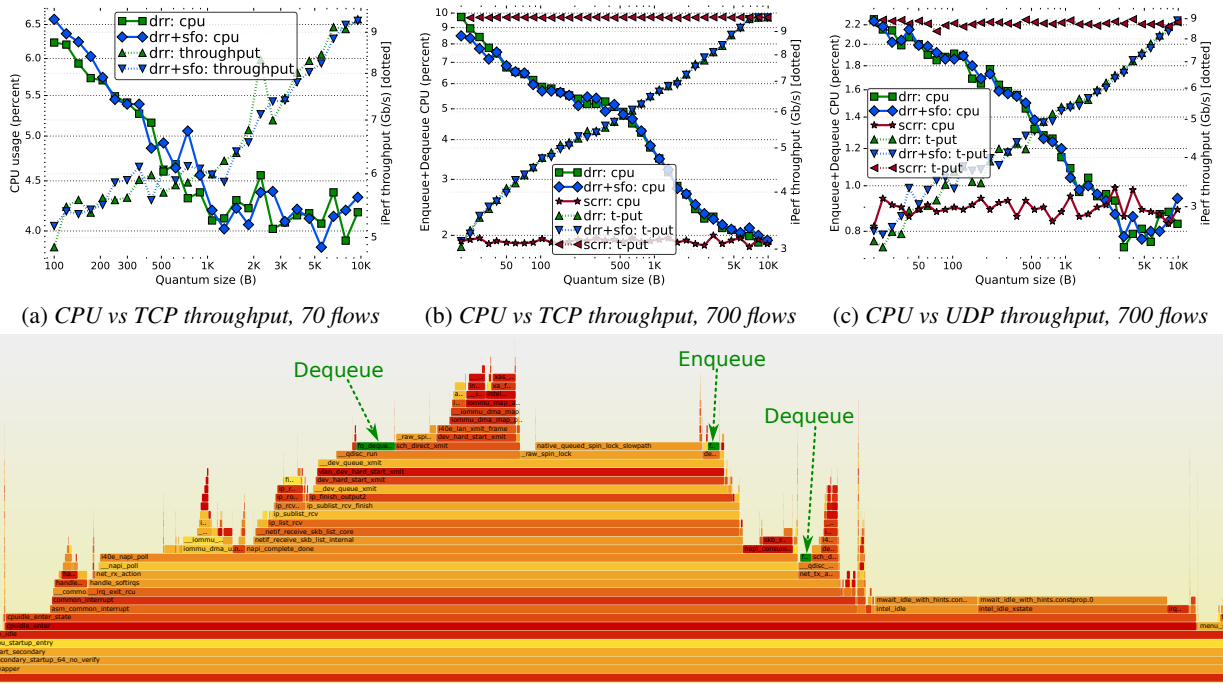
(g) *Router: Varying Response Size, DRR+SFO at 500B* (h) *Edge: Varying Request Size, DRR+SFO at 500B* (i) *Router: Varying VBR packet size, DRR+SFO at 500B*

(j) *Router: Varying Request Size* (k) *Edge: Varying Request Size* (l) *Router: Varying VBR packet size*

Figure 19: *Impact of packet schedulers on application response times for latency sensitive workloads with BBRv3 transport.*

lization. In this experiment, *SCRR* has the lowest average latency (0.42 ms), followed by *STFQ* (0.56 ms), *DRR+SFO-500* (0.57 ms), *DRR+SFO-1500* (0.62 ms), *DRR-200* (0.66 ms), *DRR+SFO-200* (0.67 ms), *SCRR-basic* (0.70) and *DRR-200* (0.71 ms).

We observe that scheduling paradigms follow a similar trend under both Cubic and BBRv3. BBRv3 includes advanced techniques to pace packets, to adjust the sending rate, and to try to keep the latency small at the queue, plus a modified slow-start algorithm [13]. However, the bursty requests are too short to show any significant difference compared to Cubic (§5.5). Our results confirm that unlike packet scheduling, congestion control has limited impact on the performance short bursty flows and *SCRR* can complement widely used In-



(a) *CPU vs TCP throughput, 70 flows* (b) *CPU vs TCP throughput, 700 flows* (c) *CPU vs UDP throughput, 700 flows*

(d) *FlameGraph, TCP, DRR, Quantum 500 B*

Figure 20: *Impact of DRR quantum on CPU overhead and traffic throughput.*

ternet congestion control by providing fairness, low resource consumption and superior application performance for such flows.

D.4 Impact of CPU overhead on throughput

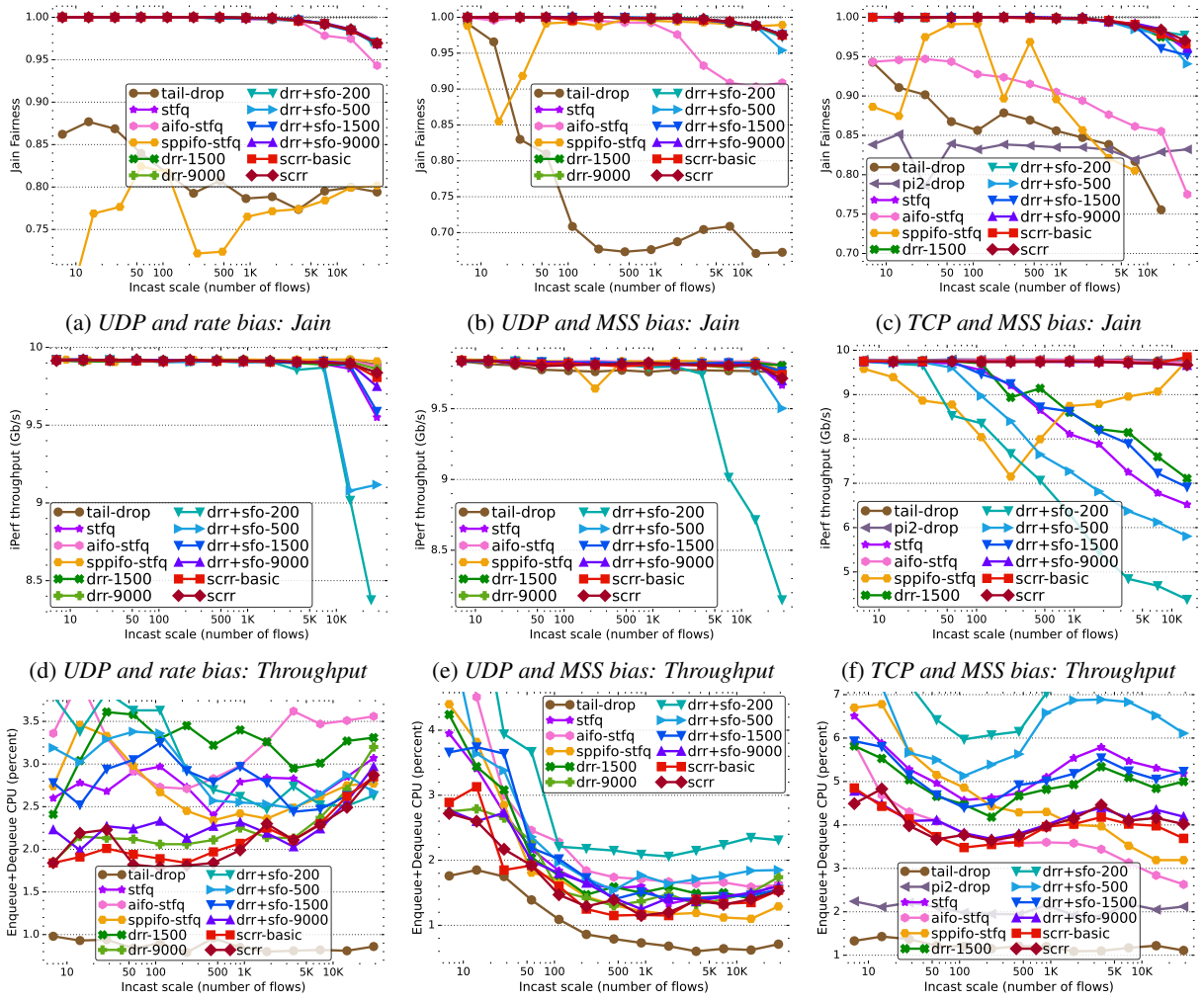
CPU utilization is a good proxy for the overall resource requirements of the packet scheduler. Some software implementations of packet schedulers in Internet gateways result in reduced network throughput due to their high CPU usage [2, 3]. It is usually tricky to estimate the impact of CPU overhead, because different packet schedulers may interact differently with the rest of the networking stack. By changing the quantum of DRR, we can artificially increase the CPU overhead of packet scheduling without changing anything else, and isolate the contribution of the packet scheduler overhead. Further, when the quanta is below 1000 B, packets are scheduled in the exact same order, so only CPU overhead changes. Experiments in Fig. 20 create an incast of heavy flows sharing a bottleneck link serviced by *DRR* and show the increase in CPU overhead due to the reduced the quantum.

Fig. 20a shows the CPU overhead of the packet scheduler and the throughput when using TCP flows. Each of the 7 sender generates 10 flows and each uses a MSS of 1000, 1500, 2000, 3000, 5000, 7000 and 8956 bytes respectively. At higher quanta, the lower per-packet processing time is balanced by the increasing packet rate, and CPU usage is constant. A small 100B quantum makes *DRR* the bottleneck and reduces the aggregate throughput to 5 Gb/s, which is half of the capacity. The CPU usage for *DRR* is less than 7%, however this puts enough pressure on the rest of the networking stack to cause such a slowdown.

Fig. 20b extends this experiment on our new testbed. Each of the 7 senders initiate 100 flows, each using an MSS of 1500, 2000, 3000, 4000, 6000, 8000 and 8956 bytes, respectively. The CPU frequency is downscaled to 1 GHz to mimic the original testbed.[5] The performance of *DRR* is similar to the experiment on the original testbed in Fig. 20a, a small 20B quantum reduces throughput to 3 Gb/s, which is a third of when quantum is 10000B. *SCRR* achieves optimal throughput, however on a slower CPU, *SCRR* could become a bottleneck and cause lower throughput. *SCRR* offers low CPU overhead because every iteration of the dequeue function is guaranteed to dequeue a packet.

Fig. 20c repeats the experiment of Fig. 20b using UDP flows, the flows are constant bit rate and the aggregate rate per sender is 2 Gb/s, for a total of 14 Gb/s. This traffic that the bottleneck can't process is dropped, the ratio of UDP packet drops (not shown) is the inverse mirror of the throughput, a lower throughput means more packets are dropped. The throughput results are broadly similar to the experiment with TCP flows, however the CPU overhead measured is much lower. Also, with the highest quantum, CPU overhead goes up. We repeated these experiments multiple times to eliminate a fluke. We do not have a good explanation for those specific results. The Linux network stack is complex, and cache performance impact and context switch impact can be

[5] In the original testbed, the CPU frequency was not downscaled.



(a) *UDP and rate bias: Jain* (b) *UDP and MSS bias: Jain* (c) *TCP and MSS bias: Jain*

(d) *UDP and rate bias: Throughput* (e) *UDP and MSS bias: Throughput* (f) *TCP and MSS bias: Throughput*

(g) *UDP and rate bias: CPU* (h) *UDP and MSS bias: CPU* (i) *TCP and MSS bias: CPU*

Figure 21: *Fairness of packet schedulers and their CPU overhead as the scale of Incast increases.*

un-intuitive. With UDP traffic, the networking stack has to process a lot more packets and ultimately drop them, causing additional CPU pressure. With TCP traffic, sub-queues may go in and out of the schedule due to the saw-tooth behavior of the TCP congestion control, increasing the amount work for the scheduler. In summary, it's not possible to determine an exact CPU overhead that will cause throughput degradation, this will depend on the traffic and the system.

Fig. 20d shows the the CPU overhead of the various kernel functions in more detail as a FlameGraph for the experiment in Fig. 20b with quantum size 500 B. The call stack for the *enqueue* function starts with the kernel polling Ethernet drivers (__napi_poll), which then passes newly received packet to the networking stack (netif_receive_skb_list_internal), those packets are routed (ip_finish_output2) and then directed to a network device (__dev_queue_xmit) and finally the *enqueue* function is called. Most of the calls to the *dequeue* function are when a packet is directed to a network device (__dev_queue_xmit), effectively after enqueuing some packets, the network stack try to make progress sending packet from that queue. Another call stack for the *dequeue* function is from a periodic timer in the network stack (net_tx_action). As expected, the *dequeue* operations consume more CPU than *enqueue*, because it needs to spin to accumulate quanta for each flow (which is a similar result to Fig. 15). During packet processing many other parts of the Linux network stack must be exercised that respectively consume CPU. With more pressure on the qdisc, the network stack processing also experiences more CPU contention and cache pressure. This explain why even if the qdisc uses a fairly low percentage of the CPU, it can be a bottleneck and lead to throughput decrease.

D.5 Fairness Under Different Biases

The crucial property of a packet scheduler is enforcing fairness and QoS. This section is an extended version of the fairness experiments already shown in §5.4. Again, to better

scale the experiments, we configured all scheduling classes with identical weights and configured various biases between flows to create potential unfairness. These experiments showcase how the scheduler achieves fairness by overcoming these biases.

The first bias we introduce is the rate at which flows arrive at the scheduler. This is a much-generalized version of the experiments in previous papers [81], where it aims to investigate how well a scheduler deals with incast traffic and a large number of misbehaving flows. The testbed is the original testbed (§5.2) and we are using the routing topology 7b. The seven senders generate UDP traffic, each with different aggregate offered rates of 1.5, 2, 3, 4, 5, 6, and 7 Gb/s, respectively. The number of UDP connections per sender varies from 1 to 4096, for a maximum of 28672 flows. Fig. 21a depicts the Jain Index measuring fairness across the flows. The *tail-drop* and *SP-PIFO* schedulers show poor fairness since it admits packets proportional to their arrival rate, favoring UDP flows with higher rates. All fair queueing schedulers, including *SCRR*, have excellent fairness, as expected, as they enforce byte-level fairness for every flow. At a very high number of UDP flows (28k), the fairness of all schedulers is impacted by hash collisions in the classifier, a well-known caveat of using hashing in flow classifiers. Most QoS classifiers have well-defined QoS classes that do not use hashing, and would not suffer from this issue.

Fig. 21d presents the aggregate throughput of the UDP flows. *DRR-200* and *DRR-500* are methods that waste CPU cycles (§2.4) and they result in decreased throughput at higher flow counts. This is because an increase in flow count increases the cache footprint of the scheduler and the probability of cache misses. Fig. 21g presents the CPU overhead on the router. In general, a higher CPU overhead is more likely to impact throughput (§2.4), however, in this experiment, the relation between the two is not obvious. We believe that the need to drop a very large number of packets distorts the CPU overhead numbers measured. A lower quantum increases CPU consumption for *DRR*, as more scheduling cycles are needed to accumulate enough quantum. All variants of *SCRR* offer lower CPU utilization amongst the schedulers.

The second bias we introduce is the packet size. Varying the packet sizes forces the schedulers to transmit more packets from flows that send smaller packets in order to satisfy the proper byte-level accounting of the schedulers. The seven sender machines generate UDP traffic, each using a different Maximum Segment Size of 2500, 3000, 3500, 5000, 6000, 8500, and 8956 bytes, respectively. Fig. 21b shows the Jain Index measuring fairness across the flows. The *tail-drop* queue yields poor fairness again. This is because the incast traffic causes the single FIFO to always be at its peak utilization. In this state, it becomes more probable for the queue to have enough space for small packets under the byte limit (18 MB, or 15 ms of traffic), which further favors flows with small packets. All other schedulers have excellent fairness, as expected. Fig. 21e shows the aggregate throughput of the UDP flows, again *DRR-200* and *DRR-500* show reduced throughput. Fig. 21h shows the CPU overhead on the router, asserting *SCRR*'s low CPU overhead.

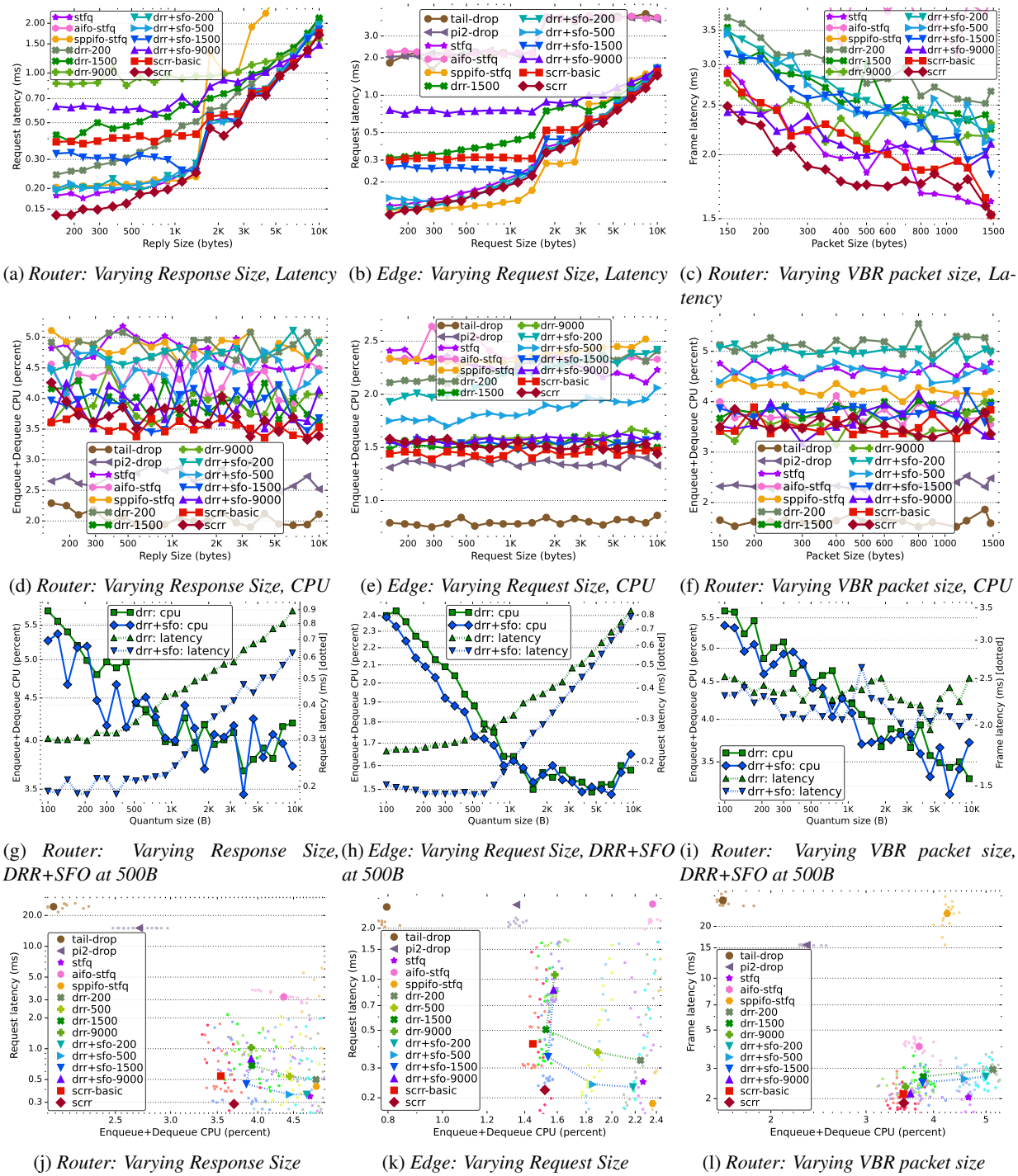
The packet size bias can also be used for TCP traffic. Fig. 21c plots the Jain Index for TCP Cubic flows. At a very high number of TCP flows (28k), some flows experience reduced performance or stalls, especially with schedulers that use the *tail-drop* queue. The TCP congestion protocol is not designed to provide byte-level fairness, therefore *tail-drop* shows only slightly better fairness compared to UDP. However, this time, flows with larger packets are favored. In the congestion avoidance phase, TCP increases the congestion window by one whole packet for each RTT, therefore flows with larger packets can increase their throughput faster. The *PI2* AQM is not designed for byte-level fairness either, which explains why it performs similarly to *tail-drop*. The same applies to *AIFO* and *SP-PIFO*. All fair schedulers provide superior fairness as they enforce byte-level fairness amongst classes. Fig. 21e shows the aggregate throughput of the TCP flows. The TCP flows are much more impacted than UDP flows, as packet drops cause the congestion control to reduce the congestion window. Fig. 21i shows the CPU overhead on the router. In this experiment, the throughput decrease can be directly correlated with the CPU overhead. As expected, the schedulers consume more CPU than the *tail-drop* and *PI2*, and as in previous experiments, the *SCRR* variants are amongst the schedulers with the lowest overhead.

D.6 Application Performance Under Cubic

We present the extended application latency results from §5.5.

Request-response Workloads. Modern networks are filled with small request-response flows [22, 62, 71]. We designed experiments showcasing how schedulers can help those small requests when they are mixed with long-lived heavy flows. Single queues often suffer from BufferBloat, where a few heavy flows can fill the queue and cause high latency for all other flows. It is fairly easy to show how schedulers provide lower latency than a single queue, however, we found it difficult to show the difference between the various schedulers on latency experienced by applications. The main issue is that those differences are mostly visible at very small request sizes, but those small requests magnify the processing overheads and inefficiencies of the packet processing pipelines, especially in the software. Additionally, the difference in scheduling delays is an order of magnitude lower than the latency added by necessary system optimizations such as interrupt coalescence, packet batching and NIC offloads. Consequently, in many cases, those differences are buried in the noise.

Our first experiment mixes small requests with long-lived heavy flows on the router in topology 7b. Two of the receiver machines generate 8 latency-sensitive flows each, the flows



(a) *Router: Varying Response Size, Latency* (b) *Edge: Varying Request Size, Latency* (c) *Router: Varying VBR packet size, Latency*

(d) *Router: Varying Response Size, CPU* (e) *Edge: Varying Request Size, CPU* (f) *Router: Varying VBR packet size, CPU*

(g) *Router: Varying Response Size, DRR+SFO at 500B* (h) *Edge: Varying Request Size, DRR+SFO at 500B* (i) *Router: Varying VBR packet size, DRR+SFO at 500B*

(j) *Router: Varying Response Size* (k) *Edge: Varying Request Size* (l) *Router: Varying VBR packet size*

Figure 22: *Impact of packet schedulers on application response times for latency sensitive workloads with Cubic transport.*

are a sequences of short back-to-back request-response with a request of 100B and varying response lengths. To highlight the difference between schedulers, we need to make them congested, because if there is no packet accumulation, packets are sent as soon as they arrive at the schedulers and they act as a FIFO. To ensure congestion, another two senders generate 32 parallel background long-lived TCP flows each. The requests are initiated on the receivers, so that the replies go from the sender to receiver and get mixed with the background traffic. For all flows, the MSS of TCP set to 1456B (MTU 1500B). LRO is disabled on the software router to minimize the unwanted latency. The latency of the TCP requests and the CPU utilization of the packet scheduler are measured under various request sizes and queue configurations.

Fig. 22a presents the latency of a selected subset of the schedulers. As the size of the replies increases, they are

split across more TCP segments, requiring more scheduling rounds to be forwarded, and we see a corresponding increase in latency. The theoretical optimal quanta for *DRR* is the MTU size, which is 1500B, however *DRR* with smaller quanta can reach the performance of *STFQ*, whereas *DRR-9000* has the worst latency among the schedulers because it always tries to dequeue 9000B worth of packets on every schedule. *SCRR* has a lower latency than *SCRR-basic* due to the combination of *Sparse Flow Optimization* (§3.3.1), *No Empty* (§3.3.3) and *Initial Advance* (§3.3.2) enhancements, which allow, most of the time, to send all the packets of the latency-sensitive flow in one attempt ahead of the background flows.

Sparse Flow Optimization implemented in DRR and SCRR (Algorithm 4) already allows previously idle sub-queues to be scheduled ahead of heavy hitters, giving an advantage to latency-sensitive flows and flows that use less than their fair share bandwidth. The replies are a burst of packets of various length, it usually includes a TCP ack and one or more TCP packets for the reply. The number of packets in the burst depend on the reply size, reply size greater than a MSS need to be broken down. It's also depend on the timing between the TCP ack and the reply. In our experiments, the TCP ack is not deferred into the reply and is sent as a separate packet. The difference in latency between schedulers is due to how they handle those consecutive packets, in particular whether the subsequent packets have to wait for a full scheduling round or not, and how long a scheduling round lasts. *Sparse Flow Optimization* allow those packet train to be scheduled ahead of backlogged flows. *Initial Advance* (§3.3.2 allow SCRR to send multiple of those packets in the same schedule. And *No Empty* (§3.3.3) allows the packets of the train that did not arrive initially to still be scheduled in the current round. Those mechanisms explain the performance of *SCRR*.

Fig. 22d shows the CPU overhead on the router. The SCRR variants are again amongst the schedulers with the lowest overhead. *SCRR* has a slightly higher aggregate CPU utilization than *SCRR-basic* mostly due to higher ratio of small packets processed (increasing the total number of packets) and the *No Empty schedule* (§3.3.3) enhancement, imposing cache pressure by re-inserting sparse flows into the schedule on their arrival.

Fig. 22g presents the performance trade-offs of *DRR* and *DRR+SFO* with 500B response sizes. A smaller quantum enables *DRR* to reduce average latency, however, it causes *DRR* to consume more CPU. In this experiment, 500B looks to be a sweet spot for quantum configuration, offering the best latency improvements.

Finally, Fig. 22j summarizes the average CPU overhead and average latency over all the request sizes. *Tail-drop* and *PI2* use a single queue, therefore, the background TCP flows quickly fill that queue and impose a full queueing delay to the response packets, or more. All fair-queueing schedulers offer much better latency performance by trading off more compute resource usage. *SCRR-basic* and *SCRR* offer the lowest CPU utilization among fair queueing schedulers. *SCRR* and *SCRR-basic* adapt to packet sizes and are therefore able to offer smaller latency than *DRR-1500* while offering a CPU usage even lower than *DRR-9000*. *SCRR-basic* has lower CPU usage than *DRR-1500* mostly because it can decide to schedule the current sub-queue without having to consult the next packet in that sub-queue. *SCRR* has the lowest average latency (0.29 ms), closely followed by *STFQ* (0.34 ms), *DRR+SFO-200* (0.35 ms), *DRR+SFO-500* (0.35 ms) and *DRR+SFO-1500* (0.45 ms).

The bursty flows are using TCP Cubic, however the latency of those flows with per-flow scheduling is mostly independent of the congestion control, and we believe other congestion controls or UDP flows should have similar performance. Our experiments with BBRv3 in §D.3 partially confirm it. With per-flow scheduling, short bursty flows never accumulate enough packets in their sub-queue at the bottleneck to cause congestion and trigger a packet drop (or ECN signal when available). With per-flow schedulers, even under heavy congestion, we did not observe any packet re-transmissions for the request-response flows. Without packet drops, the TCP Cubic never has to throttle the sending rate, and therefore the performance of those short bursty flows is entirely dictated by how they are serviced by the scheduling algorithm at the bottleneck. For similar reasons, Slow-start is not an issue for such small requests, the Initial Window of TCP is usually big enough to send the full request. With *Tail-drop*, we see a very low level of re-transmissions, and some connections that fail to connect entirely. Packets of bursty flows are added at the tail of an already full queue, where they can get dropped, and the congestion control will need to re-transmit them. This is an issue for those short request-response flow : no data is waiting after the missing packet, consequently Fast Recovery [36] usually does not happen and the retry will only happen with the slower Retransmit Time Out (RTO). This explains why with *Tail-drop* in Fig. 22j, some requests have a latency greater than the queuing delay (15 ms). With *PI2*, the number of packet dropped is much lower than *Tail-drop* [73], which helps to reduce latency to the queuing delay.

Our second experiment mixes small requests with long-lived heavy flows on the edge-switching topology 7a. This time, the request are varying in size and mixed with the background traffic to resemble applications such as POST and PUT methods in HTTP, Telemetry, Analytics, and Email which send larger chunks of data as a request. Two of the sender machines generate 8 latency-sensitive flows each, the flows are a sequences of short back-to-back request-response with a varying request length and a 50B reply. Another two senders generate 32 parallel background long-lived TCP flows each. Figs. 22b show similar results to our first experiment (fig. 22a). One big difference is *SP-PIFO*, for requests that fit in a single packet, it offers very low latency. However, *SP-PIFO* is unfair and usually has wider variance of latency when requests span multiple packets (§D.1.3). According to Fig.

22e, the CPU utilization is much more stable than our first experiment (fig. 22d) because the arrival of packets is more deterministic in this topology, it is no impacted by processing at the receiver NIC like in topology 7b. Fig. 22h shows the usual tradeoffs of *DRR* and *DRR+SFO*. Again, 500B seems to be a sweet spot for quantum configuration if latency is preferred. The default quantum of the Linux 'sch_fq' module is 3000B [33], i.e., 2× the MTU, which indicates that it was most likely optimized for low CPU utilization. Finally, 22k collects all previous results. *SP-PIFO* has the lowest average latency (0.19 ms), followed by *SCRR* (0.22 ms), *DRR+SFO-200* (0.23 ms), *DRR+SFO-500* (0.24 ms), *STFQ* (0.25 ms), *DRR-200* (0.33 ms) and *DRR+SFO-1500* (0.34 ms).

Tail-drop and *PI2* offer considerably lower latency than expected and than the first experiment. The reason is that the Linux kernel keeps in check the number of outstanding SKB packet buffers for each socket, in order to prevent queue build ups on sender machines and to reduce the latency. We measure much lower queue size in all schedulers, in *DRR* and *SCRR* the sub-queues for heavy flows are much shorter than in the first experiment by almost a factor 10. However, the mechanism to limit outstanding packet buffers reduces the latency only for *Tail-drop*, *PI2* and *AIFO*, in other schedulers it has little impact on the latency of the requests. The smaller queue size also reduce CPU utilization for all scheduler, mostly through reduction of the CPU cache footprint. This mechanism has also the potential of affecting adversely the performance of the packet schedulers, if the sub-queue of a heavy flow does not refill fast enough, it could lose some transmit opportunity, however we believe this is not happening in this experiment. There are also potential issue of CPU starvation and synchronization. We have seen unexplainable results in this topology, and we believe that the performance of *SP-PIFO* in this experiment is one of them.

Streaming Flow Performance. The previous section explores the very short packet bursts created by applications using request-reply. This section explores the longer burst patterns created by streaming applications. The traffic of video conferencing applications is usually composed of short to medium bursts that can saturate the network [79], each burst encoding a video frame. We use the isochronous support in *iperf* to generate Variable Bit Rate (VBR) traffic [8], each VBR flow is a UDP stream at 1.5 Mb/s, bursting at 30 frame per seconds. This emulates a Zoom video session at 720p [52, 61]. The traffic is Variable Bit Rate (VBR), so the length of the burst of packets varies from one frame to the next. The length of the burst of packets also depends on the packet size, it goes from an average of 333 packets (150 B) to 33 packets (1500 B) per burst. 64 streams of VBR traffic are generated on two senders, they are mixed with 128 long lived heavy flows on two other senders on the router in topology 7b.

Fig. 22c shows the average one way latency to transmit a full frame for a selected subset of the schedulers. The delay are much higher than the previous experiments, the size of the burst of packets (50k) means that multiple scheduling rounds are needed to transmit it, even at the highest quantum of DRR (9k). Conferencing often uses small packets to reduce the impact of packet losses [52, 79]. In this case, using smaller packets decreases performance, the reason is that VBR traffic is sensitive to both latency and throughput, and header and processing overheads of small packets hurt throughput of the burst. Fig. 22f shows that *SCRR* has the lowest CPU utilization amongst per-flow schedulers. Fig. 22i shows that the behavior of *DRR* is the opposite of request-response traffic, here the bigger quanta decrease latency as this traffic is more sensitive to throughput than latency. SFO is still effective and does reduce the latency of *DRR*. Fig. 22l averages the latency and CPU utilization over all packet sizes. *STFQ* maintains low latency despite its high processing overhead. *SCRR* has the lowest average latency (1.89 ms), closely followed by *STFQ* (2.04 ms), *DRR+SFO-9000* (2.14 ms), *DRR-9000* (2.34 ms) and *DRR+SFO-1500* (2.48 ms).

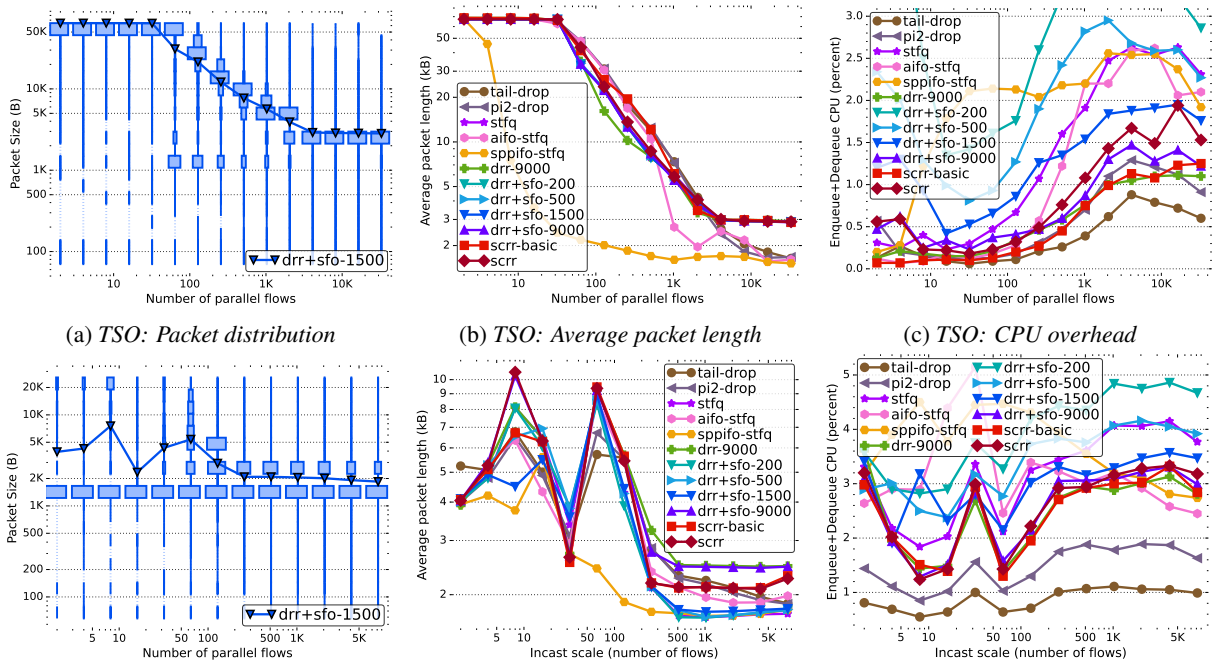
SCRR offers a unique proposition, it offers the lowest latency and lowest CPU usage for VBR frames amongst per-flow schedulers. The latency enhancement of *SCRR* enables the burst to start transmitting sooner, and the low CPU utilization allows to expedite the remaining packets of the bursts and those other packets scheduled ahead of the burst.

D.7 NIC Accelerations - TSO & LRO

We present the extended version of the NIC Accelerations results from §5.3.

The commodity acceleration functions of modern Network Interface Cards (NIC) are essential to sustain the high packet rates for high-speed Ethernet in software. TCP segmentation offloading (TSO) [11] and its software counterpart, Generic Segmentation Offload (GSO), allow the sender to defer the segmentation until the very late packet processing stages, resulting in very large packets being passed to the qdisc. Similarly, TCP Large Receive Offload (LRO) and its software counterpart, Generic Large Receive Offload (GRO), allows a NIC to aggregate received packets into larger chunks. In Linux, the current default maximum size of those packets is 64 kB and there are proposals to greatly increase this maximum [34]. This is because, a reduction in packet rate improves software performance by reducing the overhead of context switches, cache misses, and metadata handling. However, as studied earlier in this work, both TSO and LRO make packet size unpredictable for a software packet scheduler.

Fig. 23a and 23b show the impact of TSO on the packets as seen by the sender host's software scheduler in the edge switching topology (7a). This would be a typical packet distribution seen in a virtual switch deployed in a datacenter or the Cloud, or on a busy server. As the number of flows increases, the individual bandwidth share decreases and the average TSO packet size decreases from 64 kB to 1500B. Fig. 23c



(a) *TSO: Packet distribution* (b) *TSO: Average packet length* (c) *TSO: CPU overhead*

(d) *LRO: Packet distribution* (e) *LRO: Average packet length* (f) *LRO: CPU overhead*

Figure 23: *Performance of packet schedulers under TSO and LRO accelerations.*

presents the CPU overhead of the various schedulers. Smaller packets do increase CPU overhead, as expected. This is a scenario where it is difficult to find an optimal configuration for *DRR*. *DRR-1500* is configured based on the MTU and uses a fixed quantum of 1500B, but this causes too many CPU cycles to be wasted on accumulating quantum over multiple scheduling rounds to send packets that are larger than the quantum. *SCRR* lowers CPU usage by adapting to the packet sizes and has a similar CPU overhead as *DRR-9000* which uses a large quantum.

Figs. 23d and 23e show LRO packet sizes received on the router in topology 7b. This would be a typical packet distribution seen in home routers, SD-WAN gateways, access points or middleboxes. Two sender machines produce half of the workload flows, so the LRO packets are aggregated on two separate NICs. The impact of LRO is a lot less predictable than TSO, but there is still a general trend of higher number of flows resulting in smaller packet sizes. DRR with smaller quantum cannot adapt to the dynamics of the traffic, causing CPU wastage. Again, *SCRR* lowers CPU usage by adapting to the packet sizes and has a similar CPU overhead as *DRR-9000* which uses a large quantum.

The default quantum of the Linux 'sch_fq' module is 3000B [33], i.e., 2× the MTU. From the packet distributions in Fig. 23a and 23d, it's a logical choice as it's a very common packet size, and would allow to send most packets without the need to accumulate deficit for LRO and TSO with high number of flows.

Everything Matters in Programmable Packet Scheduling

Albert Gran Alcoz [1], Balázs Vass [2], Pooria Namyar [3], Behnaz Arzani [4],
Gábor Rétvári [2], Laurent Vanbever [1]

[1] *ETH Zürich,* [2] *BME-TMIT,* [3] *USC,* [4] *Microsoft Research*

Abstract

Operators can deploy any scheduler they desire on existing switches through programmable packet schedulers: they tag packets with ranks (which indicate their priority) and schedule them in the order of these ranks. The ideal programmable scheduler is the Push-In First-Out (PIFO) queue, which schedules packets in a perfectly sorted order by "pushing" packets into any position of the queue based on their ranks. However, it is hard to implement PIFO queues in hardware due to their need to sort packets at line rate (based on their ranks).

Recent proposals approximate PIFO behaviors on existing data-planes. While promising, they fail to *simultaneously* capture both of the *necessary* behaviors of PIFO queues: their *scheduling behavior* and *admission control*. We introduce PACKS, an approximate PIFO scheduler that addresses this problem. PACKS runs on top of a set of priority queues and uses packet-rank information and queue-occupancy levels during enqueue to determine whether to admit each incoming packet and to which queue it should be mapped.

We fully implement PACKS in P4 and evaluate it on real workloads. We show that PACKS better-approximates PIFO than state-of-the-art approaches. Specifically, PACKS reduces the rank inversions by up to 7× and 15× with respect to SP-PIFO and AIFO, and the number of packet drops by up to 60% compared to SP-PIFO. Under pFabric ranks, PACKS reduces the mean FCT across small flows by up to 33% and 2.6×, compared to SP-PIFO and AIFO. We also show that PACKS runs at line rate on existing hardware (Intel Tofino).

1 Introduction

Packet scheduling is a classical problem in networking — it defines the time and order at which a buffer drains packets to optimize a given performance metric. Researchers have proposed many scheduling algorithms but most of them have never been deployed in production, due to the cost and time required to implement them on new ASIC designs [4].

Programmable schedulers solve this limitation: they build an abstraction that can represent all possible scheduling algorithms [22, 31–33]. The idea is that, if we find such an abstraction and we can implement it in hardware, then we can run any algorithm on top without need for new ASICs.



Figure 1: PACKS navigates the space between SP-PIFO [6] and AIFO [37], optimizing for both rank ordering and drops.

The first attempt is the "PIFO" abstraction, which relies on the observation that we can split most scheduling functions in: a ranking algorithm that indicates the priority with which each packet should be scheduled, and a queuing structure that can schedule packets in the order of these ranks. Push-In First Out (PIFO) queues serve as a natural candidate since they sort arbitrary packet sequences based on the packets' ranks at line rate — hence the name of the abstraction [32].

PIFO queues "push" packets into arbitrary positions in the queue based on their ranks and serve them from their head. This behavior allows them to satisfy the requirements for programmable packet scheduling: (i) they always *admit* packets with the lowest ranks; and (ii) they *schedule* packets in perfect order of rank. For example, PIFO queues can "push" incoming low-rank packets before higher-rank packets that are already in the queue, even dropping the higher-rank packets (if needed) to accommodate the newly arrived low-rank ones.

However, it is hard to implement PIFO queues in hardware because they need to sort packets at line rate (even after they have been enqueued); and they may have to drop high-rank packets after they have been enqueued (e.g., if a low-rank packet arrives). Recent works *approximate* PIFO's behaviors to provide implementations that can run on existing programmable data planes [6, 16, 28, 35, 37]. But these works *only* approximate *one* of the two key PIFO behaviors (Fig. 1).

For example, SP-PIFO [6], QCluster [35], AFQ [27], PCQ [28], and Gearbox [16], only approximate PIFO's *scheduling behavior*. They map incoming packets to priority queues to minimize the rank inversions at the output. How-

ever, they do not actively control packet drops, which they leave as a byproduct effect of the schedulers' design. As such, even though these schedulers can support a broad variety of scheduling algorithms, their behavior can have a negative impact for loss-sensitive applications (cf. §2).

AIFO [37] only approximates PIFO's *admission behavior*: it executes a rank-aware admission-control policy on top of a FIFO queue that drops incoming packets imitating a PIFO queue. But because it runs on a single FIFO queue, AIFO cannot prioritize packets based on their ranks, which limits the scheduling algorithms that it can accurately approximate.

Our work We propose PACKS, a programmable PACKet Scheduler that approximates *both* the admission and scheduling behaviors of a PIFO queue on programmable hardware. PACKS runs on top of a set of strict-priority queues and combines an admission-control mechanism with a queue-mapping mechanism. Since priority queues cannot drop nor modify the order of enqueued packets, PACKS emulates the behaviors that a PIFO queue follows and executes them at enqueue.

Key insights PACKS derives its admission and queuing decisions from two key sources: the rank distribution of the last packets received (monitored via a sliding window) and the real-time buffer occupancy of each queue. PACKS integrates this data into a quantile-based admission and queue-mapping process that prioritizes packets with lowest expected ranks.

PACKS's *rank-aware* approach allows it to minimize rank inversions and outperforms existing queue-mappers that assume no prior rank knowledge and rely on per-packet heuristics. PACKS's *queue-occupancy-aware* approach ensures efficient resource utilization and reduces packet drops.

Evaluation We implement PACKS in P4 and evaluate it on real workloads and in hardware. Our results under mixed flow scenarios across various loads show that, PACKS consistently outperforms in approximating PIFO's admission behavior and reduces the rank inversions by up to 7× and 15× with respect to SP-PIFO, and AIFO, and the number of packet drops with respect to SP-PIFO by up to 60%. Under pFabric ranks, PACKS reduces the average FCT across small flows by up to 33% and 2.6× with respect to SP-PIFO and AIFO.

Contributions Our main contributions are:

- PACKS, a programmable scheduler that emulates PIFO queues on top of a set of strict-priority queues (§3).
- An admission-control algorithm and a queue-mapping technique that approximate all PIFO behaviors (§4).
- A performance analysis of PACKS on MetaOpt [24] to study its performance gaps and adversarial inputs (§4.5).
- An implementation[1] of PACKS in Java and P4 (§5).
- An evaluation showing PACKS's effectiveness in approximating PIFO using simulations and hardware (§6).

[1] Available at https://github.com/nsg-ethz/packs

2 Background

We first describe SP-PIFO [6] (§2.1) and AIFO [37] (§2.2), two programmable packet schedulers that represent how prior works approximate PIFO's scheduling and admission control behavior, respectively (Fig. 1). We then motivate PACKS based on where these schedulers fall short (§2.3).

2.1 SP-PIFO

SP-PIFO [6] approximates PIFO's *scheduling behavior* (i.e., forwarding the earliest-arrived lowest-rank packet first) on a set of strict-priority queues. It adapts the mapping between packet ranks and priority queues dynamically to minimize the number of rank inversions (i.e., the number of times a higher-rank packet is scheduled before a lower-rank one).

Mapping SP-PIFO maps incoming packets to queues based on the queue *bounds*, which define the lowest rank that the scheduler can admit into each queue. Whenever SP-PIFO receives a packet, it scans the queue bounds from lowest to highest priority, and maps the packet to the first queue with a bound lower or equal to the packet rank.

Adaptation SP-PIFO uses two mechanisms to adapt queue bounds dynamically: a *push-up* stage where it pushes future low-rank (i.e., high-priority) packets to higher-priority queues; and a *push-down* stage where it pushes future high-rank (i.e., low-priority) packets to lower-priority queues. The push-up stage occurs whenever a packet is admitted into a queue. Then, SP-PIFO updates the queue's bound to the rank of the new packet. In the push-down stage, SP-PIFO decreases the queue bounds of *all* queues when it detects a rank inversion in the highest priority queue. With these two mechanisms, SP-PIFO spreads packet ranks across queues, reduces rank inversions, and approximates PIFO's scheduling behavior.

2.2 AIFO

AIFO [37] approximates PIFO's *admission behavior* (i.e., only admitting the earliest-arrived lowest-rank packets) on a FIFO queue. It maintains a sliding window of the most recent ranks and it decides whether to admit each incoming packet based on the packet's rank and the buffer-occupancy level.

Admission AIFO uses the distance of the packet rank to the rank of the packets already in the queue and the time-discrepancy between the incoming and outgoing rate of the FIFO queue to admit packets. It increases the probability of dropping a packet as the distance between it's rank and that of the recently-admitted packets increases; and it increases the probability of dropping packets as the space available in the FIFO queue decreases (the reduction of space in the queue indicates the incoming rate is higher than the outgoing rate).

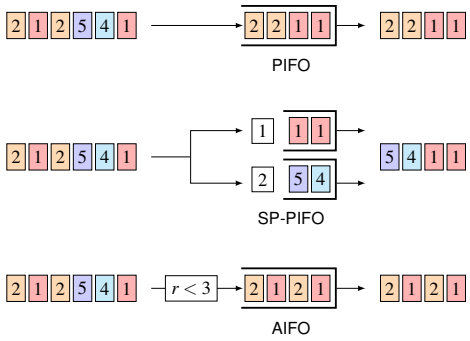
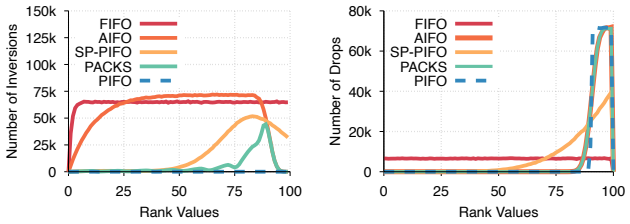


Figure 2: SP-PIFO and AIFO cannot fully approximate PIFO.



(a) Scheduling inversions (b) Packet drops

Figure 3: Scheduling performance, uniform rank distribution.

2.3 Limitations

We analyze the limitations of existing schedulers and motivate the need for PACKS with an example and a simple experiment.

Example Fig. 2 shows how PIFO, SP-PIFO and AIFO serve the packet sequence 2 1 2 5 4 1 (first packet on the right). All schedulers have capacity for 4 packets. SP-PIFO has two priority queues of two packets each, and fixed bounds with values of 1 and 2 for the highest- (resp. lowest-) priority queue. AIFO has a fixed admission control that admits ranks $r < 3$.

PIFO "pushes" the first four packets into the queue according to their rank order: 5 4 2 1. When the fifth packet arrives (1), PIFO "pushes" it into the queue between packets with ranks 1 and 2 and drops the highest-rank packet in the queue (5). When the last packet arrives (2), PIFO "pushes" it between packets of rank 2 and 4 and drops packet 4. PIFO's outgoing sequence is therefore 2 2 1 1.

SP-PIFO maps packets 1 1 to the highest-priority queue, and packets 2 2 5 4 to the lowest-priority queue (c.f., §2.1). Since the lowest-priority queue only has room for two packets, it drops the last packets to arrive (2 2). The output sequence is 5 4 1 1, which has sorted ranks (it approximates PIFO's scheduling), but does not contain the packets with rank 2 that PIFO accepted (it fails to approximate PIFO's admission).

AIFO admits the packets with rank $r < 3$, same as PIFO. However, since it runs on top of a FIFO queue, it does not prioritize any packet, which results in an output sequence not sorted by rank (i.e., 2 1 2 1 instead of 2 2 1 1).

PACKS preemptively drops packets 4 and 5, anticipating the arrival of packets with rank 2. It then sorts admitted packets across queues, achieving the optimal output (cf. Fig. 5).

Experiment These limitations generalize across ranks. We implement SP-PIFO, AIFO, PACKS and FIFO in Netbench [2, 19], and schedule a stream of packets over a bottleneck link where the ranks are distributed uniformly across $[0, 100]$ (details in §6). We measure the priority inversions generated by each rank and the number of packet drops per rank.

PIFO never causes inversions and schedules packets in perfect order (Fig. 3a). SP-PIFO approximates this behavior, especially for lower-rank packets: it maps packets with lower ranks to higher-priority queues. AIFO and FIFO generate a high number of inversions across most ranks because they run on a single queue and cannot prioritize lower-rank packets.

PIFO only drops packets with the highest ranks and has the best performance (Fig. 3b) since it prioritizes low-rank packets. AIFO closely approximates PIFO's behavior [2] by proactively dropping the highest-rank packets. SP-PIFO leaves drops as a by-product effect of its design (it just drops higher-rank packets more often because they are mapped to lower-priority queues that drain less frequently) and performs poorly. FIFO drops packets across all ranks, due to its tail-drop policy (which is rank agnostic) and has the worst performance.

PACKS's behavior is the closest to PIFO in *both* scheduling inversions and packet drops because it combines the best of both worlds: an admission-control scheme, similar to AIFO, and a queue-mapping scheme, similar to SP-PIFO.

The inefficiencies of existing works in approximating PIFO behaviors ultimately lead to performance degradation. Not exempting low-priority packets from occupying buffer space when high-priority ones are present, or not sorting packets by rank, results in increased latency, reduced bandwidth for priority applications, and longer flow completion times (§6.2).

3 Overview

We now provide an overview of how PACKS approximates the behavior of a PIFO queue on existing hardware. PACKS runs on top of a set of strict-priority queues, and incorporates: (i) an *admission-control* mechanism that decides which packets to admit, and (ii) a *queue mapper* that decides how to map admitted packets to the different priority queues (see Fig. 4).

PACKS uses this setup to approximate two PIFO behaviors: it *admits* packets with the lowest ranks; and *schedules* packets in order of their rank. What enables PIFO to achieve these behaviors is that it can map packets to any position in the queue, and it can drop packets (based on their ranks) even after it has admitted them into the queue. Unfortunately, we do not have this functionality (by default) on existing hardware.

[2] Note that the curve for AIFO and PACKS significantly overlap.

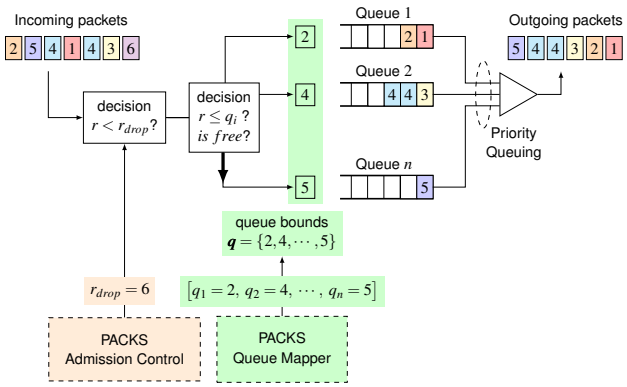


Figure 4: Overview of PACKS data-plane pipeline.

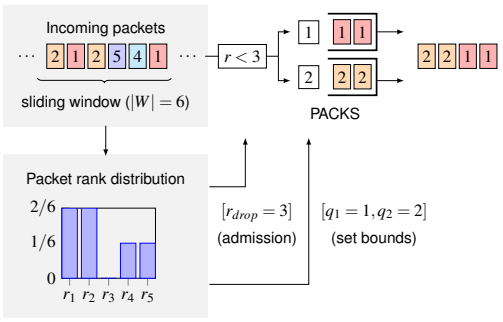


Figure 5: PACKS closely approximates PIFO's behavior.

PACKS approximates these behaviors by estimating the distribution of packets expected to arrive during a given scheduling interval. It then uses this information to compute the admission and scheduling decisions that a PIFO queue would follow and it executes them as each packet arrives. Based on the estimated rank distribution, PACKS identifies the set of lowest-ranked packets that fit into the available buffer space and proactively drops any arriving packets with a higher rank. PACKS also uses the estimated distribution to compute how to best map admitted packets into each priority queue in order to approximate the correct rank order at the output.

Rank-distribution estimation: PACKS uses a sliding window to dynamically monitor the distribution of the ranks of recently arrived packets — it considers this estimate of the rank distribution to be the best possible up-to-date estimate.

Admission control PACKS uses the distribution it estimates (see above) to decide which packets it should admit into the queue. Intuitively, PACKS should only admit the packets with the lowest ranks that can fit in the available buffer space (to mimic PIFO). Whenever a packet arrives, PACKS measures the available buffer space (as a percentage of the total buffer space) and computes a rank r_{drop} that represents a threshold such that all packets with rank $r \geq r_{drop}$ should be dropped. This rank is the lowest rank for which the quantile of the rank distribution exceeds the percentage of the remaining buffer space. This policy ensures that PACKS only admits the *lowest-rank* packets that it *expects* to arrive and fit in the available buffer space, which emulates PIFO's admission behavior.

Queue mapping PACKS then uses its estimate of the rank distribution to find the best mapping of expected packets to priority queues, to maximize the rank order at the output of the scheduler. Intuitively, the best mapping assigns packets with lower-ranks to the higher-priority queues (to prioritize low-rank packets) and minimizes the number of different-rank packets assigned to the same queue (to reduce the probability of higher-rank packets arriving before lower-rank ones, thereby generating a rank inversion in the output sequence).

PACKS defines a set of rank values $\boldsymbol{q} = (q_1, ..., q_n)$ that drive how it maps packets to priority queues (in the same way that r_{drop} drives admission control). The queue bound q_i for each queue describes the highest rank that the scheduler can admit to the queue such that these packets (i.e. those with rank $r < q_i$) are the lowest rank packets that fit in the available queue space. PACKS scans queue bounds top-down (i.e. from highest- to lowest-priority) and maps each incoming packet to the first queue where the packet rank is lower or equal to the queue bound — in this way it maps the low-rank packets to the high-priority queues: PACKS prioritizes *expected* packets of low rank over higher-rank ones (similar to PIFO).

Example Fig. 5 shows how PACKS schedules the sequence 2 1 2 5 4 1. We assume the sequence repeats, and configure PACKS with two priority queues of two packets each and a sliding window of size $|W| = 6$. After receiving the 6-th packet, PACKS has estimated the rank distribution, where the probability of receiving a packet of ranks 1 to 5 are $p(1) = 2/6$, $p(2) = 2/6$, $p(3) = 0$, $p(4) = 1/6$, $p(5) = 1/6$. Given the available buffer space (i.e., 4 packets), and based on the monitored rank distribution, PACKS sets r_{drop} to 3, since the expected 4 packets with lowest rank are those with rank 1 and 2. Then, PACKS sets q_i based on the available buffer space at each queue (i.e., 2 packets each). As such, it sets $q_1 = 1$ to map the two expected packets with lowest rank to the highest-priority queue, and $q_2 = 2$, to map the two expected admitted packets with highest rank to the lowest-priority queue. As a result, the output sequence of PACKS is 2 2 1 1, the exact same one as in the PIFO queue (see Fig. 2).

4 PACKS design

We now describe the theoretical basis supporting the design of PACKS. First, we frame the problem and introduce the design space (§4.1). Second, we provide the high-level intuition behind PACKS's design by studying the case in which it schedules a batch of packets (§4.2). Third, we generalize the algorithm to the online setup (§4.3). Finally, we formalize the PACKS's algorithm (§4.4), and analyze it both theoretically and with MetaOpt [24], an heuristic analyzer (§4.5).

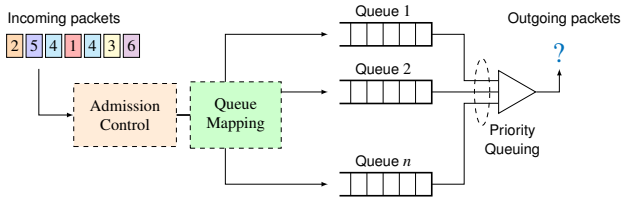


Figure 6: PACKS's design space.

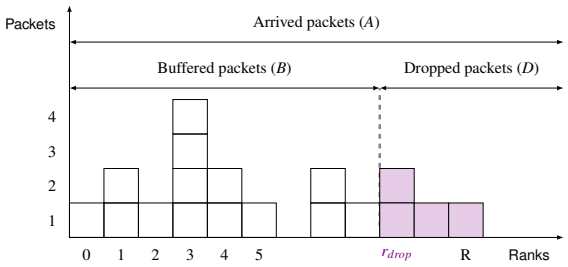


Figure 7: Admission control for a rank distribution, $\mathcal{W}$.

4.1 Design space

Let us consider the scheduling design space in Fig. 6, which represents the available resources in existing data planes [1,3]. Packets arriving at the scheduler are already tagged with ranks, either specified by the end hosts or at prior stages of the switch. The scheduler is composed by a set of strict-priority queues of fixed sizes, an *admission-control* mechanism that decides which packets to admit, and a *queue mapper* that decides how to map admitted packets to the priority queues. [3] After a packet is mapped to a queue, it is enqueued *only* if the queue has sufficient buffer space; otherwise, the packet is dropped. The scheduler continuously drains queues in decreasing order of priority, scheduling packets from low-priority queues only when higher-priority queues are empty, and schedules packets within each priority queue in a first-in first-out fashion.

Problem *How can we best approximate the behavior of a PIFO queue on top of the PACKS abstraction in Fig. 6?* The PACKS abstraction only allows for two design decisions: an admission-control and a queue-mapping algorithm. Our objective is to design such two mechanisms in a way that their overall behavior approximates the one of a PIFO queue. This is, an admission-control mechanism that (ideally) admits the earliest-arrived lowest-rank packets, and a queue-mapping algorithm that (ideally) prioritizes packets with lower rank.

4.2 High-level intuition

We introduce the high-level intuition behind PACKS's design by analyzing the case in which a PIFO queue schedules a batch of A packets. We assume, for now, that all packets have the same size, and that the PIFO queue has a capacity of B packets. For each incoming packet, the PIFO queue decides whether to admit or drop the packet. Only after processing *all* the packets, the PIFO queue schedules the admitted packets.

Approximating PIFO's admission In this setup, the PIFO queue admits the B (earliest-arrived) lowest-rank packets to the buffer, dropping the rest. Considering the rank distribution of the packets in the batch, $\mathcal{W}$, the admitted packets are the first B packets that we find when reading the distribution from left to right (see Fig. 7). As such, we can define a rank r_{drop}, such that all packets with rank equal or higher than r_{drop} are dropped by PIFO. Formally, computing r_{drop} is finding the highest rank in the distribution, for which the quantile of the distribution is below the fraction of the available buffer, B/A:

$$\begin{aligned} &\text{maximize } r_{drop}, \text{ where } 0 \leq r_{drop} \leq R, \\ &s.t., \ \mathcal{W}.quantile(r_{drop}-1) \leq B/A \end{aligned} \tag{1}$$

Once we know r_{drop}, approximating PIFO's admission on top of the PACKS abstraction is straightforward: we just have to configure PACKS's admission-control to drop all incoming packets from the batch with ranks higher or equal than r_{drop}.

So far, our model assumes that PIFO treats all packets with the same rank equally (i.e., either admitting or dropping them). In practice, however, since the PIFO queue has limited size, PIFO may only admit the earliest-arrived subset of them. To support this behavior, we extend the model by defining a time, t_{drop}, above which PIFO drops all the packets of the highest-admitted rank (i.e., $r_{drop}-1$). We can approximate this behavior on the PACKS abstraction by configuring its admission-control mechanism to drop packets based on both, r_{drop} and t_{drop}. Specifically, PACKS should drop packets if $r \geq r_{drop}$ or if $\{r = r_{drop}-1$ and $t \geq t_{drop}\}$.

Approximating PIFO's scheduling Once PIFO has decided whether to admit or drop each packet, it schedules the B buffered packets in a earliest-arrived, lowest-rank-first fashion. This requires packet sorting at line rate. We can approximate this behavior in the PACKS abstraction using priority queues.

For each priority queue, i, we define a rank, q_i, such that we only admit to the queue packets with rank lower or equal than q_i (c.f., Fig. 8). We call these ranks *queue bounds*. Formally, we let $\boldsymbol{q} = (q_1, \ldots, q_n) \in \mathbb{Z}^n$ be the set of bounds for queues 1 to n. We define a mapping strategy that uses queue bounds to map packets to their *highest-possible* priority queue, based on their rank. For each incoming packet with rank r, we scan queues top-down (i.e., from highest- to lowest-priority) and map the packet to the first queue, i, that satisfies $r \leq q_i$.[4] With this definition, we convert the problem of sorting packets at line rate based on their ranks to the problem of finding the optimal queue bounds that maximize rank order at the output of the scheduler. We define a loss function $\mathcal{U}_S : \mathrm{R}^n \times$

[3] Some devices allow extra functionalities such as flexible priority-queue configuration, round-robin scheduling, or buffer management. We use Fig. 6's abstraction for generality and to guarantee line-rate processing for *all* packets.

[4] Note that PACKS scans queues top-down, while SP-PIFO bottom-up [6].

$R_{\geq 0} \rightarrow R_{\geq 0}$, which stands for *scheduling unpifoness*, such that $\mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r)$ quantifies the approximation error of scheduling a packet with rank r based on queue bounds $\boldsymbol{q}$ compared to an ideal PIFO queue. Intuitively, it estimates the probability that a packet with rank r is scheduled after a packet with higher rank, r'. In the PIFO queue, $\mathcal{U}_{\mathcal{S}} = 0$, since packets are scheduled in perfect order. Thus, in the PACKS abstraction, a lower $\mathcal{U}_{\mathcal{S}}$ leads to a better approximation.

Our goal is to find the optimal queue bounds, $\boldsymbol{q}^*_{\mathcal{S}}$, that minimize $\mathcal{U}_{\mathcal{S}}$. Let Q be the space of all valid queue-bound vectors and $\mathcal{W}$ the distribution of packet ranks. Then, $\boldsymbol{q}^*_{\mathcal{S}}$ are:

$$\boldsymbol{q}^*_{\mathcal{S}} = \arg\min_{\boldsymbol{q} \in Q} \mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r) \tag{2}$$

Given that queue bounds are *fixed* during the enqueue process, scheduling errors cannot occur between ranks mapped to different priority queues. Thus, we can compute the total scheduling unpifoness as the sum of the individual losses at each priority queue. Letting $\mathcal{U}_{\mathcal{S}}(q_i)$ be the loss function corresponding to the queue with bound q_i, this is:

$$\mathcal{U}_{\mathcal{S}}(\boldsymbol{q}, r) = \sum_{1 \leq i \leq n} \mathcal{U}_{\mathcal{S}}(q_i) \tag{3}$$

Finally, letting $p_{\mathcal{W}}(r)$ and $p_{\mathcal{W}}(r')$ be the probability of ranks r and r', respectively, both mapped to the queue i, we can define the scheduling unpifoness of the queue as:

$$\mathcal{U}_{\mathcal{S}}(q_i) = \sum_{\substack{q_{i-1} < r \leq q_i \\ r < r' \leq q_i}} p_{\mathcal{W}}(r) \cdot p_{\mathcal{W}}(r') \tag{4}$$

With this formulation, given that we know the exact rank distribution, $\mathcal{W}$, we can easily compute the optimal queue bounds, $\boldsymbol{q}^*_{\mathcal{S}}$. For instance, [34] proposes a modified version of the Bellman-Ford algorithm that does so in polynomial time.

To provide a high-level intuition about the optimal queue bounds, we derive an upper-bound of $\mathcal{U}_{\mathcal{S}}(q_i)$ by setting $p_{\mathcal{W}}(r') = 1$. In doing so, we assume the worst case scenario in which, for each rank r, there is always a higher-rank packet, r' in the queue that can produce a scheduling error. As such:

$$\begin{aligned} \hat{\mathcal{U}}_{\mathcal{S}}(q_i) &= \sum_{q_{i-1} < r \leq q_i} p_{\mathcal{W}}(r) \\ &= \mathcal{W}.quantile(q_i) - \mathcal{W}.quantile(q_{i-1}) \end{aligned} \tag{5}$$

We can see how the optimal bounds are those that minimize the quantiles of the rank distribution for the set of ranks mapped to each priority queue. In other words, the optimal bounds are those that result in the least amount of different-rank packets mapped to each queue (i.e., those that minimize the colored area within each priority queue in Fig. 8).

Since we have to map all the admitted ranks, $0 \leq r < r_{drop}$, to some queue, removing a rank from a queue implies adding it to the adjacent queue. Thus, any reduction of unpifoness in a queue, increases the unpifoness of the adjacent queue.

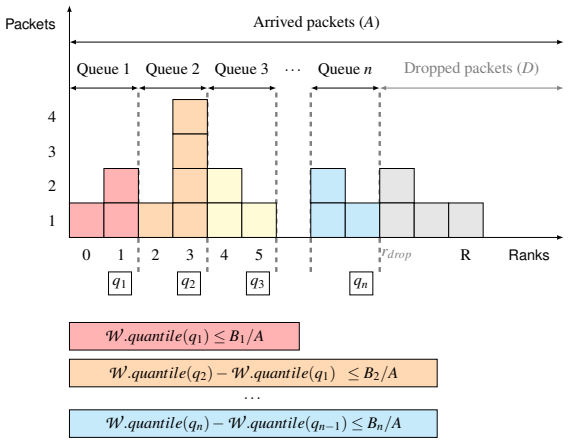


Figure 8: Queue mapping for a rank distribution, $\mathcal{W}$.

Therefore, we can only perform such an optimization step as long as there is a queue that can absorb the cost of taking in more ranks without becoming a new, greater maximum-cost queue. The optimum is achieved when the estimated scheduling unpifoness in each priority queue is balanced out.

PACKS's collateral drops Unlike in PIFO, the admission-control in the PACKS abstraction (i.e., drop if $r \geq r_{drop}$) is not its only source of packet drops. Indeed, an admitted packet can still be dropped by the priority-queue's enqueue mechanism if the available buffer space in the selected queue is not sufficient to accommodate the packet. As such, in order for the PACKS abstraction to fully approximate PIFO's admission behavior, it should not only control which packets are admitted; it should also make sure that the admitted packets are not dropped at enqueue when they are mapped to the priority queues. This brings us to the third part of the PIFO-approximation problem: approximating PIFO's efficient usage of the buffer space.

In the following, we compute the optimal queue bounds that minimize the drops that occur when mapping packets to priority queues, $\boldsymbol{q}^*_{\mathcal{D}}$, and compare them to the optimal bounds that optimize rank order at the output of the scheduler, $\boldsymbol{q}^*_{\mathcal{S}}$.

Let B_i define the buffer capacity of the i-th priority queue in the PACKS abstraction. Let $\boldsymbol{B} = (B_1, \ldots, B_n) \in \mathbb{Z}^n$ describe the buffer allocation across queues, where the sum of the buffer space of each queue is the total buffer space: $\sum_{i=1}^{n} B_i = B$. Let $\boldsymbol{q} = (q_1, ..., q_n) \in \mathbb{Z}^n$ be the set of queue bounds defining the mapping strategy, where $0 \leq q_1 \leq q_2 \leq , \ldots, \leq q_n = r_{drop} - 1$. With this strategy, we can compute the number of packets mapped to the i-th priority queue, m_i, as:

$$\begin{aligned} m_1 &= [A \cdot \mathcal{W}.quantile(q_1)] \\ m_2 &= [A \cdot \mathcal{W}.quantile(q_2)] - m_1 \\ m_n &= [A \cdot \mathcal{W}.quantile(q_n)] - m_{n-1} \end{aligned} \tag{6}$$

We define a loss function $\mathcal{U}_{\mathcal{D}} : R^n \times R_{\geq 0} \rightarrow R_{\geq 0}$, which stands for *dropping unpifoness*, such that $\mathcal{U}_{\mathcal{D}}(\boldsymbol{q})$ measures the number of packets dropped when mapping packets to queues

based on queue bounds $\boldsymbol{q}$. In the PIFO queue, $\mathcal{U}_{\mathcal{D}} = 0$, since there is no queue mapping, and drops only occur at admission. In PACKS, a lower $\mathcal{U}_{\mathcal{D}}(\boldsymbol{q})$ leads to a better approximation.

Our goal is to find the optimal bounds, $\boldsymbol{q}^*_{\mathcal{D}}$, that minimize $\mathcal{U}_{\mathcal{D}}(\boldsymbol{q})$. Let $\mathcal{Q}$ be the space of valid queue-bound vectors and $\mathcal{W}$ the distribution of packet ranks, then the bounds $\boldsymbol{q}^*_{\mathcal{D}}$ are:

$$\boldsymbol{q}^*_{\mathcal{D}} = \underset{\boldsymbol{q} \in \mathcal{Q}}{\arg\min}\ \mathcal{U}_{\mathcal{D}}(\boldsymbol{q}) \tag{7}$$

Since queue bounds are fixed during the enqueue process, and packets are dropped in each queue independently of the other queues, we can compute the total unpifoness as the sum of the individual losses at each queue, $\mathcal{U}_{\mathcal{D}}(q_i)$:

$$\mathcal{U}_{\mathcal{D}}(\boldsymbol{q}) = \sum_{1 \le i \le n} \mathcal{U}_{\mathcal{D}}(q_i) \tag{8}$$

The loss at queue i, $\mathcal{U}_{\mathcal{D}}(q_i)$, is either the difference between the number of packets mapped to the queue, m_i, and the queue space, B_i, if the number of packets mapped to the queue is greater than the queue space, or 0, otherwise:

$$\mathcal{U}_{\mathcal{D}}(q_i) = \begin{cases} m_i - B_i & \text{if } m_i > B_i \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

As such, the optimal bounds $\boldsymbol{q}^*_{\mathcal{D}}$ are the ones that minimize the difference between the number of packets mapped to each queue and the buffer size of the queue. Since all packet drops contribute equally to the loss function, there may exist multiple queue-bound vectors, $\boldsymbol{q}^*_{\mathcal{D}}$, that result in an optimal number of drops. In fact, any set of queue bounds is optimal as long as the packets mapped to each queue is lower or equal than the buffer space allocated to that queue (i.e., $m_i \le B_i$):

$$\forall i : A \cdot (\mathcal{W}.quantile(q_i) - \mathcal{W}.quantile(q_{i-1})) \le B_i \tag{10}$$

Given that PACKS's admission control already ensures that the total number of packets admitted can fit within the total buffer space (i.e., $A \cdot \mathcal{W}.quantile(r_{drop} - 1) \le B$), we can guarantee that there exists at least one set of queue bounds, $\boldsymbol{q}^*_{\mathcal{D}}$, that leads to zero drops at queue-mapping time. We can find such optimal bounds by computing the ranks for which the quantile of the rank distribution stays below the allocated queue sizes. This is $\forall i$: maximize q_i s.t. the eq. 10 is satisfied.

Same as it happened in the admission-control counterpart, there may be rank distributions for which the number of packets of a certain rank exceeds the queue capacity (even when that rank is the only one mapped to the queue). In that case, we need finer granularity than the rank-level to perform the queue mapping. Same as we did for admission control, we can overcome this limitation by introducing an enqueue-time value t_i, for each priority queue, i, such that packets are only admitted to the queue if: $r \le q_i - 1$ or if $\{r = q_i \text{ and } t \le t_i\}$. Packets not admitted to the queue i are carried over to the next queue, $i+1$, which has to account them as part of its quantile.

Sorting vs. dropping Having computed the optimal queue bounds that best approximate PIFO in optimizing rank order at the output, $\boldsymbol{q}^*_{\mathcal{S}}$, and in minimizing packet drops at queue-mapping, $\boldsymbol{q}^*_{\mathcal{D}}$, we can see that they are not always the same. Indeed, $\boldsymbol{q}^*_{\mathcal{S}}$ minimizes the quantiles of the rank distribution for the ranks mapped to each priority queue, and $\boldsymbol{q}^*_{\mathcal{D}}$ minimizes the difference between these quantiles and their respective queue sizes. Thus, which queue bounds should we use?

In general, we could pick any of the two options based on e.g., which of the two behaviors we believe is more important. However, since our goal is to design a *programmable* scheduler, we select the option that *generalizes* the most. We realize that $\boldsymbol{q}^*_{\mathcal{D}}$ are not only the best bounds for minimizing packet drops at queue-mapping time, but *also* the optimal bounds for *scheduling* in case the rank distribution is not known a priori (see eq.5 and eq.10). Indeed, if the rank distribution of incoming packets is not known, the optimal queue mapping that minimizes rank reordering is the one that distributes packets across queues proportionally to the queue sizes. Thus, $\boldsymbol{q}^*_{\mathcal{D}}$ can be seen as a worst-case bound for $\boldsymbol{q}^*_{\mathcal{S}}$, leading to a good performance in both dimensions, as we show in §6. As a result, we leverage $\boldsymbol{q}^*_{\mathcal{D}}$, as the queue bounds for our design.

4.3 Online adaptation

So far, we have assumed a simplified scenario where packets arrive to the scheduler in a *batch*-basis, all packets have the same size, and we know the complete rank distribution of the batch at enqueue. In practice, however, packets arrive in a *stream*, and the scheduler needs to perform the admission and enqueue decisions *per-packet, at line rate*. In the following, we translate our high-level intuition to an algorithm design that is practical and which we can deploy to existing hardware.

Sliding window to monitor rank distribution In the online setup we do not know the rank distribution of incoming packets, $\mathcal{W}$, in advance. Instead, the best estimation that we can make is based on the rank distribution of recently-received packets. As such, same as previous approaches [34, 37], we monitor this distribution, W, using a sliding window, and use it to drive the admission and queue-mapping decisions.

Queue occupancy to estimate congestion In the online case, packets arrive in a *continuous* stream and not in a batch basis. Thus, instead of computing the quantiles over the number of packets arrived in the batch, A, we do so over the number of packets sharing the buffer in a scheduling interval, B. At the same time, while in the batch case we could assume empty queues at start, in the online case we have to consider dynamic buffers which should absorb the short-term mismatches between traffic arrival and departure rates. We do so by measuring the buffer occupancy of the queues, and using them as an estimate of their congestion levels. [5] As a result, given b, the buffer-occupancy level at a certain packet's enqueue time, we decide to admit the packet if $W.quantile(r_{drop} - 1) \le \left[\frac{1}{1-k} \cdot \frac{B-b}{B} \right]$, where k is an optional

[5] This is a common approach in queue-management [18, 25, 37]. We could also have used the *sojourn-time* of packets, as proposed by CoDel [25].

parameter to give room for burstiness. Similarly, given b_i, the buffer-occupancy level of queue i, we perform the queue-mapping process based on queue bounds, $\boldsymbol{q}$, satisfying:

$$
\begin{aligned}
q_1 &:= \max_{r_1 \in \mathbb{N}} s.t.\ W.quantile(r_1) \leq \frac{1}{1-k} \cdot \left[\frac{(B_1 - b_1)}{B} \right] \\
q_2 &:= \max_{r_2 \in \mathbb{N}} s.t.\ W.quantile(r_2) \leq \\
&\qquad \leq \frac{1}{1-k} \cdot \left[\frac{(B_1 - b_1)}{B} + \frac{(B_2 - b_2)}{B} \right] \qquad (11) \\
&\quad \ldots \\
q_n &:= \max_{r_n \in \mathbb{N}} s.t.\ W.quantile(r_n) \leq \frac{1}{1-k} \left[\frac{\sum_{j=1}^{n} (B_j - b_j)}{B} \right]
\end{aligned}
$$

Since $q_n = r_{drop} - 1$, the lowest-priority queue's mapping policy already implies the admission control at the scheduler, which simplifies the algorithm implementation (cf. alg. 1).

Minimizing collateral drops Same as in the batch case, packets of a certain rank may exceed a queue's capacity. In the batch case, we relied on t_i to map packets to a lower-priority queue if the higher-priority queues were full. In the online case, we assess queue occupancy during mapping; if the selected queue for a given packet is full, we direct the packet to the next queue with available space. This approach addresses a key limitation of queue-mappers like SP-PIFO, which excessively drop incoming packets when mapped to the same queue (e.g., during bursts of packets with the same rank or with monotonic rank increase). As such, PACKS prevents drops and ensures an efficient usage of the buffer resources. Additionally, PACKS's *top-down* scanning process ensures that PACKS preserves the scheduling order of such packet sequences, despite mapping them to different priority queues.

4.4 PACKS algorithm

Algorithm 1 PACKS

Require: An incoming packet pkt with rank r

1: **procedure** INGRESS
2: Update sliding window W with r
3: $B \leftarrow$ buffer.total $B_i \leftarrow$ buffer(q_i).total
4: $b_i \leftarrow$ buffer(q_i).used
5: **for** $Queues(i) : i = 1$ **to** n **do** ▷ Scan top-down
6: **if** $W.quantile(r) \leq \frac{1}{1-k} \cdot \left[\frac{\sum_{j=1}^{i} (B_j - b_j)}{B} \right]$ **then**
7: **if** $b_i < B_i$ **then** ▷ Queue i not full
8: $Queues(i).enqueue(pkt)$ ▷ Select queue
9: **return;**
10: $Drop(pkt)$ ▷ Drop packet

We detail the PACKS algorithm in alg 1. For each incoming packet, PACKS decides whether to admit the packet or drop it, and how to map admitted packets to priority queues.

Admission control Whenever an incoming packet arrives, PACKS performs two main operations. First, it updates the sliding window, W, with the rank of the new packet, r. Then, it measures the current buffer occupancy, b and uses it to compute the portion of the buffer space, B, that is still free: $\frac{B-b}{B}$. PACKS admits the incoming packet if the quantile of its rank for the monitored rank distribution is lower than the fraction of available buffer space: $W.quantile(r) \leq \left[\frac{1}{1-k} \cdot \frac{B-b}{B} \right]$. Note that we weight the admission condition by an optional parameter, k, to allow for some burstiness. Also, note that in alg. 1, the admission condition is *implicit* in the queue-mapping process. Indeed, the drop action in line 10, executed when the packet has *not* been mapped to the lowest-priority queue, already serves the purpose of admission control.

Queue mapping For the admitted packets, PACKS scans priority queues top-down (i.e., from highest- to lowest-priority) and maps the packet to the first queue with available space that satisfies the condition: $W.quantile(r) \leq \frac{1}{1-k} \cdot \left[\frac{\sum_{j=1}^{n} (B_j - b_j)}{B} \right]$. If a packet is not admitted to any of the queues, because its rank is too high, or because all queues are full, it is dropped.

4.5 PACKS analysis

Similarly to other networking algorithms [7, 18, 20, 25, 34, 37], PACKS uses a window-based approach instead of a per-packet heuristic. As a result, PACKS outperforms under a stable rank distribution, if the window size is large enough to capture it.

While the window-based approach generally makes PACKS less vulnerable to adversarial packet workloads (PACKS's bounds are updated more smoothly, making them harder to disrupt, cf. Fig. 15), it also represents PACKS's Achilles' heel.

We evaluated PACKS on MetaOpt [24]–a recent heuristic analysis tool–, to understand its performance gap relative to SP-PIFO, AIFO and PIFO, and to identify adversarial inputs (cf. Appendix B). We found that PACKS is robust against adversarial sequences that make SP-PIFO drop more than 60% of high-priority packets, or make AIFO delay highest-priority packets by more than 60% of the total queue size.

We also found that PACKS's adversarial inputs consist of bursts of either very high or very low rank packets, which "pollute" the monitored distribution and prevent the well functioning of the algorithm (cf. B.3). In §6.1 we study the impact of such behaviors in depth and show how PACKS can react faster to such distribution changes by using smaller window sizes and higher burstiness allowances. In these cases, PACKS relaxes its admission criteria and its behavior converges to the one of per-packet heuristics such as SP-PIFO (cf. Fig 10).

In Appendix A, we study PACKS's optimality theoretically. We prove that, for certain window and buffer-size conditions, the departure rate for all ranks in PACKS converges to that of a PIFO queue (cf. Theorem. 1). We also suggest an upper bound for the number of inversions that PACKS produces for a generic packet sequence, with respect to PIFO (cf. Claim. 1).

5 Implementation

We implemented PACKS in P4$_{16}$ for Intel Tofino 2 [1] using 439 lines of code. Our implementation uses 12 stages and the resources outlined in Table 1. For each incoming packet, PACKS: (i) monitors the distribution of recent ranks; (ii) computes the quantile of the packet's rank on this distribution; (iii) measures queue occupancies; and (iv) uses this data to decide the packet's admission, dropping, and queue mapping.

Rank-distribution monitoring We track the rank distribution of the packets received by implementing a sliding window over a set of $|W|$ registers. Each register stores the rank of one packet, and we use a circular packet counter, from 0 to $|W|-1$, to track the position of the oldest update. Upon the arrival of a new packet, we check the counter's value and update the register pointed to by the counter with the value of the new packet's rank. In our prototype, the sliding window has a size of 16 (which can be extended by using sampling [37]). It uses 4 stages and accesses 4 registers in parallel at each stage.

Quantile computation We compute the quantile of each incoming packet's rank based on the monitored distribution. Specifically, we count how many times the packet's rank is lower than a rank in the sliding window and then divide the result by the window size. We perform the count by accessing each register of the sliding window and comparing the packet's rank with the register's value using the register's stateful ALU. We output the result of each comparison into a binary metadata field, $output_j$, which is set to 1 if the packet's rank is smaller than the register value and 0 otherwise. We aggregate the output values by progressively summing pairs of them at each stage using non-stateful ALUs, requiring $\log_2 |W|$ stages. Finally, we divide the sum of the output values by the window size, $|W|$: $W.quantile(r) = (\sum_j output_j)/|W|$. We select the window size to be a power of 2, and implement the division through bit-shift operations.

Queue-occupancy monitoring We use a *ghost thread* [5], available in Tofino 2, to monitor queue occupancy levels at enqueue. Normally, this information is only available in the *egress* pipeline, since packets need to traverse the traffic manager to access it. We address this limitation by setting up a ghost thread that periodically writes the queues' occupancy levels (from the egress) to a register accessible from the ingress pipeline. The ghost thread takes two clock cycles to update the state of each queue and handles one queue per invocation. This results in 8 clock cycles to update the state of 4 queues. To scale PACKS across a larger set of queues and ports, we approximate the admission and queue-mapping conditions by considering the overall buffer occupancy instead of individual per-queue occupancies (i.e., $W.quantile(r) \leq \frac{1}{1-k} \cdot \frac{B-b}{B} \cdot \frac{i}{n}$). Alternatively, we could use traditional packet recirculation to convey queue-occupancy information to the ingress pipeline (as done in [37]). The first option sacrifices accuracy, while the second, throughput.

Admission and queue mapping After obtaining the quantile of the packet's rank based on the monitored rank distribution, $W.quantile(r)$, and the available buffer, b, we combine them to derive the admission and mapping conditions. We rewrite them as: $B \cdot (1-k) \cdot n \cdot W.quantile(r) \leq (B-b) \cdot i$. We compute the right side of the equation by using the math unit and bit-shift operations. Simultaneously, we compute the left side of the equation by picking a k value strategically such that the operation can be performed by a bit shift on the quantile. Finally, we compute the comparison between the two terms using the *minimum* operation of the math unit, and execute the corresponding drop or enqueue action based on its result.

6 Evaluation

We evaluate PACKS's performance in approximating PIFO for various rank distributions and configuration parameters (§6.1), its practicality under complex traffic workloads (§6.2), and its bandwidth allocation when deployed on hardware (§6.3).

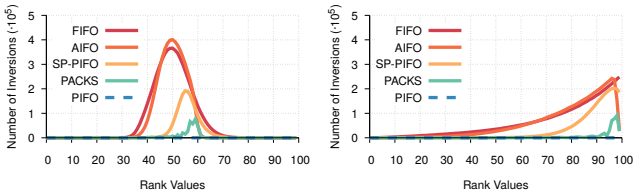
6.1 Performance analysis

First, we analyze PACKS's behavior across different rank distributions to assess its performance in approximating PIFO.

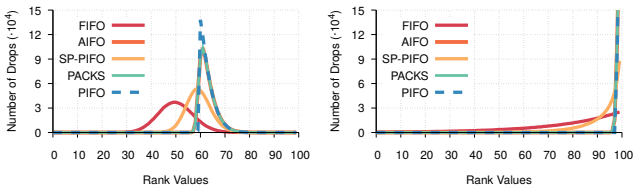
Methodology We implement PACKS, PIFO, FIFO, SP-PIFO and AIFO in Netbench [2], a packet-level simulator. We study the performance of a switch scheduling a constant bit-rate flow of 11Gbps over a 10Gbps bottleneck link for one second. We assign each packet a rank within [0-100), drawn from an exponential, Poisson, convex, or inverse-exponential distribution. We set up PACKS and SP-PIFO with 8 priority queues of 10 packets, and AIFO and FIFO with a queue of 80 packets. We set PACKS's and AIFO's window size to 1000 packets and the burstiness allowance, k, to 0. We measure the number of scheduling inversions produced by each rank (i.e., how often a packet with the rank is scheduled *before* a lower-rank packet in the queue) and the number of dropped packets per rank.

Uniform case In §2.3, we have seen how PACKS outperforms existing schemes under a uniform rank distribution (cf. Fig.3) in both the number of scheduling inversions and the drop distribution across packet ranks. Indeed, PACKS reduces the number of inversions by more than 3×, 10× and 12× with respect to SP-PIFO, AIFO and FIFO. While all schemes drop a similar percentage of packets (within ±0.03%), PACKS achieves the closest-to-PIFO drop distribution across packet ranks. PIFO only drops packets with ranks larger than 90. FIFO deviates furthest from PIFO by dropping packets across *all* ranks. It is followed by SP-PIFO, which drops packets with ranks as low as 20. AIFO and PACKS perform best, only dropping packets with ranks above 77 and 79, respectively.

Alternative distributions (inversions) We obtain similar results for non-uniform rank distributions. Fig. 9 shows the



(a) Poisson (inversions) (b) Inv. Exponential (inversions)



(c) Poisson (drops) (d) Inv. Exponential (drops)

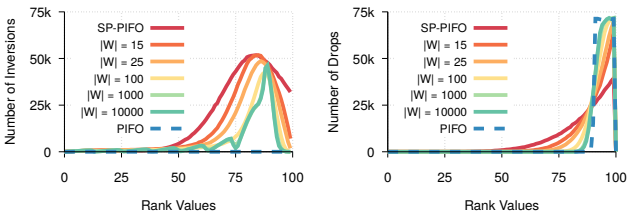
Figure 9: PIFO approximations for various rank distributions.

scheduling inversions and the packet drops across ranks for the Poisson and inverse-exponential rank distributions (we see similar results for the convex and exponential distributions). In all cases, PACKS outperforms SP-PIFO and AIFO, and gets closest to PIFO in inversions and packet drops. For the Poisson distribution, PACKS reduces the number of inversions by 5× and more than 15× and 17× compared to SP-PIFO, AIFO and FIFO, respectively. Similarly, for the inverse-exponential distribution, PACKS prevents over 7×, 14× and 15× more inversions than SP-PIFO, AIFO and FIFO, respectively. Notably, PACKS predominantly prevents inversions among lowest-ranked packets, which have higher priority.

Alternative distributions (drops) Under the Poisson distribution, all schemes drop overall a similar number of packets (within ±0.04%), being SP-PIFO the one with the highest drop rate. When considering the distribution of dropped packets across ranks, PACKS and AIFO are the schemes most closely approximating PIFO. Specifically, the lowest rank dropped by PIFO is 59, while PACKS and AIFO drop packets starting at rank 56 [6]. Conversely, SP-PIFO and FIFO show notably worse performance, dropping packets with ranks as low as 36 and 20, respectively. We observe similar results for the inverse-exponential distribution. In this case, however, while the total number of packets dropped by PACKS and AIFO is similar to the one of PIFO (+0.1% and +0.4%, respectively), SP-PIFO drops 42% more packets than them. This is due to the highly skewed nature of the distribution, which is hard for SP-PIFO to manage without admission control (cf. §4.4).

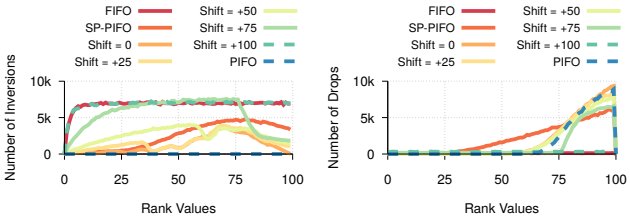
Sensitivity to window size Fig. 10 illustrates the impact of window size on PACKS's performance. Given that the rank distribution ranges from 0 to 100 and the overall buffer space is of 80 packets, PACKS performs best with window sizes above $|W| = 100$, which capture the entire distribution.

[6] Note that PACKS's and AIFO's drop distribution significantly overlap.

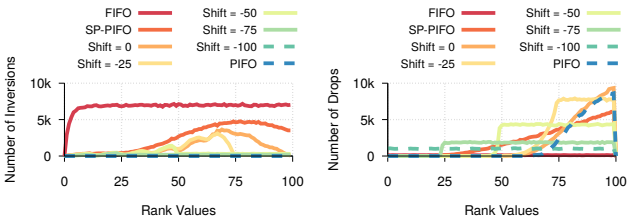


(a) Window size (inversions) (b) Window size (drops)

Figure 10: PACKS's window-size sensitivity (UDP, uniform).



(a) Positive shift (inversions) (b) Positive shift (drops)



(c) Negative shift (inversions) (d) Negative shift (drops)

Figure 11: Rank-distribution sensitivity (TCP uniform).

Since the rank distribution is stable, a higher window size consistently leads to more stable queue bounds and better performance, as indicated by the bumps in the distribution reflecting the behavior of priority queues. For example, with $|W| = 1000$, PACKS performs very close to optimal, reducing inversions by 22% compared to $|W| = 100$ and increasing the lowest-dropped rank from 69 to 78. Further increasing the window to $|W| = 10000$ doesn't improve performance so significantly, only reducing inversions by 1% and raising the lowest-dropped rank from 78 to 80, compared to $|W| = 1000$.

Window sizes below $|W| = 100$ lead to worse performance since the window cannot capture the entire distribution. Interestingly, as we reduce the window size, PACKS's behavior approaches that of SP-PIFO. Nevertheless, even with very small window sizes, PACKS still outperforms. For instance, with $|W| = 15$, which barely captures a 15% of the distribution, PACKS still produces 30% less inversions compared to SP-PIFO and starts dropping packets at rank 34 instead of 18.

Sensitivity to distribution shifts PACKS's design follows a window-based approach and, thus, relies on a predictable future packet distribution. Since real-world packet distribu-

tions are not always predictable, we now assess how PACKS performs when the monitored rank distribution differs from that of incoming packets. To do so, we modify PACKS's algorithm to consistently shift *all* ranks *in the sliding window* by a factor. This approach does not reflect a real-world scenario since, even under a drastic distribution shift in practice (e.g., a microburst), packets from the "new distribution" would arrive in a continuous stream, allowing the sliding window to adapt gradually as each packet arrives. Still, it helps us understand PACKS's performance boundaries. We run TCP flows at 80% load, with packet ranked uniformly at random from 0 to 100.

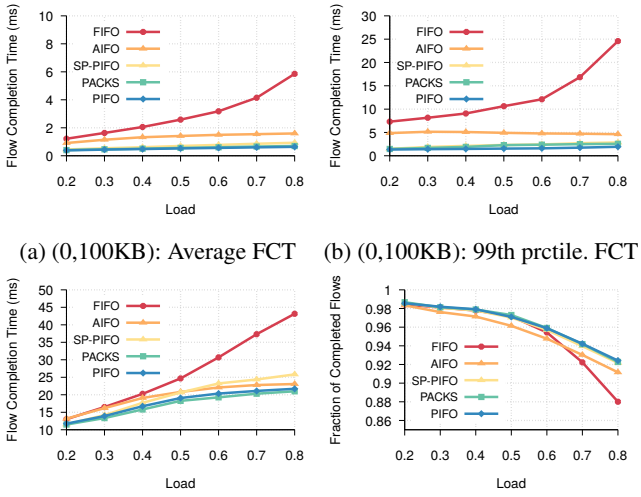
Fig. 11a and Fig. 11b show the impact of shifting the ranks of the sliding window by *positive* factors. This leads to more permissive admission and queue-mapping decisions, as if we increased the priority of incoming packets. When the shift reaches 100, all arriving packets have higher priority than the ones in the sliding window, causing PACKS to admit all packets and behave like a FIFO queue. Despite the extreme scenarios with shifts $\geq$75, PACKS exhibits significant robustness to positive distribution shifts. For instance, with a shift of +25, PACKS vastly outperforms SP-PIFO by reducing inversions by 34% and with a lowest-rank dropped of 46, as opposed to 12 in SP-PIFO. Even with a shift of +50, PACKS performs comparably to SP-PIFO in terms of total inversions while dropping 162$\times$ fewer packets below the rank of 58.

Fig. 11c and Fig. 11d show the impact of shifting the ranks of the sliding window by a *negative* factor. This is equivalent to decreasing the priority of incoming packets, which has a more detrimental impact on performance than positive shifts, and affects packet drops. Indeed, admission control drops a percentage of packets equal to the magnitude of the shift. With a -100 shift, PACKS drops all incoming packets. Similarly, a -75, -50 and -25 shift lead to dropping 75%, 50% and 25% of packets with the lowest priority, respectively. For the subset of admitted packets, PACKS maintains ideal behavior in terms of scheduling inversions. We can counteract the effect of negative distribution shifts by increasing the burstiness allowance, k, or decreasing the window size to speed up reaction time.

6.2 Performance in typical use cases

We now study PACKS's performance under two common scheduling objectives: minimizing flow completion times and enforcing fairness [6, 8, 37, 38]. These scenarios are challenging for PACKS because they involve large and non-stationary rank distributions, which are difficult to monitor accurately.

Methodology We use a leaf-spine topology with 144 servers connected through 9 leaf and 4 spine switches, and set the access and leaf-spine links to 1Gbps and 4Gbps, respectively. We generate traffic flows following the pFabric web-search workload [8]. Flow arrivals are Poisson-distributed and we adapt their starting rates for different loads. We use ECMP and draw source-destination pairs uniformly at random.



(a) (0,100KB): Average FCT (b) (0,100KB): 99th prctile. FCT

(c) [All flows]: Average FCT (d) Fraction of completed flows

Figure 12: pFabric: FCT statistics across different flow sizes.

Setup pFabric We run pFabric [8] (without starvation prevention [7]) on top of PIFO, AIFO, SP-PIFO and PACKS, and assess their efficacy in minimizing flow completion times. pFabric assigns ranks to packets based on their remaining flow sizes. As suggested in [8], we approximate pFabric's rate control using standard TCP with an RTO of 3 RTTs. We configure PACKS and SP-PIFO with 4 queues$\times$10 packets and PIFO, AIFO and FIFO with 1 queue$\times$40 packets. For PACKS and AIFO, we set $|W|$ to 20 packets, and k to 0.1.

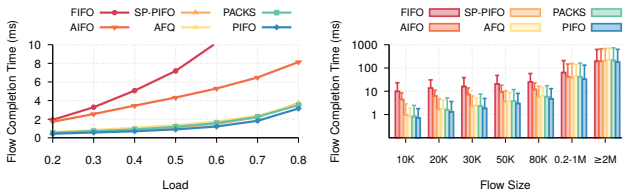
Results pFabric Fig. 12 shows the mean and 99th percentile FCT of small flows, the average FCT across all flows, and the fraction of completed flows. AIFO produces longer FCTs for small flows due to the lack of packet sorting. SP-PIFO results in longer FCTs at high loads due to queue inversions. PACKS combines the strengths of both, performing best overall.

In terms of average FCTs for small flows, PACKS achieves FCTs just 5% to 9% longer than PIFO, which is remarkable given its use of just 4 queues. In turn, PACKS outperforms SP-PIFO by 11% to 33%, AIFO by a factor of 2.25$\times$ to 2.6$\times$, and FIFO by 3.2$\times$ to 9.2$\times$ (biggest benefits under heavy loads).

At the 99th pctl., PACKS achieves FCTs 8% to 49% longer than PIFO, but remains better than SP-PIFO (from 2.2% to 12% better), AIFO (1.8$\times$ to 3.3$\times$), and FIFO (5$\times$ to 10$\times$).

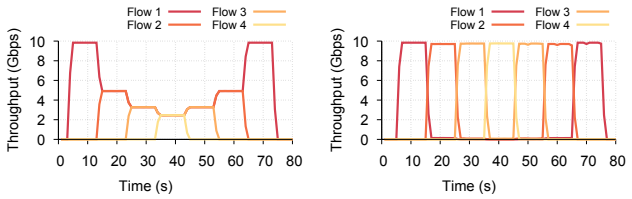
Regarding the mean FCT across all flows, PACKS achieves on-par performance with PIFO (the mean FCT of PACKS is even a bit lower–from 2% to 5%–due to long flows not completing transmission, cf. Fig. 12d). Once again, PACKS consistently outperforms SP-PIFO across all loads (with improvements ranging from 0.2% at the lowest load to 23% at the highest load), AIFO (9% to 21%) and FIFO (13% to 2$\times$).

[7] Starvation [8] is a limitation inherent to PIFO, and therefore of all its approximations. Previous works have already proposed solutions to the starvation problem (e.g., PDA [35]) which can also run on PACKS.



(a) (0,100KB): Average FCT (b) FCT breakdown for 70% load

Figure 13: Fairness: FCT statistics at different loads.



(a) FIFO Bandwidth-split (b) PACKS Bandwidth-split

Figure 14: Bandwidth allocation for increasing-priority flows.

Finally, PACKS's fraction of completed flows closely matches that of PIFO (± 0.01% to 0.2%). Moreover, PACKS achieves higher completion rates than SP-PIFO, with improvements from 0.06% to 0.2%, and AIFO (resp. 0.3% to 1.2%).

Setup fair queuing We run the Start-Time Fair Queueing rank design [17] on top of the schedulers and evaluate their performance at enforcing fairness across flows. We compare to FIFO and AFQ [27] for reference. We set the bytes-per-round of AFQ to 80 packets. We use 32 queues×10 packets in SP-schemes and 1 queue×320 packets for single-queue schemes. Same as [6,37], we generate traffic from the pFabric web-search distribution, and assess fairness by measuring the flow completion time of short flows. For PACKS and AIFO, we set the window size to 10 and the burstiness margin to 0.2.

Results fair queuing Fig. 13a depicts the average flow completion time for small flows across loads from 20% to 80%. PACKS stays within 10–24% of the ideal PIFO, and consistently outperforms FIFO, AIFO and AFQ across all loads. Specifically, PACKS reduces the average FCTs for short flows by 2.5–5.5×, 1.12×–2.4×, 9–27% with respect to FIFO, AIFO and AFQ, respectively. PACKS performs similarly to SP-PIFO (within ±6%), underperforming at lower loads, but outperforming by 6% at the highest load (80% utilization).

Fig. 13b illustrates the average and 99th percentile flow completion times across flow sizes at 70% utilization. PACKS's performance consistently stays within 17–26% of the ideal PIFO in terms of average FCT and within 15–54% for the 99th percentile across all flow sizes. It shows comparable performance to SP-PIFO (within ±10% for average FCTs and ±20% for the 99th percentile) and AFQ for average FCTs (within ±15%). However, AFQ outperforms at the 99th percentile, by up to 31%. For the smallest flows, PACKS achieves the lowest average FCT, closely trailing AFQ by 5%.

6.3 Hardware testbed

We show that PACKS performs at line rate on actual hardware by running it on the Edgecore Networks DCS810 (AS9516-32D) Intel Tofino2 Switch [1]. Same as previous works [6, 37, 38], we measure the bandwidth that PACKS allocates to different priority flows over a bottleneck link. We generate traffic between two servers, connected by a Tofino2 switch, using interfaces of 100 Gbps (sender→switch) and 10 Gbps (switch→receiver). We run four UDP flows of 20 Gbps each using MoonGen [13, 14]. We start flows sequentially (one flow at a time), in increasing order of priority with a time gap of 10 seconds between starts. We stop them sequentially in decreasing order of priority, with 10 seconds between stops.

Fig. 14 depicts the flows' bandwidth and how PACKS manages to effectively prioritize traffic from lower ranks. While the FIFO queue distributes the bandwidth uniformly across flows (failing at prioritizing traffic), PACKS successfully allocates the available bandwidth to the highest-priority flow.

7 Related work

Packet scheduling has been extensively studied for decades [8, 10–12, 17, 21, 22, 26, 27, 29]. The concept of programmable scheduling was introduced by [31, 32], which proposed the PIFO queue as an enabling abstraction. While promising, implementing PIFO queues in hardware proved challenging. Hence, a subset of follow-up works have suggested new hardware designs such as PIEO [30], BMW-Tree [36], BBQ [9], and Sifter [15]. Other works have focused on approximating PIFO behaviors on existing programmable data planes: SP-PIFO [6], QCluster [35], PCQ [28], AIFO [37], Spring [34], and Gearbox [16]. PACKS falls into the latter category.

8 Conclusions

We present PACKS, the first programmable packet scheduler that emulates PIFO queues on existing data planes in *both* rank ordering *and* packet drops. PACKS runs on top of a set of priority queues and leverages packet-priority information and queue-occupancy levels during enqueue, to schedule packets in order of priority. We show that PACKS is practical, achieves close-to-PIFO behavior, and outperforms the state-of-the-art.

Acknowledgments

We are grateful to the reviewers and our shepherd, Hyojoon Kim, for their comments. This work was partly supported by the OTKA #135606 and #135074 projects of the National Research, Development and Innovation Fund of Hungary.

References

[1] The Edgecore Networks DCS810 Switch with Tofino2. https://www.edge-core.com/productsInfo.php?cls=1&cls2=349&cls3=577&id=916, 2017.

[2] Netbench. http://github.com/ndal-eth/netbench, 2018.

[3] Broadcom StrataXGS Switch Solutions. https://www.broadcom.com/products/ethernet-connectivity/switching, 2023.

[4] Network Programmability: The Road Ahead. https://www.youtube.com/watch?v=CtxfmES4T7E, 2023.

[5] Anurag Agrawal and Changhoon Kim. Intel Tofino2: A 12.9 Tbps P4-Programmable Ethernet Switch. In *IEEE Hot Chips Symposium (HCS 32)*, 2020.

[6] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues . In *USENIX NSDI*, Santa Clara, CA, USA, 2020.

[7] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, New Delhi, India, 2011.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM*, Hong Kong, China, 2013.

[9] Nirav Atre, Hugo Sadok, and Justine Sherry. BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling. In *USENIX NSDI*, Santa Clara, CA, USA, 2024.

[10] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[11] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *ACM SIGCOMM*, Baltimore, MD, USA, 1992.

[12] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queuing Algorithm. In *ACM SIGCOMM*, New York, NY, USA, 1989.

[13] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A Scriptable High-Speed Packet Generator. In *ACM IMC*, Tokyo, Japan, 2015.

[14] Sebastian Gallenmüller, Paul Emmerich, Daniel Raumer, and Georg Carle. MoonGen: Software Packet Generation for 10 Gbit and Beyond. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[15] Peixuan Gao, Anthony Dalleggio, Jiajin Liu, Chen Peng, Yang Xu, and H. Jonathan Chao. Sifter: An Inversion-Free and Large-Capacity Programmable Packet Scheduler. In *USENIX NSDI*, Santa Clara, CA, USA, April 2024.

[16] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H. Jonathan Chao. Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing. In *USENIX NSDI*, Renton, WA, USA, 2022.

[17] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *ACM SIGCOMM*, Palo Alto, CA, USA, 1996.

[18] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM '88*, Stanford, California, USA, 1988.

[19] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond Fat-trees Without Antennae, Mirrors, and Disco-balls. In *ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[20] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *ACM SIGCOMM*, Virtual Event, USA, 2020.

[21] Paul E McKenney. Stochastic Fairness Queueing. In *IEEE INFOCOM*, 1990.

[22] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal Packet Scheduling. In *USENIX NSDI*, Santa Clara, CA, USA, 2016.

[23] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, and Srikanth Kandula. Minding The Gap Between Fast Heuristics and Their Optimal Counterparts. In *ACM HotNets*, 2022.

[24] Pooria Namyar, Behnaz Arzani, Ryan Beckett, Santiago Segarra, Himanshu Raj, Umesh Krishnaswamy, Ramesh Govindan, and Srikanth Kandula. Finding Adversarial Inputs for Heuristics using Multi-level Optimization. In *USENIX NSDI*, Santa Clara, CA, USA, 2024.

[25] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. In *ACM Queue*, New York, NY, USA, 2012.

[26] Linus E Schrage and Louis W Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. 1966.

[27] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX NSDI*, Renton, WA, USA, 2018.

[28] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling. In *USENIX NSDI*, Santa Clara, CA, USA, 2020.

[29] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *ACM SIGCOMM*, Cambridge, Massachusetts, USA, 1995.

[30] Vishal Shrivastav. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *SIGCOMM '19*, Beijing, China, 2019.

[31] Anirudh Sivaraman, Suvinay Subramanian, Anurag Agrawal, Sharad Chole, Shang-Tse Chuang, Tom Edsall, Mohammad Alizadeh, Sachin Katti, Nick McKeown, and Hari Balakrishnan. Towards Programmable Packet Scheduling. In *ACM HotNets*, Philadelphia, PA, USA, 2015.

[32] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[33] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *ACM HotNets*, College Park, MD, USA, 2013.

[34] Balázs Vass, Csaba Sarkadi, and Gábor Rétvári. Programmable Packet Scheduling With SP-PIFO: Theory, Algorithms and Evaluation. In *IEEE INFOCOM Workshops*, 2022.

[35] Tong Yang, Jizhou Li, Yikai Zhao, Kaicheng Yang, Hao Wang, Jie Jiang, Yinda Zhang, and Nicholas Zhang. QCluster: Clustering Packets for Flow Scheduling, 2020.

[36] Ruyi Yao, Zhiyu Zhang, Gaojian Fang, Peixuan Gao, Sen Liu, Yibo Fan, Yang Xu, and H. Jonathan Chao. BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree. In *ACM SIGCOMM*, New York, NY, USA, 2023.

[37] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. Programmable Packet Scheduling with a Single Queue. In *ACM SIGCOMM*, New York, NY, USA, 2021.

[38] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *USENIX NSDI*, 2021.

A Theoretical analysis of PACKS

Comparison with PIFO In the following, first, we show that, under certain conditions, the departure rate for *all* packet ranks in PACKS is the same as for a PIFO queue. Moreover, under these conditions, there is only a small difference between the sets of packets forwarded by PIFO and PACKS.

Let the set of packets forwarded (up to time t) by PIFO and PACKS be PIFO(t) and PACKS(t), respectively. Then, to measure the difference in drops between PACKS and PIFO, we define:

$$\Delta(t) = \frac{|\text{PIFO}(t) \setminus \text{PACKS}(t)| + |\text{PACKS}(t) \setminus \text{PIFO}(t)|}{|\text{PIFO}(t) + \text{PACKS}(t)|}.$$

We have $\Delta(t) \in [0,1]$, where a small value of $\Delta(t)$ indicates a small difference between PACKS and PIFO. In the following, we denote the the maximal and minimal rank probabilities with $\delta_+ := \max_i p(i)$ and $\delta_- := \min_i p(i)$.

Theorem 1 *Assume that the window size $|\mathcal{W}|$, buffer spaces $B_1, \ldots, B_n$, and the number of arrived packets, T, tend to infinity. Furthermore, assume that the maximal and minimal rank probabilities δ_+ and δ_- are bounded between two positive constants. We denote the ratio of the outgoing and incoming packet rate by v, and suppose $v < 1$ (otherwise, both PIFO and PACKS behave like a FIFO). We claim that the difference between the drops of PIFO and PACKS is at most δ_+, i.e., $\Delta(T)_{T\to\infty} \leq \delta_+$. Moreover, for each packet rank, the admission rate of PACKS is identical to the one of PIFO.*

Proof: Since the window size, $|\mathcal{W}|$, is considered very large, the empirical rank distribution in $\mathcal{W}$ tends to the real packet rank distribution. In other words, after waiting a long time, we can know the rank probabilities with high precision, that is $|p(i) - p_{\mathcal{W}}(i)| \xrightarrow[T \geq |\mathcal{W}|]{|\mathcal{W}|\to\infty} 0$. Thus, empirical quantiles, $\mathcal{W}.quantile(i)$, tend to the quantiles according to the real distribution, i.e., $\mathcal{W}.quantile(i) \to \sum_{i=1}^{r} p_i$.

Intuitively, since the buffer space B is very large, the relative queue occupancy b/B changes smoothly over time. More precisely, let $b(t)$ denote the queue occupancy after the arrival of the t^{th} packet (or, for short, 'at time t'), and let $q_n(t) = \frac{1}{1-k}\frac{B-b(t)}{B}$ denote the highest queue bound at time t. At time T, we have queue bound $q_n(T)$ as the admission bound. Let r_T be the maximum rank such that $\mathcal{W}.quantile(r_T) \leq q_n(T)$. This means that the ratio of the admitted packets is $\sum_{i=0}^{r_T} p(i)$ Thus, after the arrival of the next packet, $\mathbb{E}(b(T+1) - b(T)) = \sum_{i=0}^{r_T} p(i) - v$ (recall that, for every incoming packet, the number of drained packets is v on average). This means the following.

1. If $\sum_{i=0}^{r_T} p(i) > v$, the queue occupancy likely increases, ultimately triggering a drop in q_n and in the rate of admitted packets.
2. If $\sum_{i=0}^{r_T} p(i) < v$, the occupancy likely decreases, triggering a rise in q_n and in the rate of admitted packets.
3. Finally, in the event of $\sum_{i=0}^{r_T} p(i) = v$, the queue occupancy makes a motion very similar to the one-dimensional random walk, eventually, after a while likely triggering q_n to either drop or rise for a short time period, before bouncing back to r_T.

We note that, since the buffer spaces are considered to be very large, and the minimum rank probability δ_- is lower bounded by a positive constant, these events happen with probability 1 based on the law of large numbers. Furthermore, in case 3, $q_n(T+t) = r_T$ for any $t \geq 0$ with probability 1.

This also means that, in case 3, $\Delta(T) \xrightarrow{T\to\infty} 0$, since after a while PIFO and PACKS forward the same packets with probability 1. In cases 1 and 2, there is a single rank 'on the border' that either gets forwarded or dropped by chance both in PIFO and PACKS; thus, in these cases, $\Delta(T)_{T\to\infty} \leq \delta_+$. Note that the overall forwarding rate of this rank (and thus of all ranks) is the same for both PIFO and PACKS.

An alternative *intuitive* reasoning supporting the statement that the forwarding rates coincide for PIFO and PACKS is the following. In both cases, there are three classes of ranks: (i) small ranks that are always forwarded; (ii) large ranks that are always rejected; and (iii) a borderline rank r^* that is either forwarded or rejected by chance. Since draining is continuous both for PIFO and PACKS, the leftover bandwidth after the small-ranked packets is given to the borderline rank, r^*, as it is the only choice, again both for PIFO and PACKS. □

Next, we present an asymptotically tight upper bound for the number of inversions that PACKS produces for a packet sequence with respect to PIFO.

Claim 1 *On a sequence of S packets, given a buffer size of B, PACKS cannot cause more than $\Theta(B \cdot S)$ inversions with respect to PIFO.*

Proof: A bad sequence for PACKS: Take a sufficiently large S, with the packet ranks in the sequence being $S, S-1, \ldots, 1$. Then, PACKS will enqueue all the packets to the highest priority queue $Queues(1)$. In this setting, the behavior of PACKS basically transforms to being a FIFO using $Queues(1)$. We assume that after a brief period, when the rate of packet arrivals exceeds the departure rate, $Queues(1)$ gets full. Oversimplified, e.g., while enqueuing the first B_1 packets, none is dequeued. Then, packets are enqueued and dequeued in an alternate fashion. In this simplified example, the PIFO output rank sequence will be (first dequeued on the left): $O_P = [S-B_1, S-B_1-1, \ldots, 1, S-B_1+1, S-B_1+2, \ldots, S]$. In the meantime, the output of PACKS is the same sequence as the input was: $[S, \ldots, 1]$. We can see that PIFO forwarded

a number of $S - B_1$ packets B_1 time slots faster than PACKS. If $S \gg B_1$ and $B_1 \geq c \cdot B$ for some constant $c > 0$, then the number of inversions produced by PACKS compared to the PIFO output is $\Omega(SB)$. We can see that the same asymptotic bound hold in the more realistic scenario when some packets are dequeued in the initial phase when the queue gets full.

Upper bound if the same packets are admitted as PIFO: obviously, one packet cannot get ahead more packets than the buffer size B, hence in the output sequence of PACKS there could be not more than $O(BS)$ more inversions than in the output of PIFO. □

We note that, after a short initialization period, an ever-decreasing sequence of ranks (like in the proof of Claim 1) makes AIFO and SP-PIFO suffer similarly as PACKS in terms of the number of rank inversions.

Comparison with AIFO The next theorem states that PACKS admits the same packets as AIFO. This notable since AIFO was designed to mimic the admission behavior of PIFO.

Theorem 2 *Given the same window size, buffer size, and burstiness allowance, PACKS drops the same packets as AIFO.*

Proof: Following the notations of the AIFO paper [37], we denote the total buffer size of AIFO by C, and its queue occupancy by c. AIFO admits a packet r if $W.quantile(r) \leq \frac{1}{1-k} \cdot \frac{C-c}{C}$. Assume indirectly that there exists an $t \in \mathbb{N}$, for which the t^{th} arriving packet is enqueued in exactly one of PACKS and AIFO. We choose t as the minimum of such values. We denote the rank of this packet as r_t.

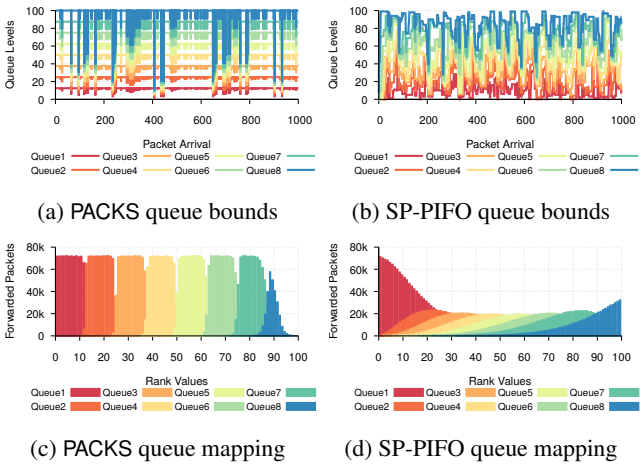
Case 1: PACKS enqueued r_t, while AIFO did not. Here we have the following inequalities explained below, yielding a contradiction:

$$W.quantile(r_t) \overset{(a)}{>} \frac{1}{1-k} \cdot \frac{C-c}{C} \overset{(b)}{=} \frac{1}{1-k} \cdot \frac{\sum_{j=1}^{n} B_j - b_j}{B} \overset{(c)}{\geq}$$

$$\overset{(c)}{\geq} W.quantile(r_t).$$

Here, we get (a) from the fact that AIFO does not enqueue r_x. For (b), we just match the notations of AIFO and PACKS. Finally, (c) holds because PACKS enqueues r_t. Combined, (a), (b), and (c) clearly yield a contradiction.

Case 2: AIFO enqueued r_t, while PACKS did not. Since AIFO enqueued r_t, we have $W.quantile(r_t) \leq \frac{1}{1-k} \cdot \frac{C-c}{C} = \frac{1}{1-k} \cdot \frac{\sum_{j=1}^{n} B_j - b_j}{B}$. Let $i \in \{1, \ldots, n\}$ be the minimal number such that $W.quantile(r_t) \leq \frac{1}{1-k} \cdot \frac{\sum_{j=i}^{n} B_j - b_j}{B}$. We know that such an i exists. Since PACKS did not enqueue r_t at all, we can deduce it did not enqueue r_t in the i^{th} queue either. This is possible only if $b_i = B_i$. If $i \geq 2$, this contradicts the minimality of i, since this means $W.quantile(r_t) \leq \frac{1}{1-k} \cdot \frac{\sum_{j=i-1}^{n} B_j - b_j}{B}$. If $i = 1$, and $n \geq 2$, then PACKS will enqueue r_t to the first queue having free space; note that such a queue exists, since before the arrival of r_t, AIFO had spare buffer space. Finally, The case of $i = 1$, and $n = 1$ also yields contradiction, since then, the AIFO would not have enqueued r_t either. The proof follows. □



(a) PACKS queue bounds (b) SP-PIFO queue bounds

(c) PACKS queue mapping (d) SP-PIFO queue mapping

Figure 15: Queue-bounds evolution and rank mapping for PACKS and SP-PIFO under a uniform distribution (8 queues).

Finally, we argue that, for the highest priority packets, PACKS causes no more rank inversions than AIFO.

Theorem 3 *For any packet sequence, given the same window size, total buffer size, and burstiness allowance, PACKS causes no more priority inversions than AIFO for the highest priority packets.*

Proof: The theorem follows from two statements: (a) AIFO and PACKS admit the same set of packets (under the same configuration, see Theorem. 2), and (b) The quantile estimate of the highest priority packet is always the smallest (equalling 0). Let t denote the index of the packet in the input sequence. Let I_{PACKS} and I_{AIFO} denote the number of higher-ranked packets that t follows in the output sequence of PACKS and AIFO, respectively. From (b), we can show that for a given packet with the highest priority and index t, there is no packet that arrives after t (having an index greater than t), and is going to be dequeued before packet t. Rephrased, this means that $I_{\text{PACKS}} \leq I_{AIFO}$. This is true for each packet of highest priority. Thus, PACKS always has at most the same total number of priority inversions as AIFO for the highest priority packets. □

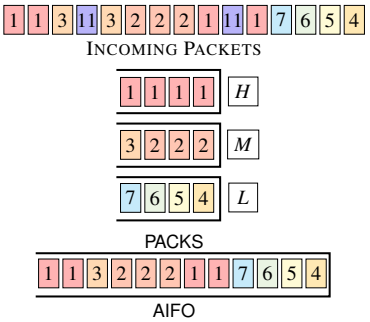


Figure 16: Adversarial input that maximizes the gap between weighted priority inversion of AIFO compared to PACKS. AIFO can delay the highest priority packet by more than 60% of the queue length. (Starting window = [1, 1, 1, 1]).

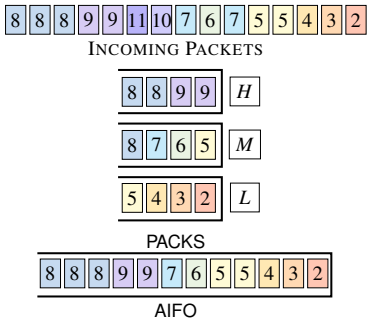


Figure 17: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared AIFO. (Starting window = [1, 1, 1, 1]).

B PACKS's analysis with MetaOpt

MetaOpt [23, 24] is a tool to compare the performance of two competing heuristics or a heuristic and an optimal solution. It identifies adversarial inputs that cause the maximum difference between the performance of the two algorithms.

We model PACKS in MetaOpt and compare it to AIFO [37], SP-PIFO [6], and PIFO [32]. Our goal is to understand *when and under what inputs* PACKS performs substantially better or worse than them. We focus on two performance metrics. The first metric is the number of packets dropped weighted by the packet's priority (where the priority is defined as the difference between the maximum rank in the distribution and the packet rank: max. rank - packet rank). The second metric is the number of priority inversions weighted by the packet's priority. These metrics help us identify the adversarial inputs that cause the heuristics to disrupt the performance of lower-rank packets (which are most important in the PIFO context).

Experiment setup We let packets take ranks between 1 and 11. We consider *all* the 15-packet traces possible with these ranks. We set the buffer size to 12 packets, and assume it empty at start. We configure PACKS and AIFO with a window size $|\mathcal{W}| = 4$ and a burstiness allowance $k = 0$. We configure SP-PIFO and PACKS with 3 priority queues of 4 packets each.

B.1 Comparison with AIFO

Packet drops We find that PACKS and AIFO always admit the same set of packets when they use the same configuration. This is expected, as we prove in Theorem. 2.

Rank inversions Fig. 16 and Fig. 17 illustrate the adversarial inputs that MetaOpt discovered for AIFO with respect to PACKS and vice versa. We find that:

> AIFO can delay the highest priority packets by more than 60% of the total queue size compared to PACKS.

AIFO only has an admission policy and suffers when the input sequence is not sorted. In Fig. 16, we show an input sequence where AIFO causes 24 priority inversions for lowest-rank packets, whereas PACKS is able to fully sort the packets.

> Adversarial inputs to AIFO consist of lower ranked packets compared to the adversarial inputs to PACKS.

PACKS underperforms when a distribution shift happens, and packets in the window are not a good estimate of the newer incoming packets. The worst case of PACKS compared to AIFO is on a packet sequence that is approximately sorted (Fig. 17). Due to the distribution shift, PACKS ends up mapping higher-priority packets to lower-priority queues and lower-priority packets to higher-priority queues.

Note that, in practice, the impact of scheduling such packet sequence with PACKS would not be significantly detrimental. Since queues are empty at start, PACKS would start sending the lower-rank packets while higher rank packets arrive.

This adversarial sequence (Fig. 17) consists of packets with higher ranks than the adversarial input to AIFO (Fig. 16), which have lower importance. This indicates that AIFO can cause higher average delays for important sensitive packets compared to PACKS. Our results show that PACKS never causes more priority inversion for the lowest ranked packets than AIFO (as we also prove theoretically in Theorem. 3).

B.2 Comparison with SP-PIFO

Packet drops Fig. 18 and Fig. 19 show the adversarial inputs that MetaOpt discovered for SP-PIFO with respect to PACKS and vice versa. We find that:

> SP-PIFO can drop more than 60% of high-priority packets while leaving 66% of the total queue size empty.

SP-PIFO lacks an admission policy and underperforms when we have a stream of packets with the highest priority (all with rank 1). In this case, SP-PIFO maps all the packets to

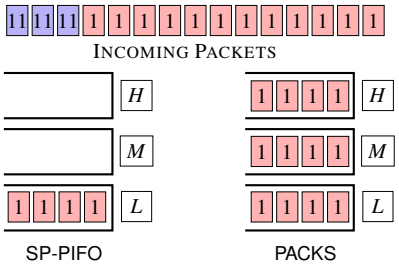


Figure 18: Adversarial input that maximizes the gap between weighted packet drop of SP-PIFO compared to PACKS. (Starting window = [1, 1, 1, 1]).

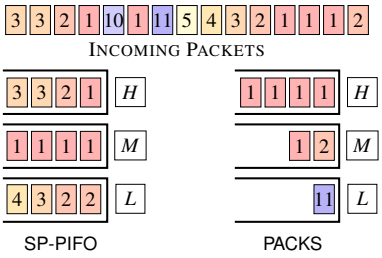


Figure 19: Adversarial input that maximizes the gap between weighted packet drop of PACKS compared to SP-PIFO. (Starting window = [1, 2, 1, 1]).

its lowest-priority queue and ends up dropping many of them while the other queues are empty. PACKS, however, fills the queues one by one from highest to lowest priority, efficiently using the buffer resources and preventing packet drops.

> PACKS drops at most 3 high-priority packets whereas SP-PIFO can drop up to 8 high-priority packets (2.33× more).

PACKS underperforms when the input sequence meets two conditions: (i) the rank of most of the packets increases, and (ii) a few of the packets in the middle of the trace have a higher rank than the ones received before or after them. Condition (i) describes an adversarial case for both SP-PIFO and PACKS, but condition (ii) helps SP-PIFO mitigate this by moving to a higher priority queue. Even with this, PACKS drops at most 3 high-priority packets more than SP-PIFO (2.33× less than the packet drop of SP-PIFO on its adversarial input).

Rank inversions In order to capture only the impact of rank inversions, we set the queue sizes long enough so that packet drops do not occur. Fig. 20 and Fig. 21 show the adversarial inputs that MetaOpt discovered. We see that:

> The adversarial input to PACKS is only slightly worse than the adversarial input to SP-PIFO.

The worst-case input for SP-PIFO with respect to PACKS causes 20 priority inversions for the highest priority packet, while the worst-case input for PACKS only causes 24 of them.

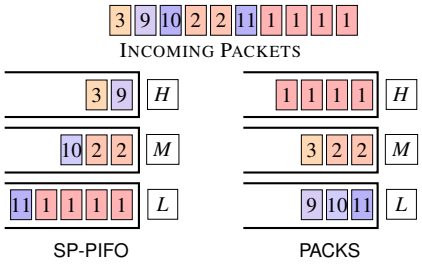


Figure 20: Adversarial input that maximizes the gap between weighted priority inversion of SP-PIFO compared to PACKS. (Starting window = [1, 1, 1, 1]).

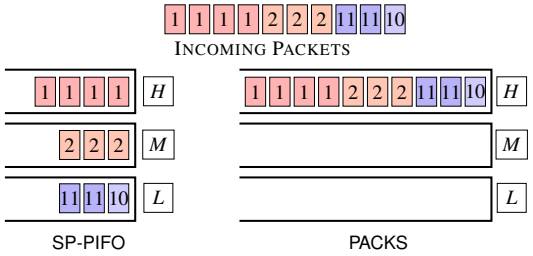


Figure 21: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared to SP-PIFO. (Starting window = [1, 1, 11, 11]).

SP-PIFO performs poorly when the rank of most of the packets is sorted, but there are a few packets in between with higher ranks than the ones received before or after them. These higher ranks cause SP-PIFO to push the rest of packets to higher-priority queues, leading to priority inversions. This pattern is the same as the one that causes PACKS to drop numerous packets compared to SP-PIFO.

PACKS underperforms when we can split the packets into multiple batches that meet two conditions: (i) the packets in each batch are in the non-decreasing order of their ranks, and (ii) all the packets in a given batch have higher rank than the packets in a subsequent batch. SP-PIFO would put each batch in one of its queues, achieving perfect sorting, whereas PACKS does not perform any sorting across batches of packets.

B.3 Comparison with PIFO

Fig. 23 and Fig. 22 show the adversarial inputs that MetaOpt discovered. We see that:

> The worst-case input to PACKS (with respect to PIFO) is the same as the worst-case input to AIFO (with respect to PIFO).

Packet drops The worst-case input is an increasing sequence of packet ranks. PACKS (similar to AIFO) computes the quantile using a sliding window. In this sequence, the quantile estimate of every packet is large, so PACKS (similar to AIFO)

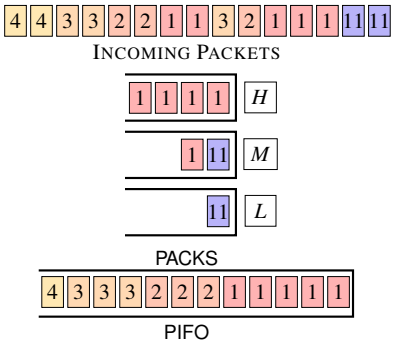


Figure 22: Adversarial input that maximizes the gap between weighted packet drop of PACKS compared to PIFO. (Starting window = [1, 1, 1, 1]).

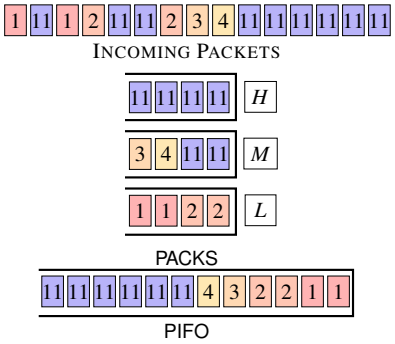


Figure 23: Adversarial input that maximizes the gap between weighted priority inversion of PACKS compared to PIFO. (Starting window = [1, 11, 1, 11]).

will drop the packets. The fact that both worst-case inputs to PACKS and AIFO, with respect to PIFO are the same, is expected, since PACKS and AIFO drop the same packets, as proved in Theorem. 2.

Rank inversions The worst-case input is a decreasing sequence of packet ranks. In that case, PACKS does not do any sorting and performs the same as AIFO (putting every packet in the highest priority queue with available space). This is also expected (cf. Claim 1 and the quick insight after its proof).

C PACKS's resource usage

We describe the resource requirements of our implementation of PACKS on Intel Tofino 2 [1] in Table. 1.

Resource Type	Usage (Average per stage)
Exact Match Crossbar	3.4 %
Gateway	3.4 %
Hash Bit	1.3 %
Hash Dist. Unit	4.2 %
Logical Table ID	10.9 %
SRAM	2.4 %
TCAM	0 %
Stateful ALU	23.8 %

Table 1: Resource requirements of PACKS on Intel Tofino 2.

When P4 Meets Run-to-completion Architecture

Hao Zheng[†], Xin Yan[△], Wenbo Li[†], Jiaqi Zheng[†], Xiaoliang Wang[†], Qingqing Zhao[△],
Luyou He[△], Xiaofei Lai[△], Feng Gao[△], Fuguang Huang[△], Wanchun Dou[†], Guihai Chen[†], Chen Tian[†]
[†] *State Key Laboratory for Novel Software Technology, Nanjing University, China*
[△] *Huawei, China*

Abstract

P4 programmable data planes have significantly accelerated the evolution of various network technologies. Although the P4 language has gained wide acceptance, its further development encounters two obstacles: limited programmability and the cessation of the next-generation Tofino chip. As a hardware manufacturer, we try to address the above dilemmas by opening the P4 programmability of our run-to-completion (RTC) chips. At present, there is no publicly available experience in this field. We introduce P4RTC, a comprehensive consolidation of our experiences applying the P4 language to RTC architecture. P4RTC introduces a new P4 architecture model and a set of beneficial extern constructs to fully leverage the RTC architecture's programmability. Besides, we share the insights we have gained from designing and implementing compilers. We also provide a performance model to facilitate profiling P4RTC's performance on user-customized P4 code. We prototype P4RTC on an RTC chip with 1.2 Tbps bandwidth. Case-oriented evaluation demonstrates that P4RTC can enhance P4 programmability and reduce the burdens of RTC development. The performance model can provide substantial insights into optimizing P4RTC programs.

1 Introduction

P4 programmable data planes have significantly accelerated the evolution of various network technologies, such as in-network computing [1, 2], congestion control [3, 4], load balancing [5], and network measurement [6–8]. As a domain-specific language (DSL) for data-plane programming, P4 has earned widespread recognition in academia and industry for its excellent abstraction. It makes network packet processing more accessible and easier to program. Empirical evidence demonstrates that P4 has become the de facto standard for data plane programmability [9, 10].

Although the P4 language has gained wide acceptance, its further development currently encounters two obstacles. Firstly, its programmability is limited, making implementing many practical designs and algorithms challenging. Currently, P4 is inherently tied to pipeline architectures (*e.g.*, RMT [11]), in which hardware constraints make many algorithms and designs have to make compromises on accuracy or performance [6, 7, 12, 13]. For instance, updating the minimum counter among multiple counters cannot be implemented directly in pipeline architectures since the stage where the minimum counter is located cannot be reaccessed unless recirculation. Secondly, Intel has announced that they will stop developing the next generation of the Tofino series [14], which currently stands as one of the most popular P4 programmable ASICs. To conclude, the P4 community urgently needs a successor hardware platform with Tbps-level throughput and more flexible programmability.

As a hardware manufacturer, we try to overcome the above dilemmas by opening the P4 programmability of our run-to-completion (RTC) chips. RTC is another packet processing architecture significantly different from the pipeline architecture, allowing different packets to have independent processing times. Packets can achieve line-rate forwarding when simple processing is sufficient, while others can be processed below line rate to enable more complex logic. Adapting the P4 language to the RTC architecture carries significant implications for the evolution of the P4 programmable data plane. On the one hand, P4 developers can use RTC devices to implement the functionalities that are difficult to implement in the pipeline architecture. In particular, RTC devices can perform any number of memory read/write operations. Thus, the 'updating the minimum counter' function can easily be implemented. On the other hand, P4 can facilitate the rapid development of the RTC-based data plane. P4 can effectively abstract away the underlying hardware details. Moreover, the recent advancements in the P4 programmable data plane can be easily applied to RTC devices with minimal effort, such as in-network computing [1, 2, 15–17], verification [18–21], packet scheduling [22–24], and formal foundations [25].

At present, there is no publicly available experience in adapting P4 to high-speed RTC devices. Existing extensions to the P4 language [26, 27] focus on reducing the complexity of developing functions and improving resource utilization. They cannot overcome the programmability limitations

of P4 inherent in pipeline architectures. Some efforts (*e.g.*, P4TC [28], P4-DPDK [29], T4P4S [30]) have successfully adapted P4 for CPU platforms. However, they haven't yet extended or optimized P4 for specialized RTC packet processors capable of Tbps throughput. Existing high-level programming languages for high-speed RTC devices are predominantly C-like languages [31]. Their API interfaces are specialized around their hardware and closed to the public community, making it challenging for researchers to understand the underlying architecture and detailed usage.

We propose P4RTC, a comprehensive summary of our experience in applying the P4 language to the RTC architecture. Our contributions can be summarized as follows:

Firstly, we present a new $P4_{16}$ architecture model, which bridges the P4 language and RTC architecture, providing a unified RTC programming abstraction. Based on the architectural model, we introduce a series of novel extern constructs to exploit the flexibility of RTC devices. We show the similarities and differences between P4RTC programming and traditional P4 programming and show how P4RTC can be used to develop functions that are challenging to realize in the pipeline architecture. We also discuss how parallel safety issues are appropriately handled in the P4 code.

Secondly, we share our insights while designing compilers, including table deployments and microcode generation. When deploying P4 tables, we must balance the read/write load among memory banks to improve performance. We develop an integer programming (ILP) model and leverage Profile-Guided Optimization (PGO) [32] strategies to improve the performance of the memory system. When generating microcode, we first transform P4 intermediate representation (IR) to microcode IR rather than directly to microcode instructions. This approach greatly enhances the correctness and efficiency of the generated microcode and makes it easier for the compiler to adapt to different microcode versions.

Thirdly, we provide a performance model to facilitate profiling the performance of P4RTC programs. Due to the resource sharing of the RTC architecture, bottlenecks can occur when many packets simultaneously access the same channel or hardware components. P4 programs no longer have deterministic throughput and delay as they are in the pipeline architecture. The performance model can provide various performance metrics (*e.g.*, pps, latency) to help users identify the bottlenecks of their P4 code under specific traffic patterns.

We prototype P4RTC in Huawei NetEngine 8000 F1A-C routers with 1.2 Tbps bandwidth. The adaptation to higher-throughput RTC chips is ongoing. We conduct several practical use cases. The first case is accurate per-flow monitoring. With the help of P4RTC, the chip can monitor up to 50M concurrent flows at near line rate, and reduce the bandwidth overhead by 86% to 90% compared with an existing Tofino-based solution [33]. The second case demonstrates that P4RTC reduces users' burdens on developing RTC devices. In five projects we examined, P4RTC can reduce lines of code (LOC) by 4.6x to 7.7x compared to microcode programming. This efficiency is comparable to another specialized high-level language developed for the chip. Additionally, using P4RTC, researchers can reproduce the functionalities of existing advances by modifying only 4.3% to 13.0% of the code. In the last case, we show how the performance model guides us in optimizing P4 programs.

This work does not raise any ethical issues.

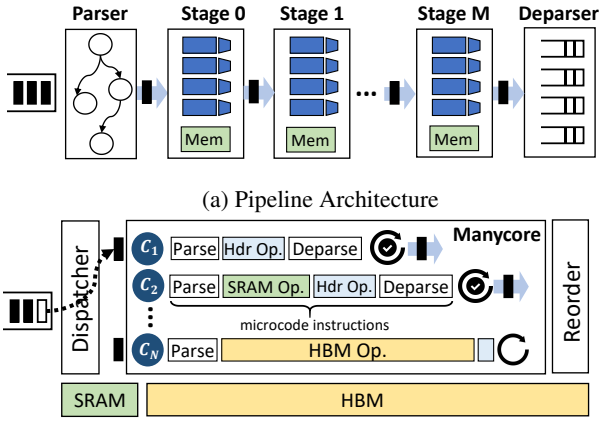


Figure 1: Difference between pipeline and RTC architectures.

2 Background and Motivation

2.1 Architecture Flavors of Dataplane

Packet processing is typically divided into a data plane and a control plane. The data plane, also known as the fast path, is responsible for the processing and forwarding of packets, whereas the control plane handles management logic (*e.g.*, adding routing entries). Developers utilize various hardware platforms, such as CPUs, FPGAs, and ASICs, to implement the data plane for fast packet processing. However, despite significant differences among the hardware platforms, the architecture for the data plane can be fundamentally categorized into pipeline and run-to-completion (RTC) [34].

As shown in Figure 1a, packets orderly pass through several stages in the pipeline architecture. A packet executes part of the processing logic at each stage and then moves to the next stage. Multiple packets can be processed simultaneously in different stages to achieve the 'pipelined' processing. This compact processing is highly efficient when handling high-speed packet arrivals [34]. However, it also introduces some limitations. Firstly, the physical pipeline length is limited in ASICs, which restricts its ability to handle complex logic. Secondly, to ensure the pipeline's efficiency and maximize throughput, resources such as memory are distributed in each stage and cannot be shared between different stages [12].

In the RTC architecture, many cores can independently parse, process, and forward packets. When a packet arrives, a dispatcher assigns the packet to an idle core, and then the

Table 1: Comparison of pipeline and RTC ASICs

Feature	Pipeline ASIC	RTC ASIC
Throughput	(Serial) High	(Parallel) Moderate
Latency	O(100) ns	O(100) ns - ?
Memory Capacity	10's MB	10's - 1000's MB
Logic Length	Limited	Unlimited

packet completes the entire processing on that core. This design allows different packets to have independent processing time, which overcomes the limitations introduced by the pipeline architecture. In particular, packets can perform unlimited read and write operations to a shared memory to implement complex algorithms at the expense of increasing processing time. As for the forwarding performance, RTC devices adopt a manycore architecture to improve the overall throughput to the Tbps level [31]. However, this can lead to memory conflicts and packet disorders.

Table 1 provides an overview of the distinct features of the two architectures. The pipeline architecture typically achieves high throughput under serial processing, while the RTC architecture needs to increase the number of forwarding cores (*i.e.*, parallelism) to improve throughput. Regarding latency, the pipeline architecture generally exhibits deterministic low latency in forwarding, whereas RTC devices' latency depends on the specific processing logic of the packets. As for memory capacity, RTC supports large-capacity memory such as high-bandwidth memory (HBM). Finally, both architectures have the potential for programmability.

2.2 Dataplane Programmability and P4

Reconfigurable Match-action Table (RMT [11]) is a famous programmable switch architecture. It is a typical pipeline architecture consisting of several reconfigurable match-action units (MAU). Developers program the data-plane logic by reconfiguring the structure of tables within the MAUs, including the match-phase key and the action-phase instructions. However, as mentioned earlier, its programmability is constrained, including the limited number of MAU stages, limited capacity and isolated memory. As another crucial packet processing architecture, some RTC devices [31] are also programmable by reloading microcode into chips. Regarding programmability, RTC devices offer more flexibility because they remove the constraints imposed by physical stages (*i.e.*, fixed lengths and isolated memory). This significantly bridges the gap left by the pipeline architecture.

For chip users, configuring the pipeline hardware or developing microcode can be laborious. P4 is a high-level language for data-plane programming, which was initially proposed in 2014 [35]. The P4 data plane consists of a series of programmable blocks, mainly including `parser`, `ingress`, `egress`, and `deparser`. These programmable blocks compose a logical pipeline named a top-level `package` or pipeline package. As an instance, the `main` object in Figure 2 is a

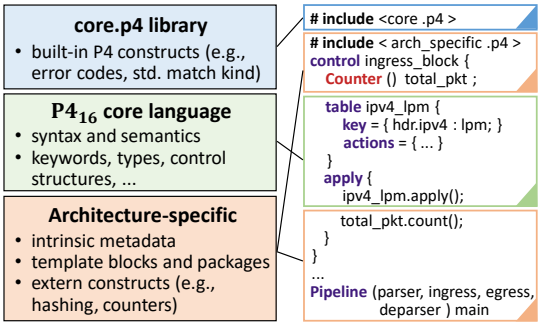


Figure 2: P4$_{16}$ language structures

pipeline package. Developers can customize programmable blocks within the pipeline package. P4 enables developers to implement diverse packet processing logic using only general-purpose operations and table look-ups. More importantly, P4 provides a general packet processing abstraction to shield developers from the underlying hardware details.

However, the original P4$_{14}$ language is complex, mainly attributed to the integration of architecture-specific features into the language, making it hard to adapt to other architectural models [36]. P4$_{16}$, as an improved version of P4$_{14}$, separates the architecture-specific features from the P4 core language. As shown in Figure 2, the basic syntax, semantics, and keywords (*e.g.*, `table`, `apply`) are defined in P4$_{16}$ core language. In contrast, the programmable blocks, extern constructs (*e.g.*, `Counter`), and pipeline packages are architecture-specific. P4$_{16}$ makes the P4 language relatively stable, leaving the architecture-specific parts to be implemented by target manufacturers. However, all existing P4 architecture models are based on the pipeline architecture, such as the PISA [37] used in P4$_{14}$, Portable Switch Architecture (PSA) [36], and Tofino native architecture (TNA) [38] introduced after P4$_{16}$. The adaptation of P4 to the RTC architecture remains unexplored, leaving a gap in the current research community. As a result, when researchers consider utilizing the P4 programmable data plane for designing new network features, they commonly regard the pipeline architecture as their primary target.

Providing a publicly available RTC programming model based on P4 is highly essential. Firstly, introducing the RTC architecture will enhance the programmability of the P4 programmable data plane. As discussed earlier, the pipeline architecture sacrifices programmability to enhance overall forwarding efficiency, which forces many algorithm implementations to make compromises. For instance, some systems execute their algorithm through recirculations, albeit at the cost of reduced bandwidth [4, 39]. Others [6, 40] opt to compromise on the design of their algorithms, implementing an approximate version instead, which inevitably results in a decrease in the algorithm's accuracy. Fortunately, the RTC architecture can make up this aspect, thanks to their unlimited logical length and large-capacity shared memory. With this

flexibility, P4 programmers can implement network features previously unachievable in the pipeline architecture. On the other hand, P4 can facilitate the rapid development of the RTC-based data plane. P4 abstracts away the tedious hardware details for users, while ongoing support for P4 allows existing advancements from the P4 community to remain valuable within the RTC architecture.

3 P4RTC Overview

As shown in Figure 3, the workflow of P4RTC consists of four stages: P4 programs, compilation, firmware, and targets.

Firstly, users define their customized data plane and control plane programs. In the compilation stage, a P4 compiler transforms the P4 code into microcode instructions, and a microcode compiler further generates a binary file according to it. The deployment of P4 tables is also completed at this stage. The binary file and configuration files generated during the compilation stage are collectively called the chip's firmware. Finally, the firmware is loaded onto the target hardware.

Given that both P4 and our RTC chip have reached mature stages individually, we strive to maximize component reuse during the implementation of P4RTC. As shown in Figure 3, the grey components are reused components. Specifically, P4 has public specifications and open-source compilers [41]. The chip used is not a brand-new design but was chosen from our existing RTC chips, previously used for fixed-function routers. Therefore, some legacy components exist, such as the microcode compiler and software development kit (SDK). However, we still encounter the following three challenges:

How to extend P4 to fully leverage the flexibility of RTC without altering P4$_{16}$ core language? Since the P4 language was initially designed around the pipeline architecture, directly referencing the existing architecture models cannot fully utilize the flexible programmability of RTC devices. To this end, we propose a new P4 architecture model for the RTC architecture. Under this architecture model, we provide a set of novel extern constructs (in `RTC.p4`) rather than modify the syntax and semantics of P4. We will introduce the new P4 architecture model and extern constructs in §4.

How does the compiler map the P4 language to the RTC architecture? Due to architectural differences in the hardware, the compiler needs to be designed explicitly for the RTC architecture, especially in table deployments and microcode generation. We implement a new backend for the open-source P4C compiler to generate RTC-specific microcode and table structures. We will introduce the novel issues in the table deployments of the RTC architecture and the experience in the microcode generation process in §5.

How do we verify the performance of the chip after loading customized P4 code? Unlike deterministic performance in the pipeline architecture, the flexibility of the RTC architecture makes it easy to introduce poor performance code. Conducting real machine experiments takes a lot of time and effort. It involves setting up connections, generating traffic, and collecting and analyzing results. More importantly, it is hard to accurately capture the performance details of the chip's internal components. To this end, we introduce a performance model to help users verify their P4 code in §6.

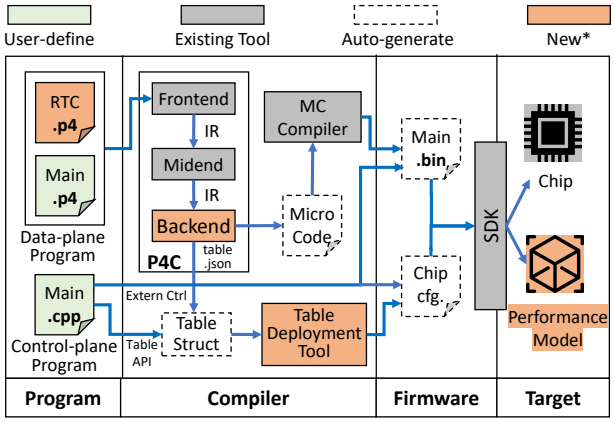


Figure 3: P4RTC system overview. The newly introduced components to address the challenges are colored orange.

4 P4RTC Programming

4.1 New P4 Architecture Model

Figure 4 illustrates the new P4 architecture model introduced by P4RTC. Considering a regular packet forwarding path, a packet enters the chip through the RX MAC and is dispatched to an idle core, where each core can be understood as a thread. Once a packet is handled by a core, all operational logic is executed on that core, where P4 programs define how to process and forward the packet. After processing, the packet is sent to a reorder engine for flow-level sequencing. Then, the packet is sent to the output queue of the destination port through the traffic manager (TM). After leaving the TM, the packet bypasses egress processing by default and is directly sent to TX MAC. If egress processing is necessary, it will again be assigned to a core. In general, the processing logic of the cores is the programmable part of the RTC architecture, while the other parts are non-programmable.

When programming RTC devices with the P4 language, the programmable blocks that users must write reserve those in the pipeline architecture, including the parser, ingress/egress control flow, and deparser. In other words, in the perspective of packets, packets are still processed by a logical pipeline composed of the traditional P4 programmable blocks. At the same time, the user's programming habits are also preserved. However, the concept of 'pipeline' in P4RTC has the following three main differences.

Many-core programming model. Packets are no longer serially processed by a single pipeline of physical stages. In contrast, there are many cores in RTC devices, and each packet is processed by an independent 'logical pipeline' operating on a dedicated core. Users define a pipeline package consisting of P4 programmable blocks, and then the cores run this package to process the packets in parallel. As shown

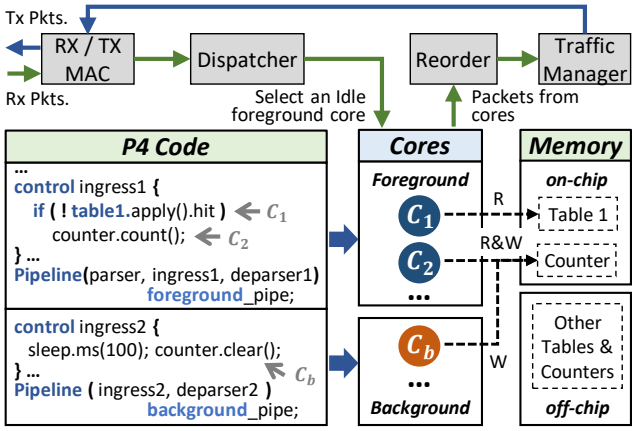


Figure 4: P4 programming abstraction for RTC architectures

in Figure 4, both the core C_1 and the core C_2 are running the *foreground_pipe* package defined in the P4 code. While C_1 is accessing `table 1`, C_2 updates a counter.

Unlimited logical length. In the RTC architecture, packet processing is performed by a dedicated core executing microcode instructions. The logical length is no longer constrained by the number of physical stages, as is the case in the pipeline architecture. This feature dramatically enhances chip programmability, but at the same time, it can easily lead to inefficient code, which degrades overall throughput.

Shared memory subsystem. Packets are no longer sequentially processed by multiple physical stages. Thus there is no need to access separate memory on these stages. Multiple cores can simultaneously access a shared memory system through on/off-chip memory controllers and crossbars. Given the shared nature of the memory, any memory location (*e.g.*, match-action tables, counters, and registers) can be accessed multiple times during the packet processing. As an instance, both C_1 and C_2 can access the counter extern in Figure 4.

4.2 Extensions for Pipeline Package

In the RTC architecture, the running instructions of each core can be defined separately. To provide a simplified view of P4 programming, the pipeline packages are grouped into two roles: foreground pipeline and background pipeline. By default, most cores run the foreground pipeline package, responsible for parsing, processing, and forwarding received packets. The other cores step back into the background. As shown in Figure 4, foreground cores C_1 and C_2 are running the P4 code in the 'foregound_pipe'. C_b is running the 'backgound_pipe' independently as a background core, which resets the counter every 100 ms.

A core running the background pipeline is stuck in an infinite loop of ingress $\rightarrow$ deparser $\rightarrow$ ingress $\rightarrow \cdots$. The background pipeline has no parser block since it does not require packet reception. However, it does not mean that background pipelines cannot process packets. Like standard P4 programming, users can use a `setValid()` method to enable protocol headers within the ingress block. When the control flow reaches the deparser, protocol headers marked as valid will be emitted. Therefore, the background pipeline can also generate customized packets.

To further exploit the programmability of the background pipeline, we introduce the following P4 extern constructs:

```
extern Foreach{
  void apply(in bit<32> start,in bit<32> end);
}
extern void break();
extern void continue();
extern Sleep{
  void us(in bit<32> value);
  void ms(in bit<32> value);
}
```

The `Foreach()` extern allows iteratively access to table entries by binding with a table. It can also be bound to a virtual table to implement a generic range for loop. The reason why we do not introduce a 'for loop' syntax like C/C++ is to retain the P4 core language, which is a crucial design philosophy of $P4_{16}$ for its target-independent feature. The `Sleep()` extern controls the activation frequency of background cores. The following example shows how to use the *Foreach* extern:

```
control back_ingress{
  action __for_act0(KEY_T k, DATA_T data)
  { /* loop body */ }
  @tbl("t")
  @methods(act0="__for_act0")
  Foreach() foreach_t;
  Sleep() sleep;
  apply{
    sleep.ms(10);
    /* setValid() related headers. */
    foreach_t.apply();
  }
}
control back_deparser {
  apply { pkt.emit(hdr); }
}
Pipeline(back_ingress, back_deparser) back_pipe;
```

In the above code, the `Foreach` extern binds a table called `t` and a loop body (*i.e.*, `__for_act0`). The core running the `back_pipe` will traverse each row of `t`, read the match key and action data of that row, and pass them to the loop body to perform the user-customized operations. Once the table traversal is complete, the control flow is transferred from `back_ingress` to `back_deparser`, and the `back_deparser` block sends a packet if any headers are valid. The `Sleep` extern can control the loop frequency.

It seems that the background pipeline is similar to traditional control plane threads. The main differences are that (i) it runs directly on the data plane and has a shorter control loop, and (ii) it has better performance scalability.

Inter-pipeline communication. To better assist in functional decoupling between different pipeline roles, we propose a `Queue` extern construct.

```
extern Queue<T> {
    Queue(bit <32> depth);
    bool push(in T value);
    bool pop(out T value);
    bool popWait(out T value);
}
```

The pipelines running on different cores can implement message push, pop, and a blocking pop through a parallel-safe queue extern. For example, the `Queue` extern can be used to implement messages passing between the foreground pipeline and the background pipeline. In the measurement project outlined in §7.1, we implement a background pipeline to periodically age out expired flow entries (*i.e.*, those that need to be reported to an analyzer). Nevertheless, employing a linear scan of the flow table proves inefficient, as the flow entries accessed by the background pipeline frequently do not meet the aging criteria. We adopt a smart aging technology to improve aging performance. When a foreground pipeline receives the FIN or RST flagged packet, it pushes the corresponding flow information into a `Queue` extern. The background pipeline just needs to pop flow IDs from the queue and directly complete the aging of the flows, which can significantly improve the aging performance. The corresponding P4RTC pseudocode can be found in Appendix A.

4.3 Extensions for Match-action Table

In P4RTC, we expand the types of match-action tables. According to the deployment location, tables can be divided into on-chip tables and off-chip tables. The on-chip tables have lower accessing latency, but the capacity is only a few tens of MB. In contrast, off-chip tables have GB-level capacity, but the access latency is an order of magnitude higher.

According to the memory layout, tables can be classified into content addressing tables (CAT) and linear addressing tables (LAT). The CAT tables perform exact matching and longest prefix matching (LPM) by storing keys in SRAM or specialized memories like content-addressable memory (CAM). In comparison, the LAT tables directly utilize the key as the memory index and then perform the action phase based on the action ID and action data stored in the memory address. LAT tables are more efficient than CAT tables, while it is limited by the capacity of the address space. They are commonly used for short-bit-width matching operations, such as Port-VLAN mapping.

The default definition of the P4 table is an on-chip CAT table. Users can use annotations to change the attributes of the table, including deploy locations, table layouts, and bank group size in the following format:

```
@linear
@offchip(x4)
table offchip_linear { ... }
```

The first annotation, `@linear`, indicates that a LAT table is required instead of CAT table. Another annotation, `@offchip`, signifies that the table needs to be deployed in off-chip memory to accommodate a larger table size. Additionally, users can specify the size of the bank group in the format `@offchip(x<group_size>)`, which improves memory read performance for off-chip tables.

In the RTC architecture, packets can operate table entries directly in the data plane, both for CAT tables and LAT tables. Compared with existing pipeline architecture that can only add or delete CAT entries from the control plane, the in-data-plane table operations have a shorter control loop, thus reducing the delay from milliseconds to hundreds of nanoseconds. To utilize this feature, we introduce a new P4 extern construct:

```
extern TableOperation {
  TableOperation();
  bool entryDel<KEY_T>(in KEY_T k);
  bool entryAdd<KEY_T, DATA_T>(in KEY_T k,
                               in bit<8> act,
                               in DATA_T data);
  bool entryGet<KEY_T, DATA_T>(in KEY_T k,
                               out bit<8> act,
                               out DATA_T data);
}
```

Table operations enable much of the logic that needs to be initially executed on the control plane to be efficiently offloaded to the data plane. A typical use case is to use the on-chip memory as a cache for off-chip big tables. We can dynamically add frequently hit entries to the on-chip memory. The code below shows an implementation example:

```
control foreground_ingress{
  table onchip_cache { ... }
  @offchip
  table offchip_tbl  { ... }
  apply{ /* Only apply the off-chip table
    when a cache miss occurs. */
    if(onchip_cache.apply().hit == false)
      offchip_tbl.apply();
  }
}
control background_ingress{
  @tbl(onchip_cache)
  TableOperation() cache_op;
  @tbl(offchip_tbl)
  TableOperation() tbl_op;
  apply{
    /* Find a heavy key, denoted as 'hk' */
    bool re = cache_op.entryAdd(hk, act, data);
    if(re == false)
      { /* The cache is full. Make eviction. */}
  }
}
```

In the above code, we implemented a simple caching strategy for an off-chip large table named `offchip_tbl`. The `foreground_ingress` only accesses `offchip_tbl` in case a cache miss happens. The `background_ingress`, which is packaged in a background pipeline, periodically loads hot entries into the `onchip_cache` through the `entryAdd()` table operation. Note that we omit the selection of the heavy key and the cache eviction strategy in the code. Through the `Foreach` extern, users can customize their heavy-key selection and idle-key eviction strategies.

Furthermore, we introduce a `lastRowIndex()` extern method. If the `lastRowIndex()` method is called after applying table or executing table operations, it returns the row id of the operated table entry. An important use of `lastRowIndex()` is attaching counters and registers with match-action tables, thereby constructing some advanced data structures. For example, suppose we have an exact matching table with size n and a counter array with size $2n$. With the `lastRowIndex()` extern, the counters can serve as a one-to-two supplement for the exact matching table, which

can implement packet&byte counters for each table entry.

```
control foreground_ingress{
  table tbl {... size=64}
  Counter() cnt(128);
  apply{
    table_1.apply();
    bit<32> row_index  = lastRowIndex();
    bit<32> pkt_index  = row_index << 2;
    bit<32> byte_index = (row_index << 2) + 1;
    cnt.count(pkt_index, 1);
    cnt.count(byte_index, getPktLength());
  }
}
```

4.4 Parallel Safety

A non-negligible concern is the handling of parallel safety of the RTC architecture. Most data-plane operations are stateless operations, which will have no parallel contention issues. However, some stateful operations involve modifying the device state. If not handled, it will lead to unpredictable errors.

The first type of stateful object is the `Counter` extern. The counter operations including `count`, `get`, and `set`, are implemented as atomic operations in hardware. The second type of stateful operation is the `TableOperation` extern that makes addition and deletion of flow tables. These operations involve inserting items into a hashing structure, such as cuckoo hashing [42] or d-left hashing [43], which incurs parallel issues when handling hash conflicts. Fortunately, users can regard the entire operation as an atomic transaction because efficient hardware locks guarantee parallel security.

However, when the above operations are combined to implement read-modify-write operations, it can lead to race conditions if not correctly synchronized. The key to ensuring parallel safety in read-modify-write operations is to prevent all other cores from initiating the same operation until the first core has completed its task. In P4RTC, we provide a `Lock` extern construct to solve this problem.

```
extern Lock {
  Lock(bit<32> size);
  bool acquire(in bit<32> idx, in bool blocking);
  bool release(in bit<32> idx);
}
```

The underlying of the `Lock` extern is the test-and-swap (TAS) atomic instruction. A core obtains a lock that can either be blocking or non-blocking. The distinction is whether the core chooses to spin and wait if it cannot acquire the lock immediately. A `Lock` extern can contain multiple lock instances, and users can implement row-level locking by utilizing the `lastRowIndex()` extern method mentioned before.

5 P4RTC Compilation

This section will discuss our experience in developing the P4RTC compiler, focusing on two main aspects: table deployments and generating microcode instructions.

5.1 Table Deployments

Table deployment entails strategically allocating P4-defined tables (*i.e.*, match-action tables, counters, registers) to the chip's physical memory resources. In the RTC architecture, there is no need to consider table dependencies as in a pipeline architecture; only capacity constraints and performance optimization issues need to be addressed.

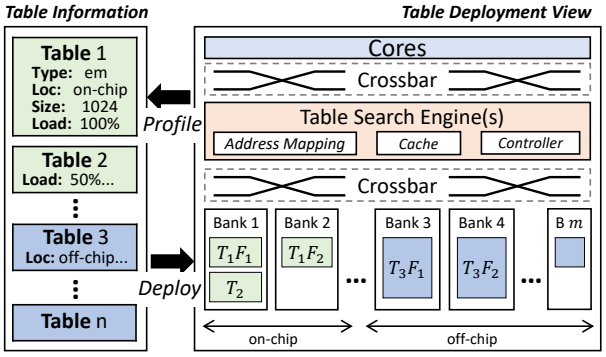


Figure 5: Table deployment view in P4RTC

Problem Constraints and Optimization Objective. The table deployment problem in the RTC architecture can be modeled as a variant of the bin packing problem (BPP), where the bins are the memory banks, and the items are the P4-defined tables. There are two main differences from the standard BPP. Firstly, in the shared memory subsystem, read and write requests caused by table matching and `TableOperation` extern will be queued in the channel of corresponding memory banks. Specific memory banks may suffer congestion due to simultaneous access by multiple cores, and the negative effect is more evident for off-chip deployed tables with higher latency. Therefore, the optimization goal is not to minimize the number of memory banks used but to minimize the maximum load among the memory banks to optimize the performance of the memory subsystem [1].

Secondly, P4RTC allows table fragmentation to utilize memory capacity fully. Specifically, we can fragment a table to satisfy capacity constraints or to reduce the load the table imposes on a single bank. However, each table fragmentation occupies an address mapping entry in table search engines (TSEs [2]), and the number of mapping entries available is limited. Consequently, this introduces a constraint that the quantity of table shards must be less than the number of supported mapping entries.

Based on the above considerations, we use an integer linear programming (ILP) model to solve the problem. Given n tables and m memory banks, each table i has a size of s_i, where $i = 1, 2, \ldots, n$. For each memory bank j, it has a capacity of c_j, where $j = 1, 2, \ldots, m$. We define an integer variable x_{ij} to represent the size of the fragment of the table i put in the memory bank j. We consider this variable as an integer (included between 0 and s_i). We also define a decision variable y_{ij} that equals 1 if the fragment of the table i is deployed

[1] Note that the load balancing of memory requests involves more hardware components (e.g., table search engines). Our experience is that as long as the load balancing of memory banks is done well, it is straightforward to handle the balancing of other upper-layer components.

[2] Table search engines are specialized memory controllers optimized for table operations, efficiently managing table search, read, and write requests.

on memory bank j, and 0 otherwise. Then, the total number of fragments of table i can be expressed as $\sum_{j=1}^{m} y_{ij}$. The following ILP model formalizes the table deployment problem:

$$\text{minimize} \quad \max_{j\in\{1,\ldots,m\}} \sum_{i=1}^{n} x_{ij} \cdot l_i \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{m} x_{ij} = s_i, \quad \forall i \in \{1,\ldots,n\} \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} \le c_j, \quad \forall j \in \{1,\ldots,m\} \tag{3}$$

$$\frac{x_{ij}}{s_i} \le y_{ij}, \quad \forall i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\} \tag{4}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} \le T \tag{5}$$

$$x_{ij} \in \mathbb{N}, \ \forall i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\} \tag{6}$$

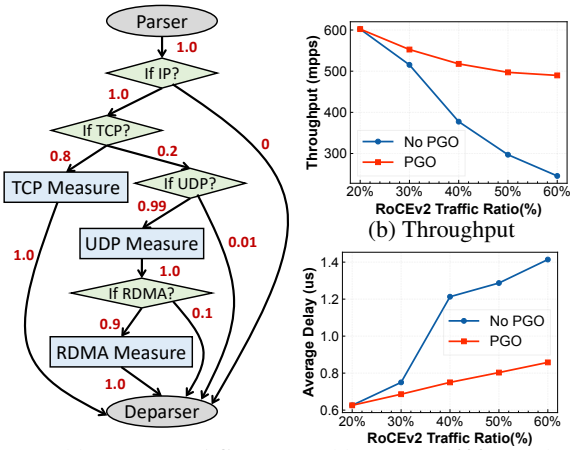
$$y_{ij} \in \{0,1\}, \ \forall i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\} \tag{7}$$

Here, l_i denotes the load incurred by the unit size of table i, while T denotes the maximum number of available mapping entries. The optimization objective (Equation (1)) is to minimize the maximum access frequency among all memory banks. Equation (2) is the table assignment constraint, ensuring that each table is fully deployed. Equation (3) guarantees that the capacities of the memory banks are not exceeded. Equation (4) mandates that the variable y_{ij} must be equal to 1 as soon as the variable x_{ij} is strictly greater than 0. Finally, Equation (5) ensures that the maximum number of mapping entries does not exceed.

Profile-guided optimization on table deployments. The above model relies on load information for each table. However, table access's read/write frequency is determined by runtime traffic and specific table entries. We employ a profile-guided optimization (PGO) compilation strategy. Initially, we adopt a profile where each branch evenly shares the load of its parent node. Then, after running the switch, a new table load information can be obtained [3], and we use the new profile to solve the table deployment problem again. Empirically, we perform PGO 2-3 times in the early stages of deployment to gradually adapt to the traffic of the new environment. In the subsequent stages, we will only proceed with a new round of PGO iteration if there is a significant change in the workload and it stabilizes in the new state.

The PGO's performance maintenance and enhancement are especially noticeable in scenarios with significant workload changes. For instance, within our traffic measurement project, we employ distinct off-chip tables to measure TCP, UDP, and RoCEv2 traffic, enabling the collection of protocol-specific metrics. The DAG graph of the program is shown in Figure 6a. Initially, TCP traffic predominated; however, with the gradual integration of RDMA technology across the network, RoCEv2 traffic has steadily gained prominence over the past years, as it also occurs in Microsoft Azure Storage [44]. We are comparing the performance of two strategies: one utilizing PGO optimization and the other not

[3]The compiler will automatically insert counter instrumentation.



(a) Program DAG (b) Throughput (c) Latency (400 Mpps)

Figure 6: Effect of PGO on performance.

using PGO. The PGO optimization strategy employs the optimal configuration based on the latest workload, whereas the non-PGO strategy consistently utilizes the initial configuration with RDMA traffic at 20%. As shown in Figure 6, as the proportion of RoCEv2 traffic increases to 60%, the non-PGO strategy's throughput drops by 64.7%, and its latency increases by 2.3x. In contrast, the performance of the PGO strategy remains good, with a slight performance degradation as RoCEv2 traffic needs to access more off-chip tables.

5.2 Microcode Generation

P4 is a high-level language that cannot be run directly on chips. Microcode are the closest language to the RTC chip's machine code, usually built from a specific instruction set designed by manufacturers. We must implement a compiler to generate the microcode instructions from P4 code. We reuse the open-source P4C project [41] but rewrite the compiler's backend. Due to space constraints in the paper, we provide an overview of the compilation process and our experiences. A detailed example of compiling P4 to microcode and intermediate representations (IR) is elaborated in Appendix B.

Define Microcode IR. Lowering P4 IR (*i.e.*, generated by P4C) directly into the microcode would increase the coupling between the compiler and microcode, making the compilation process difficult to control. We introduce a Microcode IR, enabling detailed control, optimization, and validation, considering the specific characteristics and constraints of the target hardware, as we will discuss further. Figure 7 shows part of Microcode IR definition for the `Foreach` extern:

Transform the P4 IR to Microcode IR. This is the most complex step in the compilation process. An example of the `Foreach` extern translation is shown in Figure 8.

1. *Align P4 IR with Microcode IR.* Specific Microcode IR nodes can be directly mapped from the equivalent nodes in P4 IR, such as assignment and logical operations shown in the right part of Figure 7. This step identifies the corresponding operations and control flow constructs within

```
/*McIR definition diff.
  from P4IR */
class Branch:Statement;
class Label:Expression;
class LabelStmt
          :Statement;
class EntryGet
          :Statement;
...
```

```
/*McIR definition equiv.
  to P4IR */
class AssignmentStmt
          :Statement;
class IfStatement
          :Statement;
class Lss
          :Expression;
...
```

Figure 7: Microcode IR Definition

the Microcode IR that align with those in the P4 IR. During this process, the compiler must carefully handle type translation, including endianness, padding, and bit width.

2. *Implement P4-specific and architecture-specific constructs.* Handle complex P4 constructs such as match-action tables by mapping them to corresponding Microcode IR nodes. New extern constructs introduced by P4RTC are implemented in this step. As shown in Figure 8, the `Foreach` extern is mapped to an assembly-like loop control flow, including a series of label statements, jump instructions (*i.e.*, Branch), and loop body. This process reads relevant information from multiple parts of the P4 IR based on the desired loop structures in microcode.
3. *Manage Dependencies.* Ensure that the semantic properties and constraints of the P4 program are preserved in the Microcode IR. Consider the dependencies between P4 IR operations and ensure they are correctly accounted for in the Microcode IR. This includes preserving data dependencies, control flow dependencies, and ensuring the proper ordering of instructions in the microcode to avoid hazards or incorrect behavior.
4. *Validate and Verify.* Ensure the correctness and integrity of the transformed Microcode IR. This includes maintaining correct packet processing behavior, adherence to P4 language rules, compliance with micro-architectural restrictions or limitations, and checking for potential errors and inconsistencies.

Emit Microcode Instruction. After completing the previous phases, the lowered Microcode IR can be directly mapped to the microcode instructions (please see Appendix B). The more details are lowered in the Microcode IR level, the less complexity there is in the emission stage, resulting in fewer introduced bugs. Besides, when facing different versions of microcode, Microcode IR allows developers to reuse existing compilation logic, especially target-independent compilation passes, such as assignment and logical operations.

6 Performance Model

In the RTC architecture, P4 codes and actual workloads significantly impact the chip performance. Therefore, strict performance testing must be conducted on P4 programs before deployment. Besides standard functionality models like BMv2 [45], we introduce a performance model for P4RTC to explore potential bottlenecks and optimize users' programs.

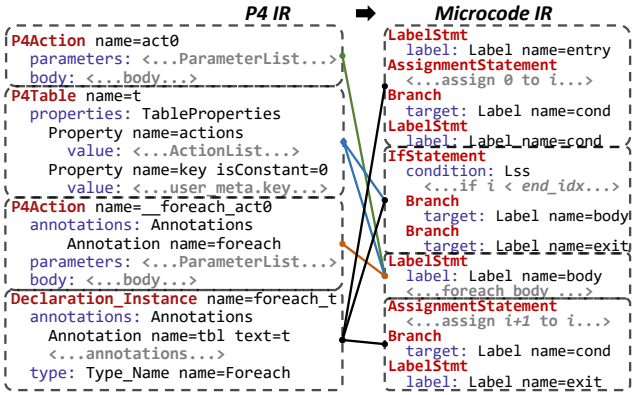


Figure 8: An example of P4 IR to Microcode IR

In general, performance models are utilized to evaluate metrics such as the throughput and latency of a chip under specific programs and workloads [46, 47]. We build a system-level model utilizing the Electronic System Level (ESL) methodology, leveraging the robust capabilities of the SystemC library [48]. We model main hardware components, encompassing cores, the traffic manager (TM), TSEs, the memory subsystem (MSS), and their interconnections (*e.g.*, FIFOs and back-pressure mechanisms). The model employs a global clock (*i.e.*, `sc_clock` in SystemC) to synchronize the chip components, define their operational frequencies, govern instruction execution, and regulate packet rates. As a result, the model can simulate cycle-level behaviors across a range of packet processing scenarios, enabling a more granular real-time depiction of each chip component's status.

As shown in Figure 9, our performance model uses four steps to obtain the performance metrics:

Step 1. Given a P4 program and a specific workload (*i.e.*, a packet trace), we perform functional simulations [45, 49] to process packets, record the microcode paths that each packet traverses, and then save the paths to a *codepath* file. For instance, the sequence `Parser` → `TCP Measure` → `Deparser` in Figure 6a represents an execution path. To realize the system-level simulation, we must also model the hardware behaviors. Specifically, when applying a P4 table, packets access various table entries, thus generating different microcode instructions that interact with multiple memory banks. However, for large P4 projects, recording all different microcodes for an extensive number of packets may result in the *codepath* file becoming excessively large. To deal with these scenarios, we use a probabilistic approach to construct the *codepath* file. We summarize all microcode paths in the *codepath* file as multiple *branches* and *sub-branches* of microcodes, each bearing distinct probabilities. The below example illustrates a sample *codepath* file:

```
<codepath name="demo" bw="1000Mpps">
  <microcode percentage="90%">
    <!-- microcode branch 1 -->  ...
    <io type="read" table="IPv4", tse="0", ...>
      <!-- sub-branch -->
```

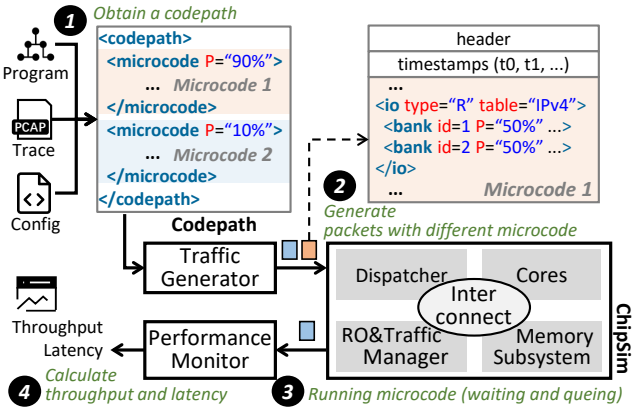


Figure 9: Workflow of performance model

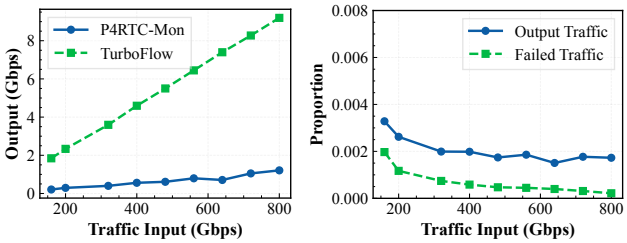
```
      <bank id="31" percentage="50%", .../>
      <bank id="32" percentage="50%", .../>
    </io>
    <action>
      <!-- sub-branch -->
      <alu name="l3_fwd" percentage="90%", .../>
      <alu name="drop" percentage="10%", .../>
    </action> ...
  </microcode>
  <microcode percentage="10%">
    <!-- microcode branch 2 --> ...
  </microcode>
</codepath>
```

There are two microcode *branches*, each with several *sub-branches*. The selection of the *branches* and *sub-branches* is performed dynamically in the following steps.

Step 2. After obtaining a *codepath* file, we upload it to a traffic generator. The traffic generator then interprets the *codepath* and generates traffic at a specified rate. Specifically, the traffic generator creates packets based on the probability of each microcode *branch*. Each packet includes a reference to the relevant microcode instructions, an identifier, and a metadata space for storing timestamps, which are used to calculate the latency experienced in each chip component.

Step 3. The chip simulator receives packets and navigates the microcode instructions. For each instruction, the simulator only emulates the performance behavior rather than executing it precisely. Take an IO instruction as an example: it will generate specific IO requests for the memory subsystem and emulate behaviors, including addressing and queuing a particular memory bank. Each behavior costs specific cycles, consistent with the chip's microarchitecture [50]. When encountering a *sub-branch*, the simulator dynamically employs a probabilistic process to execute the selected instruction, such as bank selection. When a component is over-stressed, a back-pressure signal is sent to upstream components, triggering chip-wide performance impacts.

Step 4. A performance monitor receives the packet output from the chip simulator and summarizes various performance metrics, including throughput and latency. Moreover, the performance monitor can acquire load or utilization information on each hardware component by inserting monitoring probes



(a) Bandwidth Cost (b) Traffic Proportion

Figure 10: Accurate Per-flow Monitoring

(*i.e.*, codes for monitoring) into the chip simulator.

In §7.3, we demonstrate how the performance model can guide performance optimization and assess its accuracy by comparing its outputs with those of a hardware chip.

7 Case Study

We prototype P4RTC on a 1.2-Tbps RTC chip with 8 GB high-bandwidth memory (HBM), used in Huawei NetEngine 8000 F1A-C (NE8000F1AC) routers. The adaptation to higher-throughput chips with the same architecture is ongoing. We evaluate P4RTC through several use cases and summarize the following findings:

P4RTC benefits P4 programmability. It incorporates RTC architectures into the P4 community, enabling previously unsupported functions in P4. A comprehensive case on accurate per-flow monitoring demonstrates that P4RTC can monitor up to 50 million concurrent flows at a near line rate. When the input traffic surpasses 800 Gbps, the report traffic sent from the data plane to a collector occupies less than 0.06% of the total input bandwidth (see case 1).

P4RTC benefits RTC-based devices. Besides existing C-like high-level languages, P4RTC offers another efficient method for data plane development. It reduces the lines of code (LOC) by 4.6x-7.7x compared with microcode programming. Besides, existing P4 functions can be migrated to P4RTC with minimal effort (see case 2).

The performance model benefits P4RTC optimization. It details the performance impact of different P4 codes on the chip under specific traffic inputs. In case3, we employ the performance model iteratively to identify system bottlenecks and implement targeted optimizations. Finally, we enhance the throughput to 14.4 times that of the baseline program.

7.1 Case#1: Accurate Per-flow Monitoring

Accurate per-flow monitoring is helpful for network performance analysis since it completely records flow-level traffic information. Due to the limitation of memory capacity and programmability, the task is challenging to implement on existing pipeline ASICs. Existing efforts employ a sampling strategy to monitor packets with general-purpose servers [51]. Nevertheless, this sampling strategy can lead to imprecise statistics and result in substantial information loss.

With P4RTC, implementing accurate per-flow monitoring

becomes straightforward and flexible. Specifically, we can define an off-chip big table as a flow measuring table (FMT) and the matching key as a flow key (*e.g.*, 5-tuple). Then, some counters and registers are defined to record the size, times-tamps, and other flow attributes. When a packet arrives in the foreground pipeline, it first tries to match a table entry. If a table miss occurs, we use `TableOperation` to add a new flow entry for recording. On the other hand, the FMT needs to age in time to make room for new flows. We complete the aging function by combining the foreground pipeline and background pipeline. In the foreground pipeline, we identify end-of-flow signals such as TCP's FIN and RST flags. In the background pipeline, we use the `Foreach` extern to traverse table entries, and we also use the `Queue` extern to accelerate the performance (see Appendix A). If a flow record is marked as finished or there is no packet access after a certain timeout, the flow will be reported and deleted from the FMT. This background pipeline can be bound to multiple cores to improve aging performance, where each core is responsible for aging a portion of the FMT.

We implement the above function with P4RTC. We connect a NE8000F1AC device to two ToR switches of a storage cluster (with a mix of client traffic and backend traffic) and mirror the truncated upstream packets to the device. With 4 GB of memory, it can measure up to 50M concurrent flows. It is lower than ideal (*i.e.*, 4 GB / the size of a flow record) due to hardware details such as memory alignment and redundancy of hash tables to reduce conflicts. We compare the performance with TurboFlow [33], a state-of-the-art accurate per-flow monitoring system based on Tofino. Turboflow evicts older flow records to the control plane when a hash collision occurs. In contrast, P4RTC can use the *TableOperation* extern to add flow records to an exact match table, automatically handling hash conflicts in the data plane. Therefore, it can continue to measure flows regardless of conflicts, reducing the additional bandwidth overhead caused by hash conflicts. As shown Figure 10a, our solution reduced the bandwidth overhead for sending measurement data by 86% to 90%. Note that packets will fail when multiple packets of a flow add entries simultaneously (*i.e.*, there is a row lock when adding table entries). In Figure 10b, we find that the bandwidth occupied by the failed packets is less than 0.2% of the input traffic. This is because packet failures occur only during the creation of flow entries due to lock contention. Once the table entries for the flows are established, subsequent packet counter updates proceed normally. Since the time taken to build entries in the data plane is short, the number of failed packets is minimal.

7.2 Case#2: Reduce RTC Developing Burden

Table 2 shows the comparison of the number of P4 code lines required for developing functions such as accurate flow monitoring in case 1 and SpaceSaving algorithm in Appendix C. We find that P4 can reduce the LOC development by 4.6x-7.7x compared with microcode programming in the five projects. This efficiency is comparable to NPC++, a C-like programming language developed for the chip. The NPC++-related projects are still under development. We estimate the LOC needed to replicate the same functionality using pseudo code written in NPC++ syntax. NPC++ will take about O(100) LOC and the error comes from the constant-level discrepancies between P4RTC and NPC++ languages (*e.g.*, table definition, 'for loop' syntax). Additionally, we successfully compile CocoSketch [6], ONTAS [53], and AES encryption [15] by only modifying 4.3%-13.0% of P4 code. These modifications mainly involve the architecture-specific extern constructs (*e.g.*, RegisterAction), and similar extern constructs can be easily found in P4RTC.

Table 2: Lines of Code (LOC). 'Modification (Mod.)' denotes the P4 code modifications required to migrate the original project to P4RTC. The required LOCs are comparable to NPC++, in the O(100) range.

Project	Microcode	$P4_{16}$ (Mod.)
Accurate Flow Mon.	2825	547 (N/A)
SpaceSaving [52]	1166	152 (N/A)
CocoSketch [6]	1244	270 (35)
AES Encryption [15]	2883	561 (41)
ONTAS [53]	3029	637 (27)

7.3 Case#3: Using the Performance Model

This case specifically aims to evaluate how the performance model can identify bottlenecks within the memory subsystem and guide optimization. Figure 11 is a sample P4 code we use when opening up the system process. The baseline deployment does not take any optimization. As shown in Table 3, its throughput is less than 40 Mpps. Then, we repeatedly used the performance model to guide the optimization.

Firstly, observing the status of memory banks in the baseline, we find that the utilization gap among the banks is significant. The maximum utilization is 0.99, the minimum is 0.03, and the average is low. This prompts us to adopt table fragments and use the load-aware table deployment strategy discussed in §5. After this optimization (O1), the system's performance increased 7.2 times.

After O1, we further find that load imbalance also existed among the table search engines (TSEs). We use a similar load-aware strategy to rebind the P4 tables to multiple TSEs (O2), balancing the loads of multiple TSEs. This performance optimization increases performance by 1.38 times.

Thirdly, when the bottleneck of TSEs is eliminated, the bottleneck point is once again transferred to the memory subsystem. We find all memory banks' utilization is above 90%. This reveals that the off-chip deployment of the P4 tables limits the system throughput. We offloaded a small number of hotspot table entries to on-chip memory (O3). This further improves performance to 1.45x over O2.

The Accuracy of the performance model. In the above

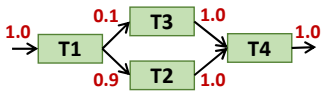


Figure 11: Case 3 P4 program. All off-chip tables.

Table 3: Performance analysis with the performance model.

Opt.	Banks Util. (Min / Max) Avg	TSEs Util. (Min / Max) Avg	Throughput Model (Mpps)	Throughput Hardware (Mpps)
Baseline	0.03 / 0.99 0.09	0 / 0.09 0.04	39.4	38.3
+ O1. Table Redeploy	0.63 / 0.90 0.68	0.03 / 0.63 0.32	283.9	279.4
+ O2. TSE Rebind	0.90 / 0.99 0.95	0.44 / 0.44 0.44	392.7	401.9
+ O3. Table Caching	0.77 / 0.96 0.82	0.63 / 0.63 0.63	569.5	561.0

experiments, the overall error of the performance was less than 3%, which is sufficient for the optimization scenario. We discuss the lessons and current limitations about the accuracy of the performance model in §8.

8 Lessons and Discussion

P4 language design for the run-to-completion architecture. The expressiveness of the RTC architecture is not limited to the externs proposed in this paper. In practice, we also developed some more specialized P4 extensions based on application requirements, such as using `@cache` annotation to directly support the table caching mechanism. Considering openness, the ultimate design of the P4 language for the RTC architecture merits discussion within open communities.

Performance challenges introduced by programmability. The NE8000F1A series comprises high-performance commodity routers designed with fixed functions that are optimized for their functional logic and configurations. Upon introducing programmability, we encountered numerous performance challenges. Some early applications consistently achieved throughputs of less than 100 Gbps. Contrasting with conventional deterministic performance switches, both toolchain developers and users need to invest additional effort to achieve the best performance.

Accuracy boundary of the performance model. Replicating all hardware details in a cycle-accurate manner is a monumental engineering challenge. The modeling granularity directly affects the accuracy and execution efficiency of the performance model. Currently, the performance model is built based on actual needs to maintain satisfactory prediction accuracy in the typical (or known) scenarios (*e.g.*, table deployments). Due to the probabilistic execution of the *codepath*, the current model cannot simulate some detailed behaviors, such as complex locking mechanisms and scenarios where the specific order of packets matters. Our ongoing development is focused on refining the model's traffic generator and chip simulator to support a richer array of scenarios and performance behaviors.

Parallel safety issues in P4RTC. Users must use the `Lock` extern reasonably according to their scenarios and consider potential performance risks. Better-performance concurrency management methods are worthy of future research, especially case-by-case optimizations and trade-offs. Our exploration in Appendix C provides an optimization case.

Future work. We will enrich P4RTC and adapt it to our higher-throughput RTC chips. Support for P4 Runtime is also in the planning process. Besides, we are applying P4RTC to more practical projects, such as traffic generators [54], in-network storage [1], and ML training acceleration [55].

9 Related Work

RMT [11] makes the concept of programmable switches a reality. Nevertheless, RMT is plagued by sub-optimal memory utilization due to the coupling between memory and the MAU stage. dRMT [56] enhances memory utilization and improves execution efficiency by moving table memories out of pipeline stages and into a centralized pool. P4ALL [26] and O4 [57] are extensions of P4 that support elastic switch programming and reduce the voluminosity of P4 code, respectively. Lyra [58] is a cross-platform language for data plane programming. Although these efforts simplify function development and enhance resource utilization, they are not designed to overcome the inherent limitations of P4's programmability within pipeline architectures.

There have been some applications and research around RTC architecture devices. Turboflow [59] employs Netronome NFP-4000 for implementing network flow records generation. Trio [31] is Juniper's programmable chipset using RTC architecture. The language design of P4RTC, along with its experience in compiler optimization and performance modeling, can serve as a reference for Trio. P4TC [28] focuses on applying the P4 language to the Linux tc layer. It is still programmed from a pipeline perspective. Designing new externs to extend the P4 language to the RTC architecture and leveraging the RTC architecture are not within the scope of its design.

10 Conclusion

We proposed P4RTC, a comprehensive summary of our experiences in applying the P4 language to the RTC architecture. P4RTC includes a unified programming model with novel extern constructs, compilers, and a performance model. Case-oriented evaluation demonstrates that P4RTC can enhance P4 programmability and reduce RTC development burdens.

Acknowledgments

We thank our shepherd, Srinivas Narayana, and the anonymous reviewers for their constructive feedback. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205, 62072228, and 62172204 , the Fundamental Research Funds for the Central Universities, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 121–136. [Online]. Available: https://dl.acm.org/doi/10.1145/3132747.3132764

[2] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "NetLock: Fast, Centralized Lock Management Using Programmable Switches," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 126–138. [Online]. Available: https://dl.acm.org/doi/10.1145/3387514.3405857

[3] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: high precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 44–58. [Online]. Available: https://dl.acm.org/doi/10.1145/3341302.3342085

[4] K. Liu, C. Tian, Q. Wang, H. Zheng, P. Yu, W. Sun, Y. Xu, K. Meng, L. Han, J. Fu, W. Dou, and G. Chen, "Floodgate: taming incast in datacenter networks," in *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 30–44. [Online]. Available: https://doi.org/10.1145/3485983.3494854

[5] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, 2016, pp. 1–12.

[6] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "CocoSketch: high-performance sketch-based measurement over arbitrary partial key query," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 207–222. [Online]. Available: https://doi.org/10.1145/3452296.3472892

[7] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "{SketchLib}: Enabling Efficient Sketch-based Monitoring on Programmable Switches," 2022, pp. 743–759. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/namkung

[8] ——, "Sketchovsky: Enabling ensembles of sketches on programmable switches," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1273–1292. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/namkung

[9] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021, conference Name: IEEE Access.

[10] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018, pp. 1–7.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[12] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 164–176. [Online]. Available: https://dl.acm.org/doi/10.1145/3050220.3063772

[13] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient Measurement on Programmable Switches Using Probabilistic Recirculation," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep. 2018, pp. 313–323, iSSN: 1092-1648.

[14] "Intel halts Tofino networking chip development - Silicon Valley Business Journal." [Online]. Available: https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html

[15] X. Chen, "Implementing aes encryption on programmable switches via scrambled lookup tables," *ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure (SPIN 2020)*, 2020.

[16] S. Vaucher, N. Yazdani, P. Felber, D. E. Lucani, and V. Schiavoni, "ZipLine: in-network compression at

line speed," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '20. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 399–405. [Online]. Available: https://doi.org/10.1145/3386367.3431302

[17] H. Zhu, T. Wang, Y. Hong, D. R. Ports, A. Sivaraman, and X. Jin, "{NetVRM}: Virtual register memory for programmable networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 155–170.

[18] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 518–532. [Online]. Available: https://dl.acm.org/doi/10.1145/3230543.3230548

[19] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 490–503. [Online]. Available: https://dl.acm.org/doi/10.1145/3230543.3230582

[20] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, "bf4: towards bug-free p4 programs," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 571–585.

[21] M. A. Noureddine, A. Hsu, M. Caesar, F. A. Zaraket, and W. H. Sanders, "P4aig: Circuit-level verification of p4 programs," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Supplemental Volume (DSN-S)*. IEEE, 2019, pp. 21–22.

[22] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 1–16.

[23] B. Vass, C. Sarkadi, and G. Rétvári, "Programmable packet scheduling with sp-pifo: Theory, algorithms and evaluation," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2022, pp. 1–6.

[24] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 685–699. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/sharma

[25] R. Doenges, M. T. Arashloo, S. Bautista, A. Chang, N. Ni, S. Parkinson, R. Peterson, A. Solko-Breslin, A. Xu, and N. Foster, "Petr4: formal foundations for p4 data planes," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–32, 2021.

[26] M. Hogan, S. Landau-Feibish, M. Tahmasbi Arashloo, J. Rexford, D. Walker, and R. Harrison, "Elastic switch programming with p4all," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 168–174. [Online]. Available: https://doi.org/10.1145/3422604.3425933

[27] A. G. Alcoz, C. Busse-Grawitz, E. Marty, and L. Vanbever, "Reducing P4 language's voluminosity using higher-level constructs," in *Proceedings of the 5th International Workshop on P4 in Europe*, ser. EuroP4 '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 19–25. [Online]. Available: https://doi.org/10.1145/3565475.3569078

[28] J. Hadi Salim, D. Chatterjee, V. Nogueira, P. Tammela, T. Osinski, E. Haleplidis, B. Sambasivam, U. Gupta, K. Jain, and S. Sethuramapandian, "Introducing P4TC - A P4 implementation on Linux Kernel using Traffic Control," in *Proceedings of the 6th on European P4 Workshop*, ser. EuroP4 '23. New York, NY, USA: Association for Computing Machinery, Dec. 2023, pp. 25–32. [Online]. Available: https://dl.acm.org/doi/10.1145/3630047.3630193

[29] P. Community, "p4-dpdk-target," https://github.com/p4lang/p4-dpdk-target, 2023, accessed: 2024-8-30.

[30] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2018, pp. 1–8, iSSN: 2325-5609. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8850752

[31] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi, "Using trio: juniper networks' programmable chipset - for emerging in-network applications," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser.

SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 633–648. [Online]. Available: https://dl.acm.org/doi/10.1145/3544216.3544262

[32] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P2go: P4 profile-guided optimizations," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020, pp. 146–152.

[33] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3190508.3190558

[34] H. Zolfaghari, H. Mustafa, and J. Nurmi, "Run-to-Completion versus Pipelined: The Case of 100 Gbps Packet Parsing," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, Jun. 2021, pp. 1–6, iSSN: 2325-5609.

[35] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2656877.2656890

[36] "P4~16~ Portable Switch Architecture (PSA)." [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[37] D. D. Robin and J. I. Khan, "P4TE: PISA switch based traffic engineering in fat-tree data center networks," *Computer Networks*, vol. 215, p. 109210, Oct. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138912862200295X

[38] I. Tofino, "Tofino2," May 2023. [Online]. Available: https://www.intel.com/content/www/cn/zh/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[39] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Efficient Measurement on Programmable Switches Using Probabilistic Recirculation," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep. 2018, pp. 313–323, iSSN: 1092-1648.

[40] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang, "{LightGuardian}: A {Full-Visibility}, Lightweight, In-band Telemetry System Using Sketchlets," 2021, pp. 991–1010. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/zhao

[41] "p4c," Aug. 2023, original-date: 2016-04-04T18:12:32Z. [Online]. Available: https://github.com/p4lang/p4c

[42] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403001925

[43] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3, Apr. 2001, pp. 1454–1463 vol.3, iSSN: 0743-166X.

[44] e. a. Wei Bai, "Empowering azure storage with RDMA," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 49–67. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/bai

[45] "BEHAVIORAL MODEL (bmv2)," Sep. 2023, original-date: 2015-01-26T21:43:23Z. [Online]. Available: https://github.com/p4lang/behavioral-model

[46] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-Aware Performance Prediction For Virtualized Network Functions," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 270–282. [Online]. Available: https://dl.acm.org/doi/10.1145/3387514.3405868

[47] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance Contracts for Software Network Functions," 2019, pp. 517–530. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/iyer

[48] "SystemC," Aug. 2023, page Version ID: 1172583305. [Online]. Available: https://en.wikipedia.org/w/index.php?title=SystemC&oldid=1172583305

[49] J. Xing, Y. Qiu, K.-F. Hsu, S. Sui, K. Manaa, O. Shabtai, Y. Piasetzky, M. Kadosh, A. Krishnamurthy, T. S. E. Ng, and A. Chen, "Unleashing SmartNIC Packet Processing Performance in P4," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, Sep. 2023, pp. 1028–1042. [Online]. Available: https://dl.acm.org/doi/10.1145/3603269.3604882

[50] "Microarchitecture - an overview | ScienceDirect Topics." [Online]. Available: https://www.sciencedirect.com/topics/computer-science/microarchitecture

[51] S. Panchen, N. McKee, and P. Phaal, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," Internet Engineering Task Force, Request for Comments RFC 3176, Sep. 2001, num Pages: 31. [Online]. Available: https://datatracker.ietf.org/doc/rfc3176

[52] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient Computation of Frequent and Top-k Elements in Data Streams," in *Database Theory - ICDT 2005*, ser. Lecture Notes in Computer Science, T. Eiter and L. Libkin, Eds. Berlin, Heidelberg: Springer, 2005, pp. 398–412.

[53] H. Kim and A. Gupta, "Ontas: Flexible and scalable online network traffic anonymization system," in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, 2019, pp. 15–21.

[54] Y. Chen, B. Tian, C. Tian, L. Dai, Y. Zhou, M. Ma, M. Tang, H. Zheng, Z. Yang, G. Chen, D. Cai, and E. Zhai, "Norma: Towards Practical Network Load Testing," 2023, pp. 1733–1749. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/chen-yanqing

[55] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling Distributed Machine Learning with {In-Network} Aggregation," 2021, pp. 785–808. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/sapio

[56] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "Drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–14. [Online]. Available: https://doi.org/10.1145/3098822.3098823

[57] A. G. Alcoz, C. Busse-Grawitz, E. Marty, and L. Vanbever, "Reducing p4 language's voluminosity using higher-level constructs," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 19–25.

[58] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 435–450. [Online]. Available: https://dl.acm.org/doi/10.1145/3387514.3405879

[59] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190558

[60] "LLVM Assembly Language Reference Manual." [Online]. Available: https://releases.llvm.org/1.1/docs/LangRef.html

[61] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," in *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2012, pp. 160–174.

[62] K. Cho, K. Mitsuya, and A. Kato, "Traffic data repository at the wide project," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '00. USA: USENIX Association, 2000, p. 51.

Appendix

Appendices are supporting material that has not been peer reviewed

A. Example of the Queue extern

In the following example, the foreground pipeline uses the Queue extern to send signals of finished flows to background pipelines. The motivation behind this design is this. First, the foreground pipeline offloads the reported tasks to the background pipeline for better logical separation. In this way, the cores running the foreground pipeline do not need more packet processing time when reporting is required. And the background pipeline can implement advanced reporting logic, such as batching and compression. However, it is inefficient for the backend pipeline to blindly scan the flow table to determine whether it needs to be aged. A simple optimization method is to pass the end-of-flow signal from the foreground pipeline to the background pipeline through the Queue to achieve more intelligent and efficient table aging.

```
Queue<FlowInfo> queue(4096);
control foreground_ingress{
  @offchip
  table FlowTable {...}
  ...
  apply{
    if (hdr.tcp.flags & TCP_FIN
              || hdr.tcp.flags & TCP_RST)
    {
        queue.push( { hdr.ipv4.srcAddr,
                      hdr.ipv4.dstAddr,
                      hdr.ports.srcPort,
                      hdr.ports.dstPort} );
    }
    /* Measurement logic here, store flow data
      in the action data of FlowTable */
  }
}
control background_ingress{
  @tbl(FlowTable)
  TableOperation() tbl_op;
  apply{
     /* Get an expired flow record */
     FlowInfo flow;
     queue.pop(flow);
     /* Read flow infomation*/
     bit<8> action_id;
     FlowData action_data;
     tbl_op.entryGet(flow, action_id,
                           action_data);
     ...
     /* Deleta the flow record*/
     tbl_op.entryDel(flow);
     /* Set valid the report header and send to
      the measurement analyzer. */
  }
}
```

B. Example of Microcode Generation

The following example introduces how to convert the `Foreach` extern into the final microcode step by step. Since the specific microcode cannot be made public for confidentiality reasons, we use LLVM assembly language [60] for demonstration, which is very close to our final microcode.

The P4 language snippet we're working with introduces an extern library for table traversal, named `Foreach`. This library also contains methods such as `continue()` and `break()`, which control the traversal flow. The `Foreach` extern is used in the P4 program as follows:

```
action act0(bit<32> value) {}
action act1(bit<32> value0,
            bit<32> value1) {}
table t {
    key = {
        user_meta.key : exact;
    }
    actions = { act0(); act1(); }
    size = 1024;
}

/* Add '@foreach' annotation to indicate this
   action is a method in 'Foreach' procedure */
@foreach
action __for_act0(bit<32> key,
                  bit<32> val) {
    // ...
    continue();
    /* Use 'continue' extern function to
       continue to the next iteration */
}
@foreach
action __for_act1(bit<32> key,
                  bit<32> val0,
                  bit<32> val1) {
    // ...
    break();
    /* Use 'break' extern function to
       break the loop */
}

// Instantiate a 'Foreach' variable.
// Bind this instance to table 't'
// Bind method '__for_act0'
//      to action 'act0' of table 't'
// Bind method '__for_act1'
//      to action 'act1' of table 't'
@tbl("t")
@methods[act0="__for_act0", act1="__for_act1"]
Foreach() foreach_t;
apply {
    // ...
    foreach_t.apply();
}
```

The `Foreach` extern needs to be bound to a table, and each table action of the table requires a separate `foreach` action (*i.e.*, loop body). This is because different actions usually have various types of action data, and we need to match the function parameter list.

Firstly, the P4C compiler translates the P4 language into the corresponding P4 IR. The P4 IR is a tree structure representing the P4 code in a more abstract form, including information about tables, actions, and extern constructs.

```
P4Action name=act0
  parameters: ParameterList
    Parameter name=value
      type: Type_Bits size=32 isSigned=0
  body: BlockStatement
    <... body ...>
P4Action name=act1
  parameters: ParameterList
    Parameter name=value0
      type: Type_Bits size=32 isSigned=0
    Parameter name=value1
      type: Type_Bits size=32 isSigned=0
```

```
  body: BlockStatement
    <... body ...>
P4Table name=t
  properties: TableProperties
    Property name=actions isConstant=0
      value: ActionList
        <... act0 ...>
        <... act1 ...>
    Property name=key isConstant=0
      value: Key
        <... user_meta.key ...>
P4Action name=__for_act0
  annotations: Annotations
    Annotation name=foreach
  parameters: ParameterList
    Parameter name=key
      type: Type_Bits size=32 isSigned=0
    Parameter name=value
      type: Type_Bits size=32 isSigned=0
  body: BlockStatement
    <... body ...>
P4Action name=__for_act1
  annotations: Annotations
    Annotation name=foreach
  parameters: ParameterList
    Parameter name=key
      type: Type_Bits size=32 isSigned=0
    Parameter name=value0
      type: Type_Bits size=32 isSigned=0
    Parameter name=value1
      type: Type_Bits size=32 isSigned=0
  body: BlockStatement
    <... body ...>
Declaration_Instance name=foreach_t
  annotations: Annotations
    Annotation name=tbl text=t
    Annotation name=methods
      NamedExpression name=act0 value=__for_act0
      NamedExpression name=act1 value=__for_act1
  type: Type_Name name=Foreach
```

We can see that the P4 IR uses the elements in P4 as the main content and is still far away from the final microcode. We first translate the P4 IR to Microcode IR. This transformation involves converting the high-level P4 constructs into lower-level ones that are more suitable for generating microcode instructions. For example, we must build a loop structure, conditions for entering or exiting the loop, and distinguishing the logic of calling different table action's loop bodies (*i.e.*, `SwitchStatement`). Here is the representation of the Microcode IR of the P4 code:

```
BlockStatement
  LabelStmt
    label: Label name=entry
  AssignmentStatement
    left: PathExpression name=i
    right: Constant value=0
  Branch
    target: Label name=cond
  LabelStmt
    label: Label name=cond
  IfStatement
    condition: Lss
      type: Type_Boolean
      left: PathExpression name=i
      right: Constant value=10
    Branch
      target: Label name=body
    Branch
      target: Label name=exit
  LabelStmt
    label: Label name=body
  EntryGet
    table: PathExpression name=t
    key: PathExpression name=idx
    act: PathExpression name=act
    data: PathExpression name=data
  SwitchStatement
    expression: PathExpression name=act
    SwitchCase
      label: Constant value=0
      statement: BlockStatement
        <... body ...>
    SwitchCase
      label: Constant value=1
      statement: BlockStatement
        <... body ...>
  AssignmentStatement
    left: PathExpression name=i
    right: Add
      left: PathExpression name=i
      right: Constant value=1
  Branch
    target: Label name=cond
  LabelStmt
    label: Label name=exit
```

Finally, the Microcode IR is transformed into microcode instructions. After the conversion from P4 IR to Microcode IR, the conversion from Microcode IR to microcode instructions is a straightforward parallel mapping.

```
%union.Data = type { %struct.Act0_Data }
%struct.Act0_Data = type { i32 }
%struct.Act1_Data = type { i32, i32 }

define void @foreach_t() {
     %act_ptr = alloca i8
     %data_ptr = alloca %union.Data
     %idx_ptr = alloca i32
     br label %enter

enter:
     store i32 0, i32* %idx_ptr
     br label %cond

cond:
     %cond_idx = load i32, i32* %idx_ptr
     %cond_result = icmp slt i32 %cond_idx, 100
     br i1 %cond_result, label %body,
           label %exit

body:
     %body_idx = load i32, i32* %idx_ptr
     %body_entry_get_result = call i1
                 @entry_get(i32 %body_idx,
                            i8* %act_ptr,
                       %union.Data* %data_ptr)
     br i1 %body_entry_get_result,
               label %body_switch,
               label %next

body_switch:
     %body_act = load i8, i8* %act_ptr
     switch i8 %body_act, label %next [
          i8 0, label %body_act0
          i8 1, label %body_act1
     ]

body_act0:
     %act0_idx = load i32, i32* %idx_ptr
     %act0_data = bitcast %union.Data*
                  %data_ptr to %struct.Act0_Data*
     %act0_result = call i1
                 @foreach_act0(i32 %act0_idx,
                               %struct.Act0_Data*
                               %act0_data)
     br label %next

```

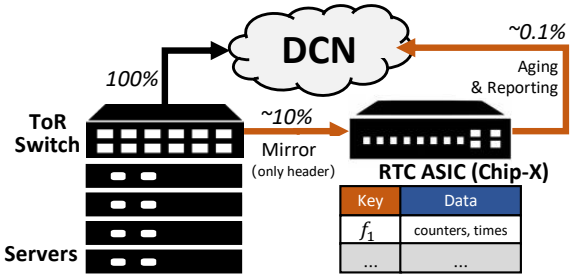


Figure 12: Testbed for Accurate Per-flow Monitoring

```
body_act1:
     %act1_idx = load i32, i32* %idx_ptr
     %act1_data = bitcast %union.Data*
                  %data_ptr to %struct.Act1_Data*
     %act1_result = call i1
               @foreach_act1(i32 %act1_idx,
                   %struct.Act1_Data*
                   %act1_data)
     br label %next

next:
     %next_idx = load i32, i32* %idx_ptr
     %next_new_idx = add i32 %next_idx, 1
     store i32 %next_new_idx, i32* %idx_ptr
     br label %cond

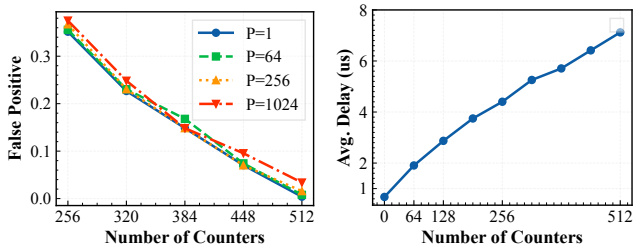
exit:
     ret void
}

declare i1 @entry_get(i32, i8*,%union.Data*)
declare i1 @foreach_act0(i32,%struct.Act0_Data*)
declare i1 @foreach_act1(i32,%struct.Act1_Data*)
```

C. Profiling Complex Algorithms

Algorithms that involve complex memory operations can now be implemented with the help of P4RTC. A typical example is SpaceSaving (SS) [61], which is utilized to detect the top-k flows. SS maintains O(k) slots in memory, where each slot contains a flow key and a counter to track large flows. Upon receiving a packet from flow i, the algorithm employs a loop to verify whether the flow is stored within a slot. If the flow is found, the algorithm increments the corresponding counter. Otherwise, it locates the slot with the minimum counter value and replaces the existing flow with flow i. These kinds of operations cannot be implemented in the pipeline architecture. With P4RTC, we can easily use `Foreach` extern to implement an SS algorithm.

However, parallel safety issues may occur during the updating process. Using locks can lead to significant performance issues as they negate the performance benefits of parallel packet processing. Therefore, we are curious whether SS can be applied directly without the locks. To deeply reveal the impact of the parallel safety issues on algorithm accuracy, we leverage our performance model (in §6) to help us analyze this process. We embed a customized execution environment in the performance model to actually run the actual instructions in the codepath, thus profiling the functional performance in a multi-threaded environment. We input a



(a) SS Accuracy Profiling (b) SS Latency Profiling

Figure 13: Profiling with the performance model

real-world packet trace collected by the WIDE Project in 2020 [62], which contains 10M packets. Figure 13a shows the false negative of the SS algorithm across numbers of cores at 1, 64, 256, and 1024. We detect top-256 large flows ($k = 256$) and the x-axis represents the number of slots allocated for SS. We surprisingly find that we can optimistically neglect the locks. Even when utilizing 1024 cores (parallelism $P = 1024$), the algorithm maintains its accuracy without noticeable degradation. This is primarily due to the stability of the large flow candidates in memory as the algorithm progresses. The accumulation of these atomic counters is parallel safe. The only potential parallel safety concerns arise during the eviction of the smallest flow. Fortunately, this operation does not impact the other slots that contain larger flows, thus the impact of parallel safety issues is limited.

The performance model can also help us balance parameter k (*i.e.*, the number of large flows that need to be identified) and performance overheads. Suppose for each k, we allocate $2k$ slots in memory to obtain a satisfactory recall ratio. Figure 13b shows the variation of average packet latency with the change of allocated slots. Developers can choose the most appropriate parameters k based on their tolerance for delays and the top-k flow requirements.

Table 4: Comparison between P4RTC with regular P4

Language	P4	P4RTC
Target	Pipeline	Run-to-completion
Architecture Model	Pipeline Only	Pipeline and Manycore RTC
Table Management	Control plane Only	Control plane and Data plane
Memory Layout	Isolated memory & Limited access	Shared on-chip/ off-chip Memory
Multi-Program Programming	Physical Pipeline Level	Logical pipeline level for each thread with different roles
Thread Safety	Safe	Unsafe
Performance Guarantee	Deterministic Performance	Varying Performance

Mutant: Learning Congestion Control from Existing Protocols via Online Reinforcement Learning

Lorenzo Pappone
Computer Science Department
Saint Louis University

Alessio Sacco
DAUIN
Politecnico di Torino

Flavio Esposito
Computer Science Department
Saint Louis University

Abstract

Learning how to control congestion remains a challenge despite years of progress. Existing congestion control protocols have demonstrated efficacy within specific network conditions, inevitably behaving suboptimally or poorly in others. Machine learning solutions to congestion control have been proposed, though relying on extensive training and specific network configurations. In this paper, we loosen such dependencies by proposing *Mutant*, an online reinforcement learning algorithm for congestion control that adapts to the behavior of the best-performing schemes, outperforming them in most network conditions. Design challenges included determining the best protocols to learn from, given a network scenario, and creating a system able to evolve to accommodate future protocols with minimal changes. Our evaluation on real-world and emulated scenarios shows that Mutant achieves lower delays and higher throughput than prior learning-based schemes while maintaining fairness by exhibiting negligible harm to competing flows, making it robust across diverse and dynamic network conditions.

1 Introduction

Years of research on congestion control algorithms for TCP [44] have produced creative solutions departing from TCP Vegas [13], Fast [30], Compound [47], and the more recent BBR [15], to name a few, with improvements in performance results of more than 15% compared to Cubic [25], the default in most Linux systems. BBR, rather than traditionally relying on packet loss to adjust the congestion window, (*cwnd*), uses round-trip time (*rtt*) and throughput data to decide how fast to transmit data packets. TCP Vegas, a delay-based protocol, only looks at the *rtt* to adjust the congestion window. An increasing *rtt*, due, for example, to a packet drop, causes Vegas to reduce its packet sending rate, while a steady *rtt* causes Vegas to increase its sending rate until the *rtt* increases again. These examples highlight that traditional congestion control solutions rely on current network stats such as *rtt*, throughput, and packet loss. As such, they are mostly reactive (not proactive) to sudden network changes. In addition, they rarely look at the past behaviors of the network or their own historical behaviors in their analysis.

With the surge of machine learning performance in recent years, promising approaches using Deep Learning (DL) or Reinforcement Learning (RL) techniques have been proposed to adjust performance, mostly by tuning the congestion window [29]. As an example, Aurora [28], a variant of the Performance-oriented Congestion Control (PCC) [21], uses Deep Reinforcement Learning to adjust the congestion control actions empirically by observing the connection state and the experienced performance. Owl [43], Orca, [2], and Marten [39], three other RL-based congestion control protocols, use Deep Q-Learning to determine the next congestion window using metrics retrieved from the network stack. These approaches share the idea of solving the congestion control problem by instructing an agent to dynamically learn the optimal *congestion window (cwnd)* via the RL framework. Despite their learning ability as the network environments change, their design is inherently limited by the training phase. Congestion control parameters often exhibit complex interactions and dependencies with various network factors, including traffic load, topology changes, and routing dynamics. Even when enough samples of the significant states are collected *offline*, i.e., ahead of time, data drift problems remain, and protocols experience performance degradations for their lack of adaptability under new, unseen, changing conditions.

The good news is that years of research on congestion control protocols have shown that some protocols can learn how to perform well by observing specific network input signals, and even though they fail in scenarios for which they were not optimized, there is often another (existing) protocol that does perform better in such scenarios. *What if we could learn from a team of protocols, each performing well in a few complementary scenarios, without requiring exhaustive and impractical data collections and without requiring the one-size-fits-all protocol?*

In this paper, we propose *Mutant*. While such a design is inspired by the so-called imitation learning [20], deciding

which protocol Mutant should “imitate" or “mutate" into is also a computationally hard problem even if we choose the best-performing k protocols of the team offline. To solve this problem bounding the optimality gap, we use the team selection theory [31]. In particular, our algorithm solves the following problem: *Given a set of congestion control algorithms available, which subset should we input to our online reinforcement learning algorithm, so that our selection is most likely to bring the best performance, i.e., smaller delays, higher throughput, or both, for each network environment and congestion state?* We reduce this problem from the top-k players in cooperating games and discuss the team selection algorithm used by Mutant in Section 5.2. There are tens of CC protocols to choose from and we show that having as many protocols as possible leads to inefficiencies, not only overhead. Similar problems have been applied in the context of supervised machine learning, where individual features are modeled as players and feature subsets as coalitions that yield the best predictive performance [19].

In Section 4 we justify the chosen subsets of protocols per each network scenario and discuss our lessons learned. Among those, (1) the datasets available and the choice of protocols to learn from are more important than the learning algorithm, and (2) learning from too many protocols is counterproductive: the performance degrades because, intuitively, we explore too much and we do not exploit enough.

Aside from algorithmic challenges, to deploy a protocol that mutates its behavior based on the underlying available user-space and kernel implementations, we needed to solve several system design challenges as well (Section 3). For example, our kernel module implementation should be able to quickly mutate into one of the congestion control algorithms used to learn with minimal overhead. For performance reasons, we focused on learning from congestion control protocols already implemented in the kernel, such as Cubic, Hybla, and TCP Vegas, but Mutant can be easily adapted to learn from any transport protocol that uses sockets and returns congestion window updates, including ML-based protocols. Our evaluation results (Section 6), on real network traces and real-world traffic, show that *Mutant* consistently performs among the best protocols in all scenarios analyzed, for both delay-sensitive and bandwidth-sensitive applications. We also analyzed *Mutant*'s fairness using the notion of harm [50], showing promising results. From our evaluation, it is also clear that the choice of which protocol set *Mutant* is programmed to emulate influences its performance, just as humans should not learn from anyone in any scenario. We review existing literature in Section 7 and we present our conclusion in Section 8.

2 Mutant: Motivation and Design Overview

Learning From Existing Protocols. The main insight behind our protocol design is the idea that there is no one-size-fits-all congestion control for all scenarios, as confirmed by our results in Figure 1. These results show the ratio between throughput and delay, and the fairness performance of four widely used TCP protocols. We consider the following six network environments sampled from our experimental sets: *fixed bandwidth*, *low capacity*, *high rtt*, *large queue* i.e., buffer of the bottleneck router, *mixed conditions* i.e., large queue with high RTT, *step* i.e., with regular and abrupt channel bandwidth fluctuations. In Section 6.1, we detail these testing scenarios. We observe that no protocol is dominant in all the cases for single-flow settings. Cubic, for example, performs well in stable conditions but fails across fluctuating bandwidth conditions (step scenario). BBR2 [16], on the contrary, handles these abrupt changes but is suboptimal for networks with low capacity. When we instead consider a multi-flow scenario, focusing on the fairness of these protocols, expressed in terms of *harm*, a notion introduced in [50] and defined in Section 6.4, we can see that, e.g., BBR2 harms others in many use cases but not in all scenarios.

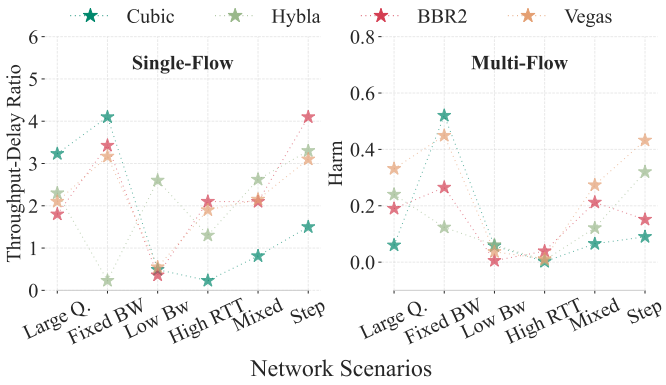


Figure 1: Motivation for our study: Performance of a few protocols in six different network environments, tput/delay on a single-flow (left) and harm [50] in multi-flow scenarios (right): low values translate to high fairness. Every protocol performs well in only a selected group of network scenarios.

The Challenge of Generalization. While adaptive and learning-based solutions have been explored, we believe that learning from past congestion window values is insufficient and poorly adapted to unseen network conditions. Current approaches attempting to achieve adequate model generalization require a long offline training phase, e.g., [42, 43, 55]. In Figure 2, we compare one state-of-the-art model as Sage [55] against our online model. The offline learning process requires vast data collection, expensive resources, and long training sessions. Mutant quickly adapts to the network scenario while achieving comparable or superior performance to the data-driven model. Motivated by these results, we propose a system that quickly adapts to diverse conditions and learns in real-time from ongoing interactions with the underlying network environment. In addition, our design principle is to offer the community a system that, rather than autonomously adapting

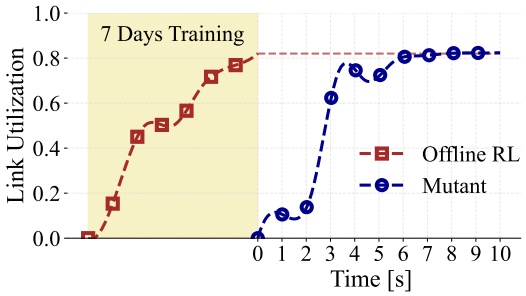


Figure 2: Mutant reduces the learning time overhead compared to pretrained, data-driven ML models for congestion control.

the best *cwnd* update strategy, considers other features to learn what would be the best protocol in a given network state.

Mutant Overview. At the core of our solution, *Mutant*, lies on an online Reinforcement Learning (RL) algorithm that *(1)* quickly learns from existing congestion control schemes by a real-time policy switching mechanism at fine-grained intervals implemented in kernel, *(2)* minimizes the reaction time to network changes, and *(3)* relies mostly on online data with minimal offline adjustments. We present the overview of Mutant in Figure 3, highlighting its main components. Our online learning logic decides on the best CC scheme in real-time during the communication. Two main components dictate the selection of the best scheme at each time step: the *Protocol Manager* and the *Learning Module*. The *Protocol Manager* is a pluggable kernel module implemented following the structure of the existing CC schemes in the Linux kernel. Indeed, Mutant is a kernel module that can be loaded as another kernel-based TCP congestion control protocol (i.e., Cubic, BBR2, Hybla, etc.). This component supports the key low-level functionalities of Mutant's system: *(1)* runs the real-time CC scheme switching logic, *(2)* maintains the state of each protocol during the execution, and *(3)* communicates the network statistics to the user-space *Learning Module* module.

The *Learning Module*, instead, is a user-space component that handles the algorithmic operations within our system. This module is responsible for executing *(1)* the online reinforcement learning process and *(2)* the best starting protocols using a *top-k* selection schema (Section 4). At each step of the RL loop, Mutant switches the CC scheme in real-time, by sampling from a pool of different protocols in the kernel. The user-space module will learn the relationship between the scheme selected and the features collected at each time step, with the primary goal of maximizing the throughput and minimizing the delay.

While designing our solution, we attempted to answer two key questions: *(1)* What pool of CC protocols should we consider in our deployment? *(2)* Which schema should we choose at runtime? To answer the first question, we questioned if all the CC schemes should be kept in the pool or if selecting only a subset of them would benefit the overall performance of our

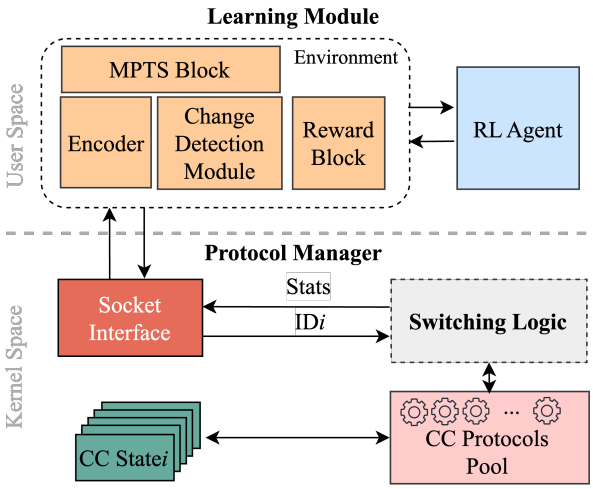


Figure 3: Overall architecture of *Mutant*. Two main components are a kernel module hosting a set of TCP protocols and a user-level module with the learning logic to decide when it is convenient to mutate into another protocol.

solution. Our results show that a subset of the best-performing policies for the specific network scenario leads to higher performance and faster convergence (see Section 6.2). We designed an algorithm that selects the top-*k* policies *per-flow* before running the online learning process (see Section 5.2). Secondly, Mutant runs a reinforcement learning procedure that can quickly learn *its own CC schema over time, based on the pool of existing protocols* (i.e., the CC heuristics in the kernel). By continuously adapting and exploring different protocols, Mutant can dynamically learn the strengths of each protocol. For instance, it might recognize that BBR2 excels in specific scenarios, such as when delays become particularly high or bandwidths change abruptly, while Cubic performs better in different conditions, for example, when packet loss events occur. In such a way, our solution can learn to adapt and refine its own congestion control policy over time, continually assessing and adjusting its strategies based on the evolving network conditions and performance metrics.

3 Design and Implementation

This section details the two main components overviewed in the previous section. First, we highlight the main challenges that originate from the kernel components and operations of the Protocol Manager. Then, we describe the Learning Module with the RL-based algorithm.

3.1 Protocol Manager

Our Protocol Manager seamlessly integrates in-kernel functionalities with the Linux kernel's TCP source code. The Mutant kernel module acts as a versatile meta-protocol for congestion control. The kernel utilizes a modular and extensible framework for implementing TCP congestion control

algorithms, with the struct `tcp_congestion_ops` at its core. This structure consists of function call pointers that define an interface for congestion control algorithms. Each function can be seen as a handler for a specific TCP event. Examples of events can be congestion avoidance (i.e., a congestion event has been triggered), receipt of a TCP ACK, or the adjustment of the TCP sender's state. Each function is designed to promptly respond to these events by executing the necessary actions that facilitate congestion management and update the internal state variables of the CC scheme. Each congestion control algorithm encapsulates its functionalities within this structure, enabling effortless pluggability into the system.

In our implementation of Mutant, we have created an independent and flexible *meta-protocol*. This module acts as a wrapper for the `tcp_congestion_ops` of the protocols in the pool and can dynamically plug and unplug them for live switching. Our live switching procedure functions as follows: when the Protocol Manager receives a new CC scheme to select from the Learning Module, it performs two important operations. Firstly, it stores the current state of the active CC scheme, which can be restored if selected again later. Secondly, it plugs the `tcp_congestion_ops` of the selected CC scheme, seamlessly replacing the previous one. Each CC scheme's state variables are maintained over time, and they are frozen and unfrozen when the scheme is selected. Additionally, upon every switch, the next protocol starts from the `cwnd` value of the preceding one. To communicate with the RL module in the user space, the protocol manager uses the *netlink* socket family. After receiving every ACK, it sends the network features to the Learning Module, which predicts the CC scheme to be selected for the next step.

3.2 Learning Module

This module hosts the online learning capability using a Reinforcement Learning (RL) approach. In RL, an *agent* performs an action, *a*, in a particular state, *s*, of an *environment* and receives a reward, *r*, for having performed this action [46]. An *agent* learns a *policy* to maximize its reward over time, where such a reward value indicates how successful the agent's action has been. This motivates the agent to seek for a series of state-action pairs that eventually maximizes its overall reward via a process of exploration and exploitation, with varying degrees.

In this work, we design an online learning algorithm based on contextual Multi-Arm Bandits (MAB) [36]. Similar to the more popular *Q-Learning*, *MAB* is an RL algorithm that makes decisions under uncertainty by balancing between exploration and exploitation of several actions, known as *bandits*. Simply, the algorithm observes a context, makes a decision by selecting one action from a set of alternative actions, and then observes the outcome of that decision. However, unlike traditional RL, bandit problems only observe the outcome of a selected action for a given state, and it does not have

Table 1: Raw input statistics used in the learning process. For each feature in **bold**, we also compute the mean, minimum, and maximum values across three observation windows of different sizes: *short* (10), *medium* (200), and *long* (1000).

`snd_cwnd`	Sender's congestion window size
`rtt_us`	Round-trip time in microseconds
`srtt_us`	Smoothed round-trip time
`mdev_us`	Mean deviation of round-trip time
`min_rtt`	Minimum round-trip time
`advmss`	Advertised maximum segment size
`delivered`	Number of packets delivered
`lost_out`	Rate of packets lost
`in_flight`	Number of packets sent
`retrans_out`	Number of retransmitted packets
`rate`	Delivery rate
`prev_proto`	ID of the previous protocol
`crt_proto`	ID of the selected protocol
`throughput`	Throughput
`loss_rate`	Loss rate

Q-tables to look up past performances and decide the best action to take [36]. By being simple in the learning nature, this approach can be used to learn *online* among possible choices and, thus, leads to faster adaptability to new network conditions.

Environment. It acts as the interface through which the agent interacts with its surroundings, offering feedback to the agent's actions and shaping its learning trajectory. The environment runs in our *Learning Module*, which collects the network parameters from the kernel at runtime and provides the observation at each time step. The duration of this time interval is customizable, but it impacts the switching frequency between different congestion control (CC) schemes within the pool. We explored various time intervals through empirical experimentation to understand their impact on model performance in Section 6.1.

Agent. The agent interacts with the Mutant environment by observing states, taking actions, and receiving rewards. The agent aims to learn the best mapping between states and actions to maximize cumulative rewards over time. In contextual MAB, the agent faces a set of actions (*arms*) and must choose the next action based on observed contexts or features. As each action yields an uncertain reward, the agent aims to learn which action maximizes cumulative rewards given different contexts. In multi-arm bandits algorithms, the agent employs a policy to balance the trade-off between exploration and exploitation. In Mutant, we implement the Upper Confidence Bound [32] as the exploration-exploitation policy (see Section 5.1).

State. In our work, we consider a diverse set of network statistics from the kernel during the communication between two hosts as our input features (Table 1). For a subset of these features, we additionally consider three observation windows to extract temporal dynamics varying the time granularity

across *short* (10 samples), *medium* (200 samples), and *long* (1000 samples) durations. This leads to a total of 55 input features collected at each step from the kernel. To efficiently manage this input space, we employ a pre-trained encoder to map these features in the latent space to achieve dimensionality reduction (see Section 5.1), resulting in a total of 16 embedding input signals.

Actions. The arm or action determines what an agent observing an environment can do to influence the environment. In our context, *Mutant* has the following set of actions represented as $A = tcp_proto_i \; \forall \; i$, where i is the index of the CC scheme deployed on the host machine. According to the learning agent, Mutant selects the next scheme based on the network conditions it has seen. For every network scenario and before the agent starts learning, we run our MPTS protocol to determine the pool of *top-k* schemes, given a fixed k. However, k is a parameter of *Mutant*, and we designed our solution with the aim of providing the flexibility of including other (future) protocols in the kernel thanks to an extensible and versatile interface [1]. Our prototype tested in this paper supports 11 different in-kernel CC schemes: Cubic [25], Hybla [14], BBR2 [15], Westwood [17], Veno [24], Vegas [13], YeAH [9], Bic [53], HTCP [33], Highspeed [23] and Illinois [35]. Selecting the best scheme at each time interval is a non-trivial problem. We discuss such a problem and solution in Section 4.

Reward. A reward is a utility function that allows us to calculate the effectiveness of the action by measuring a reward value for an agent observing an environment. In our context, we want our agents to maximize the network throughput while minimizing the delay. We express this objective as an attempt to maximize the Power metric, a well-known quantity defined as $pw = \frac{Throughput}{Delay}$. At the same time, we also aim to minimize the number of packet losses. As such, our agents seek to maximize their overall reward at time step t, R_t defined as:

$$R_t = \frac{(thr_t - \zeta \cdot l_t)^{\kappa}}{d_t}, \tag{1}$$

where thr_t, d_t, l_t are the delivery rate, average delay, and loss rate observed at timestep t. ζ and κ are coefficients that determine the impact of the loss rate over the throughput and the importance of the throughput over the delay, respectively. We compute the throughput as the delivery rate during the communication directly in the kernel, defined as $thr_t = \frac{r_t}{s_t}$, where r_t is the number of packets delivered and s_t the time interval in *us*.

Algorithm 1: Mutant's Online Learning Algorithm

```
Let δ represent the step duration
Let N denote the total number of steps
Let x_t denote the raw network features
Compute top-k CC schemes via MPTS (Algorithm 2).
Initialize thr_max and del_min
for step in n_steps do
    Predict id_i
    Switch current scheme to id_i
    while δ has not expired do
        Read x_t from kernel module
        Compute the embeddings z_t = f_enc(x_t)
        Update state with z_t
        Update thr_max and del_min
    Compute μ_a = θ*^T z_t
    Compute UCB_a = μ_a + sqrt(2 log(t) / n_a)
    Update MAB policy parameters
R_max = thr_max / del_min
R_t = ((thr_t − ζ·l_t)^κ / d_t) · (1 / R_max)
```

4 Problem Statement: Selecting the Best Protocols to Learn From

A cooperative game is characterized by a pair (N, ν) containing a set of players $N = \{p_1, ..., p_n\}$ and a value function $\nu : \mathcal{P}(N) \rightarrow \mathbb{R}$, where $\nu(\emptyset) = 0$ by definition. The players can form coalitions $S \subseteq N$ and obtain a combined benefit given by $\nu(S)$, which is called the worth of S.

Our problem is characterized by a set of strategies, or arms $\mathcal{A} = \{a_1, ..., a_n\}$ and at each discrete time step t, the learner can *pull* an arm a_i. To each arm i there is an associated probability distribution $p_i \in [0, 1]$. The observed reward for arm i is drawn from p_i. We model our congestion control protocol set selection in Mutant as a multi-armed problem.

In particular, the resolution of the top-k arms identification problem determines the protocols to use. The Mutant agent selects the k arms with the highest mean rewards to form a coalition at the end of T evaluations. In the multi-arm bandit literature, the parameter T is denoted *fixed budget*. Such budget is given beforehand, and once exhausted, the learner returns its guess about the top-k arms. The learner's performance is hence measured by the probability of returning a correct output. Formally, the objective of the learner is to select the set $\{a_1, ..., a_k\}$ corresponding to the set of arms with the k highest mean rewards, $\mu_1, ..., \mu_k$. In the rest of this section, we assume that the final set of arms is ordered by the reward, i.e., $\mu_1 > ... > \mu_k$. This ordering assumption preserves the generality, while the assumption that the means are all distinct is made for the sake of notation (the complexity measures differ slightly if the top k means are ambiguous).

We assess the effectiveness of the Mutant agent's strategy by using the probability of misidentification, i.e.:

$$e_T = \mathbb{P}(\{a_1, ..., a_k\} \neq \{1, ..., k\}). \tag{2}$$

Considering the following complexity measure for the single

[1] Mutant code is publicly available at https://github.com/lorepap/mutant

best arm identification:

$$H_1 = \sum_{i=0}^{n} \frac{1}{\Delta_i^2} \quad \text{and} \quad H_2 = \max_i \frac{i}{\Delta_i^2}, \tag{3}$$

where $\Delta_i = \mu_1 - \mu_i$ for $i \neq 1$ and $\Delta_1 = \mu_1 - \mu_2$ (difference in mean rewards). Prior work [7] showed that these two complexity measures are equivalent up to a log factor, i.e.,

$$H_2 \leq H_1 \leq \log(2n) H_2. \tag{4}$$

Intuitively, H_1 represents a lower bound on the number of evaluations necessary to identify the best arm. To verify if an arm has mean $\mu^* = \max_i \mu_i$ or μ_i, and agent needs to sample $\frac{1}{\Delta_i^2}$ times. The surprising fact shown in [7], that we reuse in our analytical result, is that the order of H_1 evaluations suffices to identify the best arm. As for upper bounds, the quantity H_2 proved to be a useful surrogate for H_1 to express the bounds on e_T.

To model our k-best arms identification problem, we then define the gaps and the complexity measures as follows:

$$\Delta_i^k = \begin{cases} \mu_i - \mu_{k+1} & \text{if } i \leq k \\ \mu_k - \mu_i & \text{otherwise} \end{cases}$$
$$H_1^k = \sum_{i=0}^{n} \frac{1}{(\Delta_i^k)^2}, \quad H_2^k = \max_i \frac{i}{(\Delta_{(i)}^k)^2}, \tag{5}$$

where the notation $(i) \in \{1, .., n\}$ is defined such that $\Delta_1^k \leq ... \leq \Delta_m^k$. In our case, the H_1^k replaces the H_1 as the lower bound of the k-best arms identification problem.

5 Mutant Online Learning Logic

After the team selection process, we oversee the ultimate lineup and dynamically choose the protocol responsible for controlling congestion. We formulate the online learning procedure in Mutant (Algorithm 1) - which changes the CC scheme in the pool based on the real-time collection of network features - as a Contextual Multi-Armed bandit problem (CMAB) [36]. Unlike the traditional multi-armed bandit problem, where the rewards of each arm are independent of the context, in CMAB, the reward of each arm (i.e., the CC schemes) depends not only on the arm chosen but also on a set of contextual features (i.e., the network parameters) associated with the current state of the environment. Formally, at each time step t, the agent observes a context vector $\mathbf{x}_t$ that describes the current state of the environment and selects an arm i based on a policy $\pi(\mathbf{x}_t)$. The policy maps each context vector to a probability distribution over the arms, indicating the likelihood of selecting each arm given the context. The agent then receives a reward $r_{t,i}$ associated with the chosen arm.

To address this trade-off problem, we leveraged the Linear Upper-Confidence Bound policy, which we combine with an algorithm to preliminary select the policies in the pool (Section 5.2). We design our algorithm as a cooperative game. Unlike competitive games, where players compete to maximize an often selfish utility, cooperative games foster teamwork and coordination among participants [41]. In our system, all players (protocols) aim to reach "high throughput and low delay" for all flows. An intriguing question in the context of cooperative games concerns the importance of a single player's contribution. In particular, one crucial question is: What are the top players performing in this game?

5.1 Challenges to Achieve an Online Learner

Exploration vs. Exploitation Trade-Off. One main challenge in online RL is balancing the trade-off between exploration, i.e., trying out new actions or strategies to gather information about the environment, and exploitation, i.e., leveraging known information to maximize short-term rewards. While too much exploration may lead to sub-optimal performance, excessive exploitation can result in missed opportunities for discovering better solutions, given the uncertainty surrounding the efficacy of the chosen protocol and the varying network conditions. This difficulty arises because achieving quick and cost-effective convergence becomes impractical without investing significant time and resources. As an online RL approach, our model must minimize the exploration time and ensure consistently good performance by selecting the best-performing protocol for the underlying network settings.

The key challenge in the CMAB problem is to learn an effective policy that balances the exploration of different arms with the exploitation of arms that are likely to yield high rewards, considering the contextual information. To this end, we use the LinUCB (Linear Upper Confidence Bound) algorithm [34], which extends the classic Upper Confidence Bound algorithm [8] to the contextual setting.

LinUCB assumes a linear relationship between the expected reward of an arm and associated contextual information. Specifically, the expected reward for pulling arm a at round t, given a context feature vector x_t with dimensionality d, can be expressed as $\mu_a = \theta^{*T} x_t$, where θ^{*T} captures the weights associated with each feature. However, since we do not have direct access to θ^*, LinUCB employs an estimate $\hat{\theta}_t \in \mathbb{R}^d$ that is updated over time based on observations. Our approach entails Bayesian linear regression [12] to model such a relationship between the contextual features and the expected rewards for each action, thus forecasting the weights θ^{*T}. This linear relaxation simplifies the traditional RL process, making the learning policy a more lightweight process suitable for our purpose of learning online.

In addition to predicting the reward, LinUCB computes an upper confidence bound (UCB) for each action's expected reward. This bound represents a trade-off between the estimated mean reward and the uncertainty in that estimate. In conclusion, by selecting actions with higher UCB values, LinUCB can balance exploration (selecting actions with uncertain re-

wards) and exploitation (selecting actions with potentially high rewards), aiming to maximize cumulative rewards over time.

Reduce the Input Representation. From the raw network features indicated in Table 1, we compute maximum, minimum, and mean values for some metrics over short, medium, and long-size sliding windows. While this approach allows to extract temporal dynamics at runtime, it also leads to a total of 55 different network features. Such a large state space is hardly managed by LinUCB, which approximate a linear relationship between the context (i.e., network features) and the reward of each arm (i.e., the protocols in the pool). If, on the one hand, such a linear algorithm enables a more lightweight policy, on the other hand, this simplistic assumption poses new challenges as it needs more representational power, especially with a large context vector.

Inspired by [40], to overcome this limitation, we design an *encoder* network to represent a large network feature vector into a smaller latent space. On top of that, this operation should be done in an online fashion. At each step, the high-dimensional raw observation x_t is processed by a pre-trained encoder that outputs the low-dimensional embeddings z_t. We use a Gated-Recurrent Unit [18] layer with two fully connected (FC) layers as our encoding network architecture to capture the sequence-level information from the online network data. Thus, the expected reward for each arm becomes $\mu_a = \theta^{*T} z_t$, where z_t is the output of the last hidden layer of the encoding network for the context x_t.

Dynamic Reward Normalization. Since the reward defined in Equation 1 captures the ratio between the throughput and the round-trip time, as the bandwidth can change over time, the most-rewarded protocol can change accordingly. Because of this, we need to normalize the reward according to the maximum achievable reward for the particular network setting. We adopt an online change detection algorithm based on ADWIN [11]. This algorithm can detect the throughput and the round-trip time shifts during the communication for each selected CC scheme. Every time a change is detected, we recalculate the maximum reward by replacing the throughput and round-trip time values in Equation 1 with each adaptive window's maximum throughput and minimum round-trip time. This approach ensures that the normalized reward reflects the best-performing CC scheme at each step, even when the underlying network setting changes.

5.2 Top-*k* selection with MPTS

We present an algorithm — *Mutant Protocol Team Selection (MPTS)* – to find the best CC algorithms to learn from, and we show that such an algorithm has a bounded error. Before proceeding, we introduce the following notation. For each arm i and for all rounds $t \geq 1$, we have $W_i(t) = \sum_{s=1}^{t} \mathbb{1}_{I_t=1}$ to be the number of times arm i was pulled from rounds 1 to t, and $X_{i,1}, X_{i,2}, ..., X_{i,W_i(t)}$ the sequence of associated rewards. Then,

Algorithm 2: Mutant Protocol Team Selection (MPTS) algorithm for k-best protocol identification.

Let $\mathcal{A}_1 = \{a_1, ..., a_n\}, k(1) = k, n_0 = 0$, $\overline{\log}(T) = \frac{1}{2} + \sum_{i=2}^{T} \frac{1}{i}$, and for $j \in \{1, ..., n-1\}$ $n_j = [\frac{1}{\overline{\log}(T)} \frac{T-n}{n+1-j}]$;

Initialize ;

for *each phase* $j \in \{1, ..., n-1\}$ **do**

1. For each active arm $a_i \in \mathcal{A}_j$ select arm a_i for $n_j - n_{j-1}$;
2. Let $\sigma_j : \{1, ..., n+1-j\} \to \mathcal{A}_j$ be the bijection that orders the empirical means by $\hat{\mu}_{\sigma_j(1),n_j} \geq \hat{\mu}_{\sigma_j(2),n_j} \geq ... \geq \hat{\mu}_{\sigma_j(n+1-j),n_j}$. For $1 \leq r \leq n+1-j$, we have these empirical gaps $\hat{\Delta}_{\sigma_j(r),n_j} =$ $\begin{cases} \hat{\mu}_{\sigma_j(r),n_j} - \hat{\mu}_{\sigma_j(k(j)+1),n_j} & \text{if } r \leq k(j) \\ \hat{\mu}_{\sigma_j(k(j)),n_j} - \hat{\mu}_{\sigma_j(r),n_j} & \text{if } r \geq k(j)+1 \end{cases}$;
3. Let $i_j \in argmax \hat{\Delta}_{i,n_j}$ and ties are broken arbitrarily. Deactivate arm a_{i_j} and $\mathcal{A}_{j+1} = \mathcal{A}_j \backslash \{a_{i_j}\}$;
4. If $\hat{\mu}_{i_j,n_j} > \hat{\mu}_{\sigma_j(k(j)+1),n_j}$, then arm a_{i_j} is accepted, i.e., $k(j+1) = k(j) - 1$ and $a_{k-k(j+1)} = i_j$;

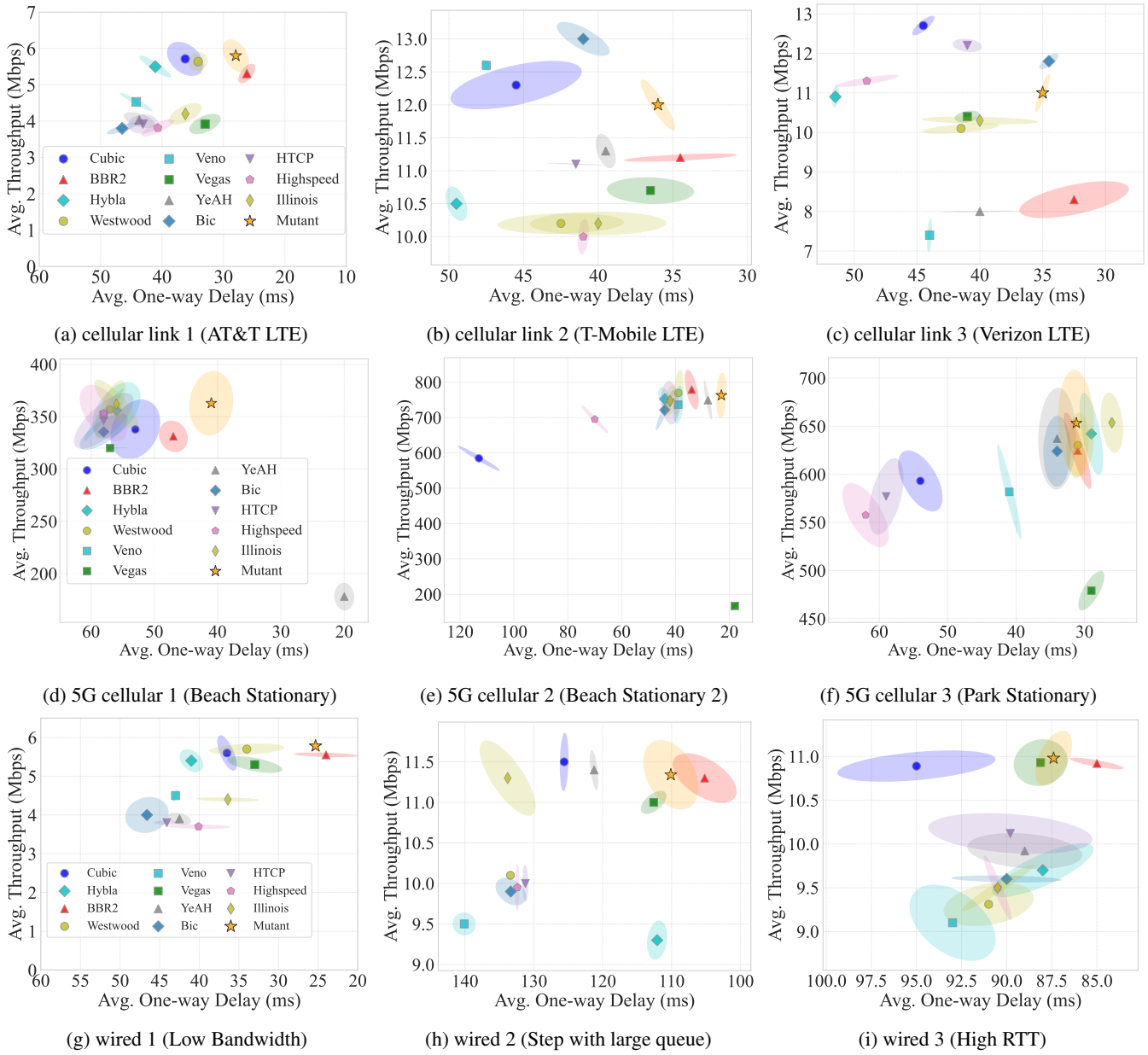
return the k accepted arms $a_1, ..., a_k$;

let $\hat{\mu}_{i,s} = \frac{1}{s} \sum_{t=1}^{s} X_{i,t}$ the empirical mean of arm i after s evaluations and $X_{i,s}(k)$ and $\hat{\mu}_{i,s}(k)$ the corresponding quantities in the multi-bandit problem.

The concept behind MPTS is similar to *Successive Rejects*, which was developed to address the single best arm identification problem [7]. However, MPTS has the additional feature of occasionally accepting an arm if it can be conclusively shown to be one of the top k arms.

Informally MPTS evolves following these steps. The time (the T rounds) is first divided into $K-1$ phases. At the end of each phase, the algorithm either accepts the protocol (arm) with the highest empirical mean or dismisses the protocol with the lowest empirical mean. In any of these two cases, the corresponding protocol is deactivated. During the following phase, it pulls each active protocol with the same frequency. During a specific phase j, the key to choosing whether to accept or reject is to rely on estimates for the gaps Δ_i^k.

Assuming, for example, that the algorithm has already accepted $k - k(j)$ protocols, $a_1, ..., a_{k-k(j)}$, it means that $k(j)$ must be still found. At the end of the phase j, MPTS computes for the $k(j)$ empirical best arms (among the active arms) the distance in terms of empirical mean to the $(k(j)+1)^{th}$ empirical best arm among the active arms. For the active arms not among the $k(j)$ empirical best arms, instead, MPTS computes the distance to the $k(j)^{th}$ empirical best arm. MPTS deactivates the arm i_j that maximizes these empirical distances. If i_j is the currently best (empirically) arm, it is accepted and



(a) cellular link 1 (AT&T LTE)

(b) cellular link 2 (T-Mobile LTE)

(c) cellular link 3 (Verizon LTE)

(d) 5G cellular 1 (Beach Stationary)

(e) 5G cellular 2 (Beach Stationary 2)

(f) 5G cellular 3 (Park Stationary)

(g) wired 1 (Low Bandwidth)

(h) wired 2 (Step with large queue)

(i) wired 3 (High RTT)

Figure 4: Average throughput and one-way delay over different emulated network environments.

variables are updated as $k(j+1) = k(j) - 1$, $a_{k-k(j+1)} = i_j$. Otherwise, the arm i_j is rejected. The formal description of MPTS is reported in Algorithm 2. We now show that the error probability of MPTS is bounded, with the following Theorem.

Theorem 5.1. *Let n be the number of available protocols, T the number of evaluations, and H_2^k the complexity measure defined in Equation 5. The error probability of the MPTS algorithm in the k-best protocol identification problem has the following error bound:*

$$e_T \leq 2n^2 \exp\left(-\frac{T-n}{8\overline{\log}(T)H_2^k}\right). \tag{6}$$

Proof. See appendix. □

6 Performance Evaluation

In what follows, we describe Mutant general evaluation using simulated and real-world traffic. In particular, in Section 6.1, we give details about the environments used for our experiments. In Section 6.2, we present Mutant's performance and compare them to state-of-art ML-based schemes. In Section 6.3, we analyze the importance of selecting the best-k protocols in the pool. We elaborate more about Mutant's fairness in Section 6.4. Finally, in Section 6.5 we present an

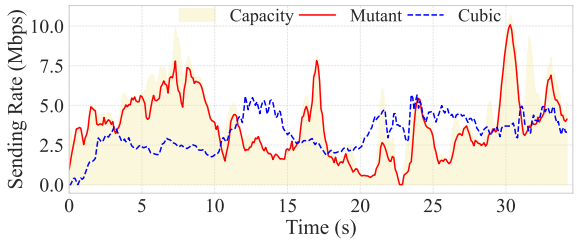


Figure 5: Mutant quickly adapts to highly variable network conditions compared to Cubic, the default congestion control in most Linux implementations. An optimal protocol would follow exactly the available capacity.

ablation study on Mutant's main components.

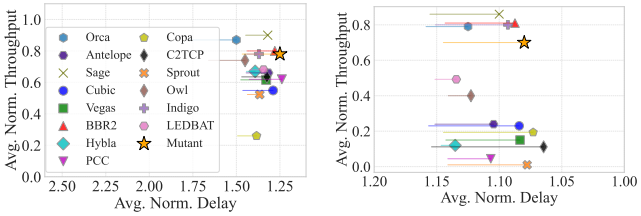
6.1 Evaluation Settings

Across our experiments, we test our protocol on publicly available realistic network conditions [2, 6, 37] to emulate *cellular* links even high bandwidths. Additionally, we consider a combination of emulated *wired* network scenarios, i.e., a typical Internet network environment consisting of a fixed bandwidth, delay, and queue size. We range the bandwidth, minimum rtt, and buffer queue size values within [12, 96] Mbps, [10, 120] ms, and [1, 20] $\times$ BDP, respectively.

We collect the performance of other CC baselines using Pantheon testbed [54] and Mahimahi emulator [37]. An experiment involves establishing an `iperf3` session between a single client and a Linux-based host machine. This setup allows us to capture and analyze various network metrics and evaluate the performance of different congestion control algorithms in a controlled experimental environment. For our real-world scenarios, we test Mutant on *(i)* intra-continental and *(i)* inter-continental network environments using Fabric [10] testbed. In our experimental setup, we establish the default quantity of kernel protocols within the pool, denoted as k, to be 6. This decision followed a comprehensive evaluation via a greedy search approach, as detailed in Section 6.3. Additionally, we set $\delta = 10^{-}2$s, i.e., the interval for alternation among CC schemes, (see Section 6.5).

6.2 Throughput-Delay Performance

In our experiments we compared Mutant with the following baseline, divided into four categories: (i) end-to-end TCP designs, e.g., Cubic [25], Vegas [13], Hybla [14] and the schemes in the pool, PCC [21], Copa [5]; (ii) end-to-end cellular, i.e., LTE protocols, e.g., C2TCP [1], Sprout [51]; (iii) ML-based protocols, such as Owl [43], Indigo [54], Orca [2], Sage [55], Antelope [56], and (iv) mixed schemes, e.g., LEDBAT [45]. Figure 4 shows the average throughput and delay across the cellular and wired sets of experiments, where the shaded ellipses represent the standard deviation across experiments from the mean value (i.e., the center of the ellipses). In particular, the *step* scenario (Figure 4h) is obtained by doubling the bandwidth after a defined time period. We compare Mutant against the pool of schemes it learns from. We select different emulated network environments for wired and cellular experiments and run each protocol for 2 minutes. We selected a diverse set of network scenarios with different ranges of bandwidth, from low (i.e., LTE and wired), to high[2] (i.e., 5G cellular). For each environment, we preliminary compute the protocols in the pool by running MPTS with $T = 100$, then running Mutant with the resulting pool. Results prove that Mutant is consistently among the best protocols across all tested scenarios and learns how to adapt to the specific environment. Somehow surprisingly, these findings show that Mutant does not depend on the best-performing protocol but can learn its policy by switching among the protocols pool and adapting quickly to the network scenario to maximize its reward. Indeed, Mutant can learn from the pool of protocols and outperform them. In Figure 5, we present how Mutant's best-k setting can quickly adapt to changing network conditions, while Cubic clearly shows the inability to achieve full link utilization. This shows that Mutant learns to select the protocols based on when they perform at their best over time, even in highly variable network scenarios. We explore further the reward suboptimality in Appendix B.



(a) Intra-Continental (b) Inter-Continental

Figure 6: Average normalized throughput, average normalized delay and 95^{th} percentile (end of lines) on real-world experiments.

We then validate Mutant on real-world network environments. We consider intra-continental and inter-continental experiments by setting up different hosts across the US and Europe using Fabric testbed [10]. To make a fair comparison with the ML baselines, we preliminary run the MPTS algorithm for the top-k protocols on every scenario. For each baseline, we run one flow for 30 seconds and repeat the experiment 10 times. As demonstrated in Figure 6, Mutant consistently achieves high performance levels when compared to other baselines. Most notably, Mutant can surprisingly match or even outperform pre-trained ML-based congestion control protocols (such as Orca, Sage, Antelope, and Indigo), even without extensive offline training. Mutant can also outperform online learning schemes, such as PCC, which show a competitive delay but a low throughput. We also highlight

[2]Due to Mahimahi limitations on traces with bandwidths larger than $\sim$ 200 Mbps, we leveraged a modified version of Mahimahi [6] to test Mutant on real 5G cellular traces with high bandwidth.

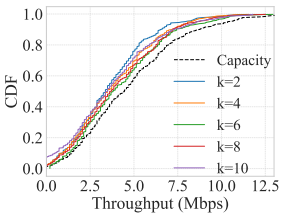


Figure 7: CDF of the throughput over a cellular link varying k.

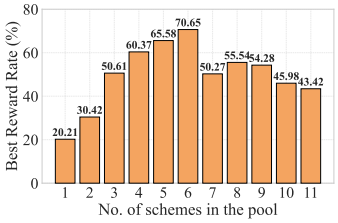


Figure 8: Mutant's performance varying the number of protocols in the pool (k).

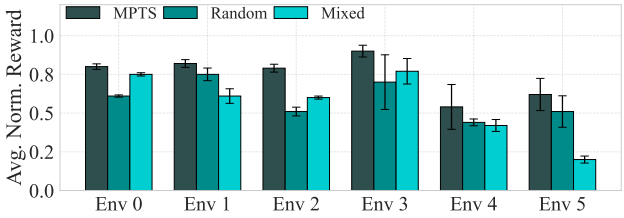


Figure 9: Impact of different protocol selection strategies.

that Mutant is capable of achieving competitive performance even within a short duration of the flow transmission. Overall, Mutant shows a 3.85% lower delay than BBR2 (appearing as the best-performing protocol in the pool) and a 3.60% lower delay than the average delay of other machine learning-based solutions like Sage, Orca, Indigo, and Antelope. Notably, it can achieve this result while maintaining a high throughput. This analysis yields an important result: *it is possible to learn from a pool of protocols in an online fashion and achieve competitive performance compared to offline ML algorithms.*

6.3 MPTS: Top-k Evaluation

To identify the optimal k for our Mutant Congestion Control protocol, we employ our MPTS Algorithm 2 while varying the number of protocol k in the pool, which also defines the action space in the RL model. Empirically, we observe that the inclusion or exclusion of certain protocols significantly influences the performance of the model, leading to either performance improvement or deterioration. Consequently, it is crucial to determine the best-performing configuration, where the k-size "team" in the algorithm effectively represents the protocols that positively contribute to the overall performance of Mutant.

We run an experiment to empirically find the value of k to be set for each network scenario. We run MPTS for a total of $T = 100$ rounds, varying k between 2 and 11. During each round of the MPTS algorithm, the available protocols are selected based on a uniform distribution. This selection process ensures that each protocol has an equal chance of being evaluated. Upon completing the T total rounds of the MPTS algorithm, we compute the best k protocols based on their cumulative mean reward, as defined in Equation 1. Intuitively, the higher the number of protocols in the pool, the more time Mutant must explore to find the best protocol at a given step of the algorithm. We evaluate how the number of protocols in the pool impacts the performance of Mutant on a cellular link in terms of throughput over time, reporting results in Figure 7. While it benefits the exploration of more protocols, we found that with more than $k = 8$, performance degrades without improving, even with more time to explore.

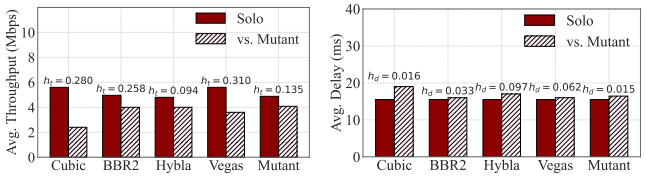
We then test each resulting configuration across all network scenarios and collect the average throughput and delay. We present the overall performance in Figure 8 in terms of *best reward rate*, namely the number of times Mutant results in the top 10% of the best-performing protocols for a given network environment.

Finally, to underline the importance of selecting the best-k protocol preliminary (via MPTS), we run the following experiment: we consider six different network environments sampled from the sets in Section 6.1, and three protocol selection strategies: (i) MPTS with $k = 6$, (ii) a random selection of $k = 6$ initial CC schemes and (iii) a mixed selection of CC schemes for each of the following categories: delay-based (Vegas, BBR2), loss-based (Cubic, Westwood), hybrid (Illinois) and specialized (Highspeed, Hybla). Subsequently, we run Mutant with the resulting pool of CC schemes for each selection strategy for 2 minutes on the six network scenarios. In Figure 9, we present the average normalized reward (Equation 1) for the three selection strategies. Interestingly, the selection of the initial pool of schemes significantly impacts the performance of our model. When a diverse set of CC schemes (i.e., protocols of different natures) is considered, Mutant shows a degradation of performance. The takeaway message is that *the efficacy of Mutant is significantly enhanced by an adequate selection of the starting protocols for the specific network environment, and our MPTS algorithm enables consistently high performance across the test scenarios.*

6.4 Fairness and Harm Analysis

In this section, we evaluate the *harm* [50] that a *Mutant* flow, α, causes another flow, β, considered as another congestion control algorithm, e.g., Cubic. As mentioned by Ware et al. [50], *harm* instead of *fairness* is a better performance metric. If the amount of harm caused by flows using a new algorithm γ on flows using an algorithm ω is within a bound derived from how much harm ω flows cause other ω flows, we can consider γ deployable alongside ω. Therefore, we use this metric to show that *Mutant* can be deployed to real-world systems.

Harm, unlike *fairness*, takes into account that different flows can have different demands. As such, for example, a new CC scheme γ competing against a *Reno* flow on a 10Gbps link would not be flagged as *unfair* because it took advantage of the remainder of the link capacity due to the well-known *Reno's* slow additive increase and aggressive reaction to loss [26], which prevents *Reno* from fully utilizing the link. Also, it is



(a) Throughput. (b) Delay.

Figure 10: Mutant's *harm* [50] to other protocols and *self-harm* (right-most bars) is consistently low, meaning that the protocol *fairness* and *friendliness* are high when competing with other protocols or against other Mutant instances.

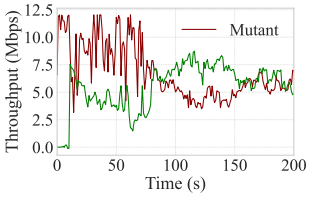


Figure 11: Friendliness competing with a Cubic flow.

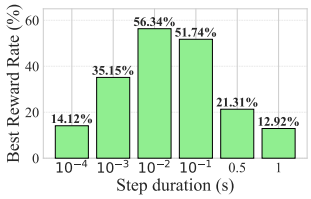


Figure 12: Impact of the step duration δ.

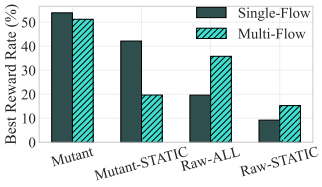


Figure 13: Impact of different groups of input features.

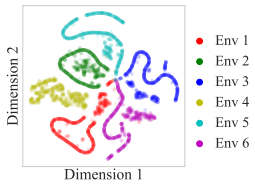


Figure 14: t-SNE [49] of the embeddings extracted from the last layer of the encoder.

multi-metric. It is not based solely on throughput unlike Jain's Fairness Index (JFI) [27]. On the contrary, it also considers performance metrics like latency, flow completion time, etc.

Much like *fairness*, *harm* ranges from [0 - 1] where 1 is maximally harmful and 0 is harmless. We use the following equations to calculate the *harm*, h, caused by *Mutant's* on another CC scheme. For metrics where "more is better" (like throughput), we use $h_t = \frac{x-y}{x}$, where x is the solo performance of β without α and y is the performance of β after the introduction of α. While for metrics where "less is better" (like latency or loss), we use $h_d = \frac{y-x}{y}$ to calculate *harm*.

We evaluated the harm of *Mutant* when competing with other flows. In Figure 10, we report the performance in terms of throughput and delay, where Mutant is used in competition with different CCAs. We also assess the protocol's self-harm. In particular, we tested some protocols in Mutant's pool on an emulated cellular link. First, we run each CC scheme individually in a single-flow scenario (i.e., without interference from Mutant's flow). Then, we run Mutant concurrently with a flow using a different CC scheme to observe their competition. Our results reveal that *Mutant* is friendly (i.e., causes minimal harm) to all the other protocols, with a negligible influence on the flows of other protocols, by just looking at the h_t and h_d values. From these experiments, we can see how, in addition to being friendly, Mutant also has high self-fairness i.e., it minimally harms other Mutant instances.

Additionally, we provide a sample of the Mutant's behavior when competing with a Cubic's flow in Figure 11 on a cellular link. It is worth noticing how Mutant learns to adapt to the new flow and converge to a fair share over time. This is due to the fact that even though Equation 1 does not explicitly include a fairness term, Mutant has the capability to penalize any unfair protocol behavior and promptly switch to an alternative one in response to a sudden surge in packet loss - which is accounted for in the formulation. Over time, Mutant converges to the fair share of the link capacity to maximize the reward.

6.5 Ablation Study

In this section, we validate some of the main Mutant's components and their impact on its overall behavior. First, as Mutant's switching logic is a core feature of the system, we study the impact of different values of δ (i.e., the switching frequency). Second, we evaluate the importance of input features in the online learning process. In particular, we validate the impact of mapping the network features into the latent space. Third, we study the encoder's ability to identify different network conditions.

Impact of switching frequency. We start analyzing the performance of different values of the step duration δ, i.e. the time interval a CC scheme in the pool runs before the next action of the RL algorithm is performed. We report in Figure 12 the *best reward rate* across all our network test environments - i.e., the number of times Mutant results in the top 10% of the best-performing protocols for a given environment. As emerging from the graph, *Mutant benefits from high-frequency transitions between schemes in the pool, instead of selecting a single protocol for extended duration.*

Input Features. To evaluate the effect of different input feature strategies, we run an experiment for single and multi-flow environments to examine the following input feature configurations: Mutant-STATIC, Raw-ALL, and Raw-STATIC. Mutant-STATIC excludes all mean, minimum, and maximum statistics from the input feature set. Raw-ALL removes the encoder from Mutant's workflow and directly feeds the network features into the learning model. Raw-STATIC eliminates both the encoder and the mean, minimum, and maximum statistics. Figure 13 exhibits that Mutant benefits from input features represented in the latent space. These results also reflect Mutant's behavior when only a partial set of features is input to the encoder: indeed, without any historical information given by the different observation windows, there is a clear performance degradation.

Quality of the embeddings. We present a t-Distributed Stochastic Neighbor Embedding (t-SNE) [49] analysis (a

popular technique employed to visualize high-dimensional data) to validate further the importance of encoding the input network features in Mutant. Figure 14 shows such a two-dimensional t-SNE representation of the embeddings in six random network scenarios sampled from the test set indicated in Section 6.1. This visualization highlights that our encoder sharply separates different network scenarios in the latent space, helping Mutant detect diversities among states.

7 Related Work

The problem of congestion control has been extensively discussed in the literature, given its importance for reliable data transmissions. While congestion control has been a popular research topic for over 30 years, new recent solutions have employed machine learning-based algorithms to overcome the limitations of traditional approaches. In the following, we describe how machine learning (ML) has been helpful and how we differ from these recent solutions.

Since TCP congestion control is a fundamental service, it is not surprising that significant improvements and variations have been proposed over the years. Among them, we can mention TCP Vegas [13], Fast [30], BBR [15], and Compound [47]. Several other protocols focus on data centers, e.g. Data Center TCP (DCTCP) [3], but those are not the focus of our approaches. Data centers have been recently departing from TPC-based solutions, moving towards service meshes. Past congestion control protocols have been classified according to the metric they consider. Delay-based protocols consider *rtt* and/or average delivery rate measurements to decide how fast data should be sent over the network. BBR [15], e.g., belongs to such class and is considered to be resistant to the bufferbloat problem, but it frequently exceeds the link capacity, causing excessive queuing delays. On the other hand, loss-based protocols rely on indications of lost packets, and most of the traditional approaches, e.g., Compound [47] and Cubic [25], rely on some predefined functions or rules to handle network conditions.

However, all these solutions are limited by their nature of fixed-rule strategies, and their performance is limited in networks that require rapid adaptations. To address this problem, a plethora of online learning congestion control protocols have appeared, e.g., Remy [52], Performance-oriented Congestion Control (PCC) [21], PCC-Vivace [22], Copa [5]. After defining an objective function to be optimized, the process consists of learning the best actions to take either periodically or upon receiving ACKs.

This learning process can be further optimized by the advances in ML, and in particular in reinforcement learning (RL). RL has permeated many congestion control mechanisms, such as Orca [2], Owl [43], SPINE [48], and Aurora [28]. While Aurora is built upon the previous PCC protocol and then extended with a Deep-RL approach, Owl and Orca extend the Cubic implementation by adding an RL learning model that can work even in unexplored conditions. SPINE, instead, follows the additive-increase/ multiplicative-decrease (AIMD) approach of Cubic, but the parameters of the increment and decrement are the output of the RL model. As these papers, our protocol also leverages past solutions to combine the classical approach with advanced DRL techniques, but with a novel twist. Prior learning solutions suffer from design suboptimalities. If, on the one hand, delay-based protocols cannot avoid starvation, an extreme form of unfairness, in paths with excessive delay variations [4], loss-based ones suffer the bufferbloat problems. To solve this tussle, we bank on the idea there is no one-size-fits-all protocol, as learned from decades of designing congestion control protocols. Our design is based on the notion of learning from existing protocols, and predicting which protocol (set) would perform best in future seen network and congestion scenarios.

The idea of switching among available CCs has been floated before [38, 55, 56]. With them, we share the idea of learning from existing protocols; however, we differ in several ways from these solutions. Firstly, Mutant is not reliant on a data-driven offline procedure, meaning that its learned congestion control policy can adjust to different network conditions over time. Secondly, Mutant can adapt quickly to changes in the network environment with minimal offline adaptations, as its main functions run online. Finally, Mutant can smoothly transition between different CC protocols in the kernel space, regardless of the flow duration, while preserving past TCP information and states for each protocol during the change.

8 Conclusion

In this paper, we presented the design, implementation, and evaluation of *Mutant*, a congestion control scheme that learns from a pool of existing congestion control algorithms online. *Mutant*'s online learning algorithm understands which current state-of-the-art congestion control algorithm can perform well in the near future. Our evaluation shows that our solution can adapt to different network congestion scenarios efficiently if it has the correct pool of CC schemes to learn from. Perhaps surprisingly, we found that more knowledge in terms of protocol choices may hurt performance, given the explorative nature of the protocol. Mutant obtained high throughput and low delays more consistently than our benchmark CC protocols across different network conditions. Additionally, we evaluated *Mutant* fairness by measuring the harm index. Our results confirmed that *Mutant* is fair and causes negligent harm and self-harm to other flows.

Acknowledgment

This work has been supported by NSF awards #2201536 and #2133407. We would like to thank Princewill Okorie for his initial work on this project, our shepherd Dr. Yasir Zaki, the anonymous reviewers, and Dr. Gianni Antichi for his feedback on a preliminary version of this manuscript.

References

[1] Soheil Abbasloo, Tong Li, Yang Xu, and H Jonathan Chao. Cellular controlled delay tcp (c2tcp). In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 118–126. IEEE, 2018.

[2] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, page 632–647, 2020.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '10)*, page 63–74, 2010.

[4] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '22)*, pages 177–192, 2022.

[5] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, 2018.

[6] Rohail Asim, Muhammad Khan, Luis Diez, Shiva Iyer, Ramon Aguero, Lakshmi Subramanian, and Yasir Zaki. ZEUS: An Experimental Toolkit for Evaluating Congestion Control Algorithms in 5G Environments. *arXiv preprint arXiv:2208.13985*, 2022.

[7] Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. Best arm identification in multi-armed bandits. In *Proceedings of the 23rd Annual Conference on Learning Theory (COLT)*, pages 41–53, 2010.

[8] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

[9] Andrea Baiocchi, Angelo P Castellani, Francesco Vacirca, et al. Yeah-tcp: yet another highspeed tcp. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

[10] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. FABRIC: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing*, 23(6):38–47, 2019.

[11] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM, 2007.

[12] Christopher M Bishop. Pattern recognition and machine learning. *Springer google schola*, 2:645–678, 2006.

[13] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[14] Carlo Caini and Rosario Firrincieli. Tcp hybla: a tcp enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, 2004.

[15] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control. *Queue*, 14(5):20–53, 2016.

[16] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. TCP BBR v2 Alpha/Preview Release. https://github.com/google/bbr/blob/v2alpha/README.md, 2019.

[17] Claudio Casetti, Mario Gerla, Saverio Mascolo, Medy Y Sanadidi, and Ren Wang. Tcp westwood: end-to-end congestion control for wired/wireless networks. *Wireless Networks*, 8:467–479, 2002.

[18] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[19] Shay B. Cohen, Gideon Dror, and Eytan Ruppin. Feature selection via coalitional game theory. *Neural Computation*, 19:1939–1961, 2007.

[20] Zihan Ding. Imitation learning. In Shanghang Zhang Hao Dong, Zihan Ding, editor, *Deep Reinforcement Learning: Fundamentals, Research, and Applications*, chapter 8, pages 273–306. Springer Nature, 2020.

[21] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, Oakland, CA, 2015.

[22] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. Pcc vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, 2018.

[23] Sally Floyd. Highspeed tcp for large congestion windows. Technical report, 2003.

[24] Cheng Peng Fu and Soung C Liew. Tcp veno: Tcp enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

[25] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[26] Janey C Hoe. Improving the start-up behavior of a congestion control scheme for tcp. *ACM SIGCOMM Computer Communication Review*, 26(4):270–280, 1996.

[27] Raj Jain, Dah-Ming Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998.

[28] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 3050–3059. PMLR, 2019.

[29] Huiling Jiang, Qing Li, Yong Jiang, GengBiao Shen, Richard Sinnott, Chen Tian, and Mingwei Xu. When machine learning meets congestion control: A survey and comparison. *Computer Networks*, 192:108033, 2021.

[30] Cheng Jin, David X Wei, and Steven H Low. FAST TCP: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4. IEEE, 2004.

[31] Mehdi Kargar and Aijun An. Discovering Top-k Teams of Experts with/without a Leader in Social Networks. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM '11)*, pages 985–994. ACM, 2011.

[32] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On bayesian upper confidence bounds for bandit problems. In *Artificial intelligence and statistics*, pages 592–600. PMLR, 2012.

[33] Douglas Leith. H-TCP protocol for High-speed Long Distance Networks. In *Proc. International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet '04)*, 2004.

[34] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[35] Shao Liu, Tamer Başar, and Ravi Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodologies and tools*, pages 55–es, 2006.

[36] Tyler Lu, Dávid Pál, and Martin Pál. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pages 485–492. PMLR, 2010.

[37] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[38] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic TCP Initial Windows and Congestion Control Schemes Through Reinforcement Learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[39] Zhiyuan Pan, Jianer Zhou, XinYi Qiu, Weichao Li, Heng Pan, and Wei Zhang. Marten: A built-in security drl-based congestion control framework by polishing the expert. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, 2023.

[40] Carlos Riquelme, George Tucker, and Jasper Snoek. Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling. *arXiv preprint arXiv:1802.09127*, 2018.

[41] Walid Saad, Zhu Han, Mérouane Debbah, Are Hjorungnes, and Tamer Basar. Coalitional game theory for communication networks. *IEEE signal processing magazine*, 26(5):77–97, 2009.

[42] Alessio Sacco, Matteo Flocco, Flavio Esposito, and Guido Marchetto. Partially Oblivious Congestion Control for the Internet via Reinforcement Learning. *IEEE Transactions on Network and Service Management*, 20(2):1644–1659, 2022.

[43] Alessio Sacco, Flocco Matteo, Flavio Esposito, and Guido Marchetto. Owl: Congestion control with partially invisible networks via reinforcement learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[44] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

[45] Stanislav Shalunov, Greg Hazel, Jana Iyengar, and Mirja Kühlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817, December 2012.

[46] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[47] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–12. IEEE, 2006.

[48] Han Tian, Xudong Liao, Chaoliang Zeng, Junxue Zhang, and Kai Chen. Spine: an efficient drl-based congestion control with ultra-low overhead. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22)*, pages 261–275. ACM, 2022.

[49] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[50] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*, pages 17–24. ACM, 2019.

[51] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of the 10th USENIX NSDI*, page 459–472, USA, 2013.

[52] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.

[53] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524. IEEE, 2004.

[54] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: The Training Ground for Internet Congestion-Control Research. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 731–743, USA, 2018. USENIX Association.

[55] Chen-Yu Yen, Soheil Abbasloo, and H Jonathan Chao. Computers Can Learn from the Heuristic Designs and Master Internet Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 255–274. ACM, 2023.

[56] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. Antelope: A framework for dynamic selection of congestion control algorithms. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

Appendix

A MPTS Proof

Proof Sketch of Theorem 5.1.

Proof. Let's consider the event ξ defined as:

$$\xi = \{\forall i \in \{1,...,n\}, j \in \{1,...,n-1\}, \left|\frac{1}{n_j}\sum_{s=1}^{n_j} X_{i,s} - \mu_i\right| \leq \frac{1}{4}\Delta^i_{(n+1-j)}\}; \quad (7)$$

by Hoeffding's inequality and a union bound, the probability of the complementary event $\overline{\xi}$ is bounded by inequality:

$$\begin{aligned}\mathbb{P}(\overline{\xi}) &\leq \sum_{i=1}^{n}\sum_{j=1}^{n-1}\mathbb{P}\left(\left|\frac{1}{n_j}\sum_{s=1}^{n_j} X_{i,s} - \mu_i\right| > \frac{1}{4}\Delta^i_{(n+1-j)}\right) \\ &\leq \sum_{i=1}^{n}\sum_{j=1}^{n-1} 2\exp\left(-2n_j\left(\frac{\Delta^i_{(n+1-j)}}{4}\right)^2\right) \\ &\leq 2n^2\exp\left(-\frac{T-n}{8\overline{\log}(T)H_2^k}\right).\end{aligned} \quad (8)$$

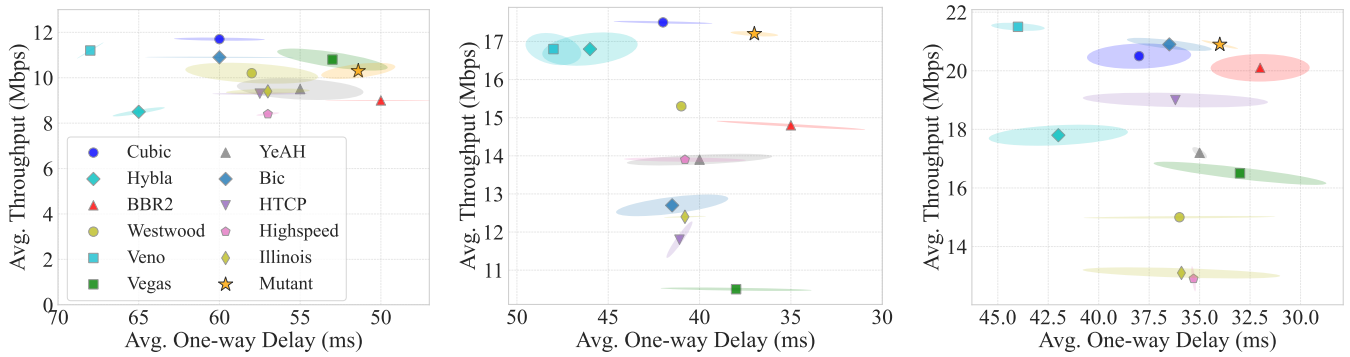
The last inequality is a direct conclusion from the fact that:

$$\begin{aligned}n_j\left(\Delta^i_{(n+1-j)}\right)^2 &\geq \frac{T-n}{\overline{\log}(T)(T+1-j)\left(\Delta^i_{(n+1-j)}\right)^{-2}} \\ &\geq \frac{T-n}{\overline{\log}(T)H_2^k}.\end{aligned} \quad (9)$$

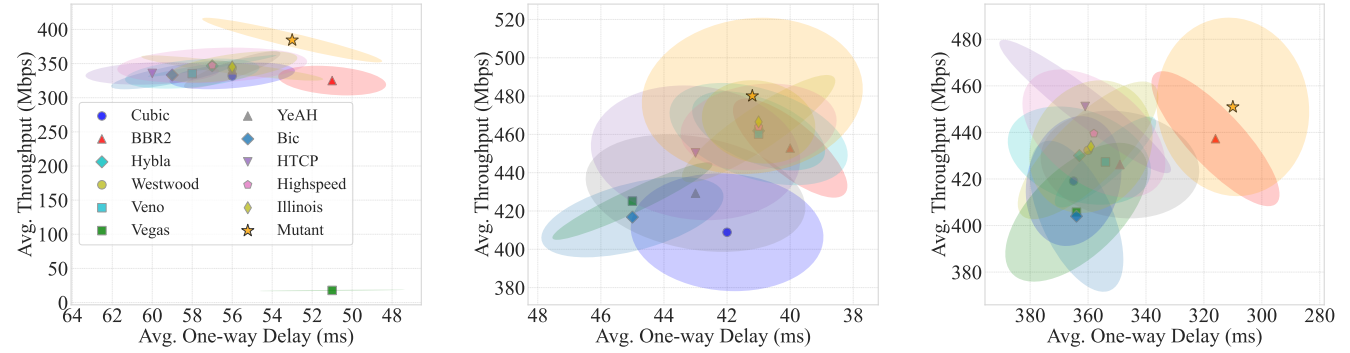
It is therefore sufficient to show that on event ξ, the algorithm does not make any error. This conclusion can be proved by induction on j. □

B More on Mutant's Performance

Mutant demonstrates higher performance across a wide range of network conditions in terms of both throughput and delay. We extended our evaluation to include additional cellular traces (Figure 15), encompassing both LTE and 5G networks with different ranges of bandwidth, where data are sampled from [2] and [6], respectively. Mutant consistently ranks among the top-performing protocols across diverse scenarios and shows its ability to adapt and maintain optimal performance.

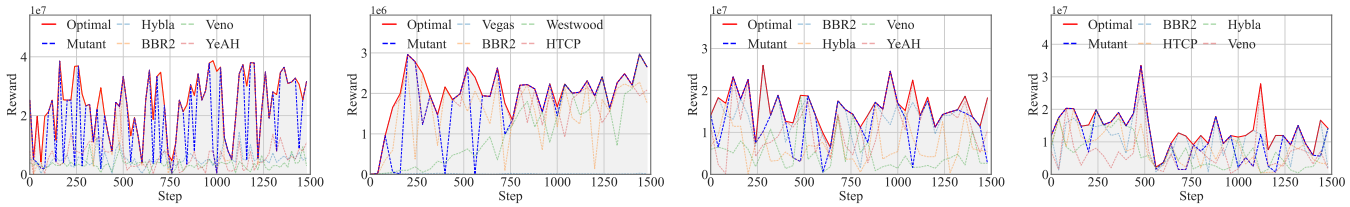


(a) cellular link 4 (Walking in Times Square) (b) cellular link 5 (Bus Riding) (c) cellular link 6 (Stationary Times Square)



(d) 5G cellular link 4 (Corniche) (e) 5G cellular link 5 (Corniche Walking) (f) 5G cellular link 6 (City Drive)

Figure 15: Average throughput and one-way delay over different real cellular network traces.



(a) 5G Cell. (Beach Stat.) (b) 5G Cell. (Beach Stat. 2) (c) 5G Cell. (City Drive) (d) 5G Cell. (Park Stationary)

Figure 16: Performance comparison of Mutant against individual protocols in four distinct 5G cellular scenarios. Mutant's reward performance is also against an idealized system that always chooses the "optimal" protocol.

In Figure 16, we report the reward of our Mutant protocol across four distinct high bandwidth 5G cellular scenarios. The reward function follows Equation 1. We compare Mutant's reward against an optimal system, which selects the best-performing protocol at each step of the run, and the reward of each of the CCAs in the pool. The pool of best-performing protocols is selected via the MPTS algorithm with $K = 4$. Our Mutant model was then trained to convergence and subsequently tested on network traces for 1500 steps and $\delta = 10^{-2}$ (see Section 6.1) in each scenario. We observe that Mutant, consistently achieves high rewards, demonstrating its adaptive capability by dynamically selecting the best-performing protocol at each step and for every network trace It is worth noting that the occasional dips in Mutant's performance, which correspond to intentional exploration (hence suboptimal) steps, are the only instances where it deviates from the optimal choice. However, these exploration phases enable Mutant to continually evaluate and adapt to potential changes in network conditions.

CATO: End-to-End Optimization of ML-Based Traffic Analysis Pipelines

Gerry Wan[1] Shinan Liu[2] Francesco Bronzino[3] Nick Feamster[2] Zakir Durumeric[1]
[1]*Stanford University* [2]*University of Chicago* [3]*ENS Lyon*

Abstract

Machine learning has shown tremendous potential for improving the capabilities of network traffic analysis applications, often outperforming simpler rule-based heuristics. However, ML-based solutions remain difficult to deploy in practice. Many existing approaches only optimize the predictive performance of their models, overlooking the practical challenges of running them against network traffic in real time. This is especially problematic in the domain of traffic analysis, where the efficiency of the serving pipeline is a critical factor in determining the usability of a model. In this work, we introduce CATO, a framework that addresses this problem by jointly optimizing the predictive performance and the associated systems costs of the serving pipeline. CATO leverages recent advances in multi-objective Bayesian optimization to efficiently identify Pareto-optimal configurations, and automatically compiles end-to-end optimized serving pipelines that can be deployed in real networks. Our evaluations show that compared to popular feature optimization techniques, CATO can provide up to 3600× lower inference latency and 3.7× higher zero-loss throughput while simultaneously achieving better model performance.

1 Introduction

Machine learning (ML) models have grown to outperform traditional rule-based heuristics for a variety of traffic analysis applications, such as traffic classification [36, 44], intrusion detection [73], and QoE inference [15, 47]. Over the past few years, researchers have explored various approaches to developing more accurate models, ranging from better feature selection to employing sophisticated model types and traffic representations [4, 12, 25, 29, 37, 46, 51, 55, 57, 62, 76, 81]. However, the predictive performance of ML-based solutions often overshadows an equally critical aspect—the end-to-end efficiency of the serving pipeline that processes network traffic and executes the model.

For traffic analysis, a significant challenge lies not just in developing accurate models, but in meeting the performance demands of the network. Many network applications must operate in real time with sub-second reaction times and/or process hundreds of gigabits per second of traffic without packet loss [72]. Unfortunately, models developed without consideration of the associated systems costs of serving them in real networks often turn out to be unusable in practice [14]. Current approaches to this problem typically rely on lightweight models [43], programmable hardware [36, 74], or early inference techniques [11, 54], but many of these unnecessarily compromise on predictive performance [16, 64, 74].

Recent studies have stressed the need to balance both the systems costs and predictive performance of ML-based traffic analysis solutions [14, 64]. However, achieving this balance is difficult. The end-to-end latency and throughput of a serving pipeline, which includes packet capture, feature extraction, *and* model inference, are difficult to approximate without in-network measurements. Furthermore, the search space over optimal feature representations is exponential in the number of candidate traffic features, and also depends on how far into a flow to wait before making a prediction. The added complexity of not just considering one objective, but two, makes end-to-end optimization of such systems an open challenge.

In this work, we present CATO, a generalizable framework that systematically optimizes the systems costs and model performance of ML-based traffic analysis pipelines. We start by formalizing the development of ML models for traffic analysis as a *multi-objective* optimization problem. We then combine multi-objective Bayesian optimization, tailored specifically for traffic analysis, with a realistic pipeline profiler to efficiently construct end-to-end optimized traffic analysis pipelines. CATO simultaneously searches over the selected features and the amount of captured traffic needed to compute those features, factors which have been shown to significantly impact both efficiency and predictive performance [14, 34]. During this search, CATO performs direct end-to-end measurements on the resulting serving pipelines to both *optimize* and *validate* their in-network performance.

We evaluate CATO on live network traffic and offline traces across a range of classification and regression traffic analy-

sis tasks. Our experimental results show that compared to popular feature optimization methods, CATO can reduce the end-to-end latency of the serving pipeline by up to 3600×, from several minutes to under 0.1 seconds, while simultaneously improving model performance. Additionally, CATO can increase zero-loss classification throughput by up to 3.7×.

We hope that our work helps to realize the potential impact of using machine learning to manage and improve networks. Code is available at: https://github.com/stanford-esrg/cato.

2 Background and Motivation

The networking community has long attempted to use machine learning (ML) to perform traffic analysis tasks like QoE inference [15, 26, 39, 47, 48], traffic classification [36, 44, 65], intrusion detection [73], and load balancing [18]. As traffic increasingly becomes encrypted, ML has also been shown to be a promising technique for understanding otherwise opaque network traffic, replacing traditional deep packet inspection and other rule-based heuristics [12, 53, 63].

While significant progress has been made in improving the predictive capabilities of models used for traffic analysis [12], the real-world deployability of a model is based not just on conventional notions of ML performance (e.g., accuracy, F1 score), but also on the associated systems-level performance (e.g., latency, throughput) of the entire serving pipeline [14]. Given the real-time demands of network operations, any applications that rely on ML must nonetheless operate within tight performance budgets. Even small delays can cause substantial packet loss and render a model ineffective [15, 31], making systems performance even more crucial for traffic analysis. As a result, ML-based traffic analysis cannot solely target high predictive performance—the end-to-end efficiency of the entire serving pipeline must be jointly optimized as well.

2.1 ML-Based Traffic Analysis

ML-based traffic analysis typically begins with the ingestion of raw traffic and ends with a prediction of a traffic property, such as a service quality metric. While traffic analysis applications are diverse, we focus on the class of problems that involves per-flow or per-connection inference, such as traffic/device classification, QoE inference, or intrusion detection. These applications typically make a prediction about an entire flow or connection, then initiate an action such as triggering an alert, blocking or rerouting the flow, or performing further analysis downstream (Figure 1).

Traffic inference extends beyond merely executing the model; it also involves packet capture, connection tracking, flow reassembly, and feature extraction. Raw traffic undergoes multiple operations, including header parsing, computation, and encoding before arriving at the representation that is used as input to the model. The final model inference step makes the prediction, with its predictive performance determined

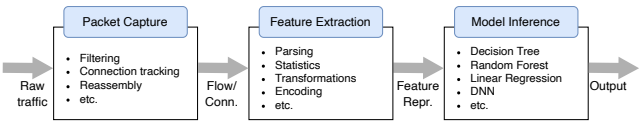


Figure 1: A typical serving pipeline for ML-based traffic analysis. Usability of a model hinges on both its predictive accuracy and the systems performance of the entire pipeline.

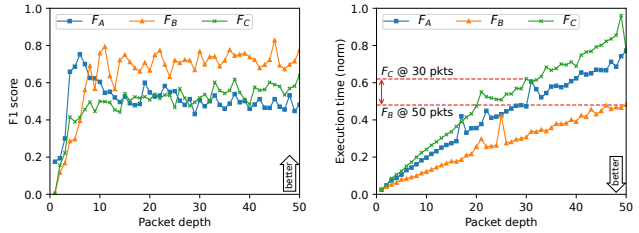
by the model type (e.g., random forest, neural network), the computed features, and the amount of data captured from the flow. The end-to-end systems performance depends on *all* of these aspects together.

For traffic analysis in particular, the choice of features computed from the network traffic is often as important, if not more so, than the model itself [29, 34, 55]. While many previous works have focused on the model inference stage [43, 46, 57, 64, 74, 81], design decisions made in the earlier stages of the serving pipeline are crucial to its performance and practicality, and warrant careful consideration.

Optimizing Predictive Performance. Many techniques have been proposed to accurately make predictions about network traffic. These approaches range from popular feature selection methods that choose highly predictive features based on summary statistics [51], packet lengths [11], timing [37], and/or frequency domain characteristics [25], to new techniques like GGFAST [55], which generates specialized "snippets" to classify encrypted flows.

Following advancements in domains like computer vision and natural language processing, network researchers have also proposed sophisticated deep learning models [46, 57, 81] and traffic representations [29, 62] designed to further optimize the predictive performance of traffic analysis applications. However, most of these machine learning techniques are evaluated using offline packet traces on metrics like accuracy, precision, recall, or F1 score, overlooking the need to both *optimize* and *validate* their in-network systems performance. As a result, ML-based traffic analysis solutions that have been demonstrated to have high accuracy in controlled laboratory experiments often turn out to be unusable in real-time deployments because of the systems costs associated with running them [14].

Optimizing Serving Efficiency. To address the systems requirements of traffic analysis on modern networks, some works propose using lightweight models [43, 74] or choose features that reduce model inference time [69]. However, these techniques can over-compromise on predictive performance for speed, and often overlook the efficiency of other pieces of the serving pipeline like packet capture and feature extraction. Other approaches aim to optimize serving efficiency by making predictions as early as possible [10, 11, 21, 60]. While many traffic analysis solutions implicitly rely on the entire network flow or connec-



(a) Packet Depth vs. F1 Score. The best feature sets differ at varying packet depths.

(b) Packet Depth vs. Exec. Time. It can be cheaper to extract low-cost features at greater depths.

Figure 2: Effects of different (feature set, packet depth) configurations on F1 score and execution time. We highlight the size and complexity of the search space.

tion [7, 9, 15, 40, 62], these "early inference" techniques make predictions after observing a predefined number of packets. However, there is no ideal *packet depth* (i.e., the number of packets to use from any given flow) that is universally effective across applications. Choosing an appropriate value typically requires prior domain-specific knowledge or resorting to manual trial-and-error [54]. Consequently, existing works that do explicitly choose a packet depth often opt for values such as 10, 50, or 100 packets with little justification [5, 29, 30, 45, 50, 55, 56, 64]. As we will show in Section 5.2, this approach can miss significant opportunities for gains in both efficiency and model performance.

2.2 Challenge of End-to-End Optimization

Despite progress towards individually optimizing the predictive performance or serving efficiency of ML-based traffic analysis, improvements to either area in isolation often result in solutions that fail to achieve optimal combined model and systems performance objectives. Systematically designing traffic analysis pipelines that jointly optimize both of these objectives remains an open challenge.

We illustrate this challenge by attempting to design an IoT device classifier using the dataset published by Sivanathan et al. [65]. Our goal is to construct a serving pipeline that is *Pareto-optimal* across both its execution time and F1 score. In other words, it should not be possible to further reduce the execution time without also reducing the F1 score, or vice versa. We combine techniques from prior work by experimenting with different features and early inference, both of which have been shown (and we confirm) to significantly impact multiple aspects of the pipeline, including predictive performance, model inference time, and feature extraction time [14, 34, 51, 54]. For this example, we choose from six candidate flow features (Appendix A, Table 4) and vary the packet depth collected in each flow from 1 to 50, which is consistent with values used in prior work [11, 45, 54, 55]. We train the model, compile the complete serving pipeline, and exhaustively measure the F1 score and execution time for all $2^6 \times 50 = 3{,}200$ (feature set, packet depth) combinations.

As seen in Figure 2, F1 score and execution time vary with the chosen features and packet depth. For readability, we plot only three (labeled $F_A/F_B/F_C$) out of the 64 possible feature sets, but find qualitatively similar results across those not shown. We can see in Figure 2a that the best feature sets by F1 score differ dramatically at different packet depths. While F_A has the highest F1 score within the first 10 packets, the ranking flips at higher packet counts. Interestingly, the predictive performance of F_B and F_C increase with packet depth, whereas the opposite is true for F_A. In Figure 2b, we observe that for the *same* feature set, execution time generally increases with packet depth. However, the overall cost of waiting 50 packets to extract F_B is lower than the cost of waiting 30 packets for F_C. This reveals that having the flexibility to optimize the timing of feature extraction can significantly enhance serving efficiency, but is not always as straightforward as simply minimizing analyzed packets. If we look across both figures, we see that over-optimizing on execution time can also adversely affect F1 score and vice versa. The non-linear trade-offs between objectives further highlight the challenge in identifying Pareto-optimal solutions without exhaustive measurement.

The trade-offs between predictive performance and systems costs form a multi-dimensional and multi-objective search space that extends beyond merely identifying which features result in more accurate models. It also includes considerations for features that are efficient to extract, as well as how much data must be captured to compute and represent the features. While exhaustive measurement of end-to-end systems costs and model performance is feasible for just six candidate features, it quickly becomes impractical when scaling up to the dozens to hundreds of flow features typically considered by developers. In our example, it took 5 days to train, compile, and measure all 3,200 serving pipelines, but it would take over 7,000 years with 25 candidate features. The size and complexity of the search space, coupled with the need to consider and validate both model performance and systems cost objectives, makes end-to-end optimization challenging. Addressing this challenge is the central contribution of our work.

3 Cost-Aware Traffic Analysis Optimization

We introduce CATO (Cost-Aware Traffic Analysis Optimization), our solution for cost-aware ML-based traffic analysis optimization. The goal of CATO is to automatically construct traffic analysis pipelines that jointly minimize the end-to-end systems costs of model serving while maximizing predictive performance. At its core, CATO combines a multi-objective Bayesian optimization-guided search with a novel pipeline generator and feature representation profiler to produce serving pipelines suitable for deployment in real networks.

Symbol	Description
$\mathcal{F}$	Set of candidate network flow features
N	Maximum connection depth
$\mathcal{P}(\mathcal{F})$	Power set of $\mathcal{F}$
$\mathbb{X}$	Search space defined as $\mathbb{X} = \mathcal{P}(\mathcal{F}) \times N$
F	Set of features in a feature representation
n	Connection depth from which F is extracted
x	Feature representation $x = (F, n)$
$\texttt{cost}(x)$	Systems cost objective function
$\texttt{perf}(x)$	Predictive performance objective function
Γ	Set of Pareto-optimal solutions

Table 1: Summary of Variables

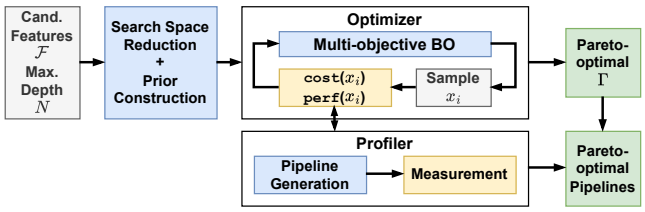


Figure 3: CATO combines a multi-objective BO-based Optimizer and a realistic pipeline Profiler to construct and validate efficient ML-based traffic analysis serving pipelines.

3.1 Problem Definition

CATO takes as input a set of candidate network flow features, denoted by $\mathcal{F}$. In line with conventional machine learning practice, these are typically derived from domain expertise or determined by the capabilities of the traffic collection tool. Common examples for traffic analysis include mean packet size, bytes transferred, connection duration, etc. [51], but can also include more complex features like frequency domain [25,77] and application-layer characteristics [15]. CATO also takes as input a maximum connection depth $N \in \mathbb{R}$, which serves as an *upper bound* on the amount of data in the connection that is considered for inference. Concretely, this can be the number of initial packets (i.e., packet depth), bytes, or time into the connection prior to feature extraction. These inputs define the *search space* $\mathbb{X} = \mathcal{P}(\mathcal{F}) \times N$, where $\mathcal{P}(\mathcal{F})$ is the power set of $\mathcal{F}$, i.e., the set of all possible subsets of $\mathcal{F}$, from which CATO selects feature representations.

A *feature representation* $x = (F, n)$ consists of a set of features $F \subseteq \mathcal{F}$ and a value $n \leq N$ that indicates the connection depth from which F is extracted. Each feature representation gives rise to a serving pipeline with an associated end-to-end systems cost and predictive performance, denoted by the functions $\texttt{cost}(x) : \mathbb{X} \to \mathbb{R}$ and $\texttt{perf}(x) : \mathbb{X} \to \mathbb{R}$. We note that these functions are general and can be user-defined according to the specific objectives of the traffic analysis problem. For instance, $\texttt{cost}(x)$ can refer to the end-to-end inference latency, execution time, (negative) throughput, etc., while $\texttt{perf}(x)$ can be defined as F1 score, accuracy, (negative) mean-squared-error, etc. We list a summary of variables in Table 1.

Multi-objective Optimization. We formalize the development of ML models for traffic analysis as a multi-objective optimization over the search space $\mathbb{X}$. The aim is to identify the Pareto front $\Gamma \subseteq \mathbb{X}$, which consists of all non-dominated points in $\mathbb{X}$. In other words, Γ contains the maximally desirable feature set / connection depth configurations, where no further improvement in systems cost or model performance can be achieved without compromising the other objective.

We deliberately choose a multi-objective optimization over a single-objective approach. Unlike a single-objective problem that aims to maximize model performance while satisfying system constraints, or vice versa, a multi-objective solution offers several advantages. The first is that the exact system and model performance requirements may not be known a priori (e.g., due to variable traffic rates or shared system resources), making it difficult to precisely define constraints. Second, if requirements change, a single-objective approach would necessitate redefining and rerunning the optimization with new objectives and constraints [68]. By expressing the problem as a multi-objective optimization, CATO identifies multiple Pareto-optimal solutions that each achieve a different balance between systems cost and model performance, providing the flexibility to accommodate changing application needs (e.g., adjusting accuracy thresholds or imposing new latency constraints) without re-optimization.

3.2 CATO Overview

CATO constructs end-to-end optimized traffic analysis pipelines according to the systems cost and model performance objective functions. It does so by efficiently identifying Pareto-optimal feature representations, and generating ready-to-deploy serving pipelines for a given model from those feature representations. Figure 3 depicts the high-level design, which consists of the Optimizer and the Profiler:

- The Optimizer takes the set of candidate features and maximum connection depth, and performs a multi-objective Bayesian optimization-guided search over the feature representation space. It periodically queries the Profiler for the systems costs and model performance of its sampled feature representations, which it uses to further refine the search for the Pareto front.
- The Profiler accepts queries from the Optimizer, compiles binaries for the end-to-end serving pipeline, and runs them to accurately measure $\texttt{cost}(x)$ and $\texttt{perf}(x)$. These measurements serve the dual purpose of guiding the Optimizer towards Pareto-optimal solutions and validating the in-network performance of the generated pipelines.

3.3 The CATO Optimizer

In general, measuring $\texttt{cost}(x)$ and $\texttt{perf}(x)$ for an arbitrary feature representation is computationally expensive. It involves generating the serving pipeline, training and evaluating the ML model, and measuring performance costs either through simulation or in physical testbeds. The massive size

of the search space and the computational cost of evaluating the objective functions precludes the possibility of exhaustively searching all possible configurations. To handle this intractability, CATO leverages Bayesian Optimization (BO), building on recent developments in multi-objective design space exploration [52] and sample-efficient BO [32] to efficiently estimate the Pareto front in $\mathbb{X}$.

Why Bayesian Optimization? Bayesian optimization is a technique designed for global optimization of black-box objective functions [35], and has seen success in domains like hyperparameter tuning [66, 70], compiler optimization [28, 52], and robotics [17]. It is particularly useful for expensive-to-evaluate, non-linear objectives, as in our case with $\texttt{cost}(x)$ and $\texttt{perf}(x)$. Moreover, the discontinuous nature of these objective functions make the use of traditional gradient-based or linear optimizers a poor fit for our problem.

BO works by building a probabilistic surrogate model for the objective function(s), and uses it to make decisions about which points (e.g., feature representations) in the search space to evaluate next. Typically, BO begins by sampling an initial number of points at random to build the surrogate model. Each subsequent iteration involves generating a set of candidate points, using the surrogate model to predict the objective function's output on the candidate set, and choosing the next "best" (e.g., maximizing Expected Improvement) point to evaluate using the real objective function. The surrogate model is then updated to include the newly evaluated point, and the process repeats until some stopping criteria is met, such as a maximum number of iterations or a satisfactory level of convergence [24, 35]. While BO is not guaranteed to work well in high-dimensional, multi-objective spaces [61], we describe how we augment it for our context later in this section. We compare the performance of BO against other search techniques in Section 5.3 and show that our approach is efficient at approximating the Pareto front.

BO Formulation. We formulate CATO's search process as a multi-objective Bayesian optimization problem with $|\mathcal{F}|+1$ dimensions: one dimension per feature in $\mathcal{F}$ and one for the connection depth n. Each feature parameter is represented by a binary indicator variable, which denotes whether or not the feature is included. The connection depth parameter is separately encoded as an integer or real-valued variable upper bounded by the maximum connection depth N. This setup lets CATO *concurrently* search over both features and connection depth while optimizing for systems cost and model performance. We define this search as a minimization of the two functions $\texttt{cost}(F,n)$ and $-\texttt{perf}(F,n)$. These objective functions are managed by the Profiler (Section 3.4), which generates complete serving pipelines according to the feature representations sampled by the Optimizer, and returns the end-to-end systems cost and model performance metrics.

Tailoring BO for Traffic Analysis. In its basic form, BO has several limitations. Conventional applications of BO typically involve single-objective, low-dimensional (fewer than 20) search spaces [24, 61]. However, our traffic analysis problem is inherently multi-objective, high-dimensional, and involves a complex search space with mixed categorical (features) and numerical (connection depth) variables. To address this, we augment the CATO Optimizer with two preprocessing techniques to improve its sample efficiency (Figure 3). The first is a dimensionality reduction step that strategically discards candidate features that are unlikely to improve the model's predictive performance regardless of its impact on the end-to-end systems costs. By default, we exclude features with a mutual information [71] score of zero, which indicates no direct informational relationship with the target variable.

The second technique incorporates prior probabilities into the BO formulation, accelerating the search by providing the Optimizer with "hints" about the approximate locations of Pareto-optimal feature representations. To account for both objectives, CATO constructs two sets of priors: one over the feature space that targets $\texttt{perf}(x)$, and one over the connection depth that targets $\texttt{cost}(x)$. The set of priors over the feature space encodes each feature's relative contribution to the model's performance, and are derived from the mutual information scores computed in the dimensionality reduction step. Formally, we define the prior probability of whether a feature f is part of a Pareto-optimal feature representation $x=(F,n)$ as $\mathbb{P}(f \in F | x \in \Gamma) = (1-\delta)\frac{I(f)}{I_{max}} + \frac{\delta}{2}$, where $I(f)$ represents the mutual information of f with respect to the target variable, I_{max} is the maximum mutual information among all candidate features, and δ is a damping coefficient. The damping coefficient is used to adjust the priors to prevent the feature with the highest mutual information from always being included. $\delta = 0$ signifies no damping, while $\delta = 1$ results in uniform priors for all features. These probabilities encourage CATO to more frequently explore regions of the search space that include features with higher predictive power.

The prior over the connection depth is represented by a probability mass function that decays linearly as the connection depth increases. The rationale is that for the same feature set, waiting longer to capture more packets or bytes before feature extraction correlates with worse systems performance. This prior encourages CATO to more frequently explore representations that require fewer packets or less data, despite not requiring domain-specific knowledge about the optimal connection depth at which to collect features. As we will show in Section 5.5, CATO is robust to reasonably large connection depth ranges.

We emphasize that these preprocessing steps can be performed efficiently without needing to evaluate the objective functions. Despite the term "prior," no prior knowledge about the optimal features or connection depth needs to be supplied by the user. CATO automatically derives the priors used to accelerate its search, thereby streamlining its usability.

3.4 The CATO Profiler

The CATO Profiler evaluates the feature representations sampled by the Optimizer based on the concrete definitions of $\texttt{cost}(x)$ and $\texttt{perf}(x)$. To accomplish this, it generates code for the packet capture and feature extraction stages of each sampled point, trains the model, and runs the full serving pipeline to *directly measure* its end-to-end systems costs and model performance. This measurement serves two purposes: (1) guiding the search process of the Optimizer, and (2) validating the in-network performance of identified solutions.

Why Measure? Using heuristics to estimate the end-to-end systems cost of a traffic analysis pipeline is difficult. Much like how existing heuristics that approximate model performance often fail to capture interdependencies and correlations among features [6, 34, 71], systems cost heuristics can similarly fail to capture the complexities of packet capture and feature extraction. Traffic analysis is particularly sensitive to this, since the processing steps during feature extraction often overlap in non-trivial ways. For instance, computing mean TCP window size and the number of ACKs sent in a connection require parsing each packet down to its TCP header, a shared task that must be factored into the end-to-end cost. Likewise, computing the mean window size also involves calculating its sum, the latter of which can then essentially be used for free. Other factors like resource contention and characteristics of the network traffic (e.g., bursty vs. non-bursty) can also unpredictably affect the end-to-end systems cost [58].

We argue that rather than trying to model or predict these complex systems-level interactions, it is both more accurate and useful to perform direct measurement. With direct measurement, CATO captures the actual end-to-end cost of the serving pipeline, encompassing all critical components including packet capture, feature extraction, and the model inference itself. Accurate measurement not only helps the Optimizer make well-informed decisions, but also helps users build confidence in validating whether identified solutions are operationally viable. Although this approach can be computationally expensive, the cost of training the model, generating the full serving pipeline, and measuring its performance is balanced by the sample efficiency of the Optimizer.

Pipeline Generation. To evaluate different feature representations during the search process, we require an *automated* way to measure $\texttt{cost}(x)$ and $\texttt{perf}(x)$ for any $x \in \mathbb{X}$. With a search space size of $O(2^{|\mathcal{F}|} \times N)$, manually implementing packet capture, feature extraction, and model inference for each evaluated point is impractical. One approach that enables flexible evaluation is "runtime branching," which uses branching logic at runtime to determine which paths in the code should be executed to extract a given feature representation. However, runtime branching introduces additional overhead that can contaminate the cost measurements of performance-sensitive traffic analysis pipelines. Instead, CATO employs *conditional compilation* to build and run customized end-to-end serving pipelines tailored to each configuration. The resulting binary matches the performance of a manually implemented pipeline, containing only the set of operations needed to collect traffic data up to the specified connection depth, extract the corresponding features, and execute the model inference. This technique not only constructs fully operational traffic analysis pipelines, but also provides the flexibility to accurately measure any point in the search space.

Pipeline Measurement. CATO presents a testbed interface that replicates a real-world deployment scenario of the pipeline. For model performance measurements, the Profiler trains a fresh model for each representation sampled by the Optimizer and directly measures its predictive performance to account for any interaction effects between features. The final performance metric is derived from a hold-out test set. We note that CATO operates on pre-labeled datasets, meaning it does not focus on automatic labeling or ground truth generation. The framework is designed to optimize model accuracy and system performance based on this labeled input. For systems cost measurements, CATO either simulates traffic inputs from the training data, or, when feasible, deploys the full serving pipeline in its target network environment (e.g., a passive monitoring or bump-in-the-wire deployment model) for end-to-end measurements. While each measurement can be expensive, the Optimizer is intentionally designed to minimize the number of measurements needed to approximate the Pareto front. We report wall-clock times for several of our evaluated use cases (Section 5.1) in Appendix E.

4 Implementation of CATO

We detail our implementation of CATO, covering the Optimizer, Profiler, model training, and objective functions.

Bayesian Optimization. We implement the CATO Optimizer using HyperMapper [52], a Bayesian optimization framework for design space exploration. HyperMapper supports multi-objective optimization with mixed-variable search spaces, but is not tailored specifically for high-dimensional BO. We use πBO [32] for prior injection, but adapt its implementation to incorporate CATO-generated priors in multi-objective use cases. We use a random forest as the surrogate model, which has been shown to perform well compared to more traditional Gaussian processes for discontinuous and non-linear objective functions [52]. The prior over the packet depth is constructed using the Beta distribution with $\alpha = 1$ and $\beta = 2$. We initialize the Optimizer with three iterations of random search space exploration and choose $\delta = 0.4$ based on empirically tuned values (Section 5.5).

Pipeline Generation. The CATO Profiler generates serving pipelines using a modified version of Retina [72], a Rust framework that compiles traffic *subscriptions* into efficient packet processing pipelines. A subscription defines the rules for how incoming traffic should transformed into a specific

```
1  fn on_packet(&mut self, packet: Packet) {
2      #[cfg(any(feature="iat_sum"))]
3      {
4          let pkt_timestamp = packet.timestamp();
5          self.iat_sum += pkt_timestamp - last_timestamp;
6          let last_timestamp = pkt_timestamp;
7      }
8      #[cfg(any(feature="ttl_min",feature="winsize_max"))]
9      let eth = packet.parse_eth();
10     #[cfg(any(feature="ttl_min",feature="winsize_max"))]
11     let ipv4 = eth.parse_ipv4();
12     #[cfg(any(feature="ttl_min"))]
13     self.ttl_min = self.ttl_min.min(ipv4.ttl());
14     #[cfg(any(feature="winsize_max"))]
15     {
16         let tcp = ipv4.parse_tcp();
17         self.winsize_max = self.winsize_max.max(tcp.winsize());
18     }
19 }
20
21 fn extract(&mut self) -> Vec<f64> {
22     vec![
23         #[cfg(feature="iat_sum")]
24         self.iat_sum,
25         #[cfg(feature="ttl_min")]
26         self.ttl_min,
27         #[cfg(feature="winsize_max")]
28         self.winsize_max,
29     ]
30 }
```

Figure 4: An example portion of the CATO Profiler's template subscription module. Each operation is predicated on its associated features and conditionally compiled with the `cfg` macro. For instance, if the evaluated feature set consists of `ttl_min` and `winsize_max`, then only lines 9, 11, 13, 16, and 17 will execute on each new packet, and only those two features will be extracted in the final feature representation. This enables dynamic cost profiling that matches the characteristics of a manually implemented feature extraction stage.

representation, and invokes a callback on the returned data. We implement the model inference stage in the callback, and subscribe to a template feature representation that can be modified at compile-time to the specific representation being evaluated. To dynamically generate the custom packet capture and feature extraction stages, we create a Retina subscription module that implements the processing steps needed to extract *all* candidate features. Each operation (e.g., parse an IPv4 header, add to a cumulative sum of packet inter-arrival times, etc.) is then annotated with a configuration predicate that specifies the subset of features necessitating its execution. If the feature representation being evaluated contains at least one of the predicated features, the predicate evaluates to true and the operation is conditionally compiled into the binary. This technique avoids redundant computation in shared steps, such as parsing headers, and ignores operations associated with features that are not included. For packet capture, we annotate the subscription with an early termination flag that stops data collection once the connection depth is reached.

Figure 4 shows pseudo-code for an example portion of the template subscription module. We implement 67 candidate features (Appendix A, Table 4) in 1,600 lines of Rust code. We note that the chosen candidate features are not specific to any use case: they are widely used in traffic analysis applications and are common features exposed by open source tools [3, 15, 16, 23, 51, 64, 72, 77]. We use number of packets into the connection to measure connection depth.

We note that our Profiler implementation uses Retina [72] to target commodity servers. However, CATO's core design principles remain applicable to optimizing and validating hardware-based traffic analysis pipelines, which we discuss further in Section 6.

Model Training. As an optimization framework, CATO is general to the specific type of model used in the traffic analysis pipeline. We implement support for three model types: decision trees (DT), random forests (RF), and deep neural networks (DNN). For DT and RF, we use scikit-learn's DecisionTreeClassifier and RandomForestClassifier, with 5-fold nested cross validation and grid search for hyperparameter tuning. We tune the maximum tree depth from 3–20 and set the number of estimators to 100 for RF. To match the speed of the Rust-based feature extraction stage, we retrain the best-performing DT and RF models in Rust using the SmartCore [1] library and evaluate the final Rust model on a hold-out test set containing 20% of the data.

For DNN, we implement a fully connected feedforward neural network in TensorFlow, consisting of three hidden layers with ReLU activation and L2 regularization. We apply dropout to prevent overfitting and use the Adam optimizer for training. Since Rust lacks mature DNN libraries, we train and evaluate the DNN models entirely in Python/TensorFlow. Additional details are provided in Appendix C.

Objective Functions. We use end-to-end inference latency, zero-loss classification throughput, and pipeline execution time as three different metrics for systems cost. End-to-end inference latency measures the duration from the arrival of the first packet in the connection to the model's final prediction. This includes the time spent extracting features from raw traffic, the model inference time, and time spent waiting for packets to arrive. Zero-loss throughput is the highest ingress traffic rate that can be sustained by the serving pipeline with no packet drops, which we negate to match the sign of $\texttt{cost}(x)$ minimization. The execution time measures the total CPU time spent in the serving pipeline, excluding time between packets. This metric is less dependent on the specific characteristics of the input traffic, and is an indirect measure of both latency and throughput. Although these can be combined into a single cost metric, we evaluate them separately to show CATO's flexibility. Depending on the traffic analysis use case, which we detail in the next section, we use either the F1 score or root-mean-squared error, calculated from the predictions on the hold-out test set as the model performance metric.

5 Evaluation

We evaluate CATO over a variety of configurations and use cases. Section 5.1 details our datasets and testbeds. In Section 5.2, we show that CATO can help traffic analysis applications achieve substantially lower inference latency and higher

Use Case	Type	Traffic	Model
app-class	Classification	Live	Decision Tree
iot-class	Classification	Dataset	Random Forest
vid-start	Regression	Dataset	Deep Neural Network

Table 2: Evaluation Use Cases

throughput without compromising model performance, and in many cases improve upon both metrics. We also compare it with Traffic Refinery [14], a recent system for cost-aware ML on network traffic. Section 5.3 compares the efficiency of the CATO Optimizer to alternative Pareto-finding approaches, and Section 5.4 performs an ablation study of the Profiler. In Section 5.5, we run micro-benchmarks on CATO's sensitivity to various search space sizes and hyperparameters.

5.1 Datasets and Testbeds

We consider three use cases in our evaluations: web application classification (app-class), IoT device recognition (iot-class), and video startup delay inference (vid-start). These are typical analysis tasks of varying complexity that are representative of the type of ML-based inference performed on network traffic. Table 2 summarizes them, with more details provided in Appendix B.

Web Application Classification. While open-source traffic classification datasets exist, replaying them at modern line rates is challenging without duplicating flows. To evaluate serving pipelines against real traffic at high speeds, we develop a use case that identifies one of six common web applications from live traffic on a large university network. This type of classification is typically used by web application firewalls or in the early stages of network QoE inference pipelines [2, 15]. For ground truth, we label connections using the server name in the TLS handshake. We train and evaluate decision tree models using flow statistics captured from the network, then deploy them to the same network for real-time serving using Retina [72].

IoT Device Recognition. To help make our results reproducible, we also consider an IoT device recognition use case based on the dataset published by Sivanathan et al. [65]. We use a random forest to classify connections as belonging to one of 28 IoT device types. Although real-time throughput experiments are not feasible without duplicating flows, we use this dataset to report micro-benchmarks and evaluate CATO's ability to approximate the true Pareto front.

Video Startup Delay Inference. We further demonstrate CATO's generalizability to different traffic analysis tasks and model types through a regression use case that predicts the startup delay of video streams. Startup delay inference is widely used in analysis of encrypted video traffic as a measure of QoE [8, 15, 49]. We choose startup delay (rather than other QoE metrics) to provide a regression task that complements the previous two classification use cases. We also adopt a more complex DNN instead of a tree-based model, using the YouTube dataset published by Bronzino et al. [15].

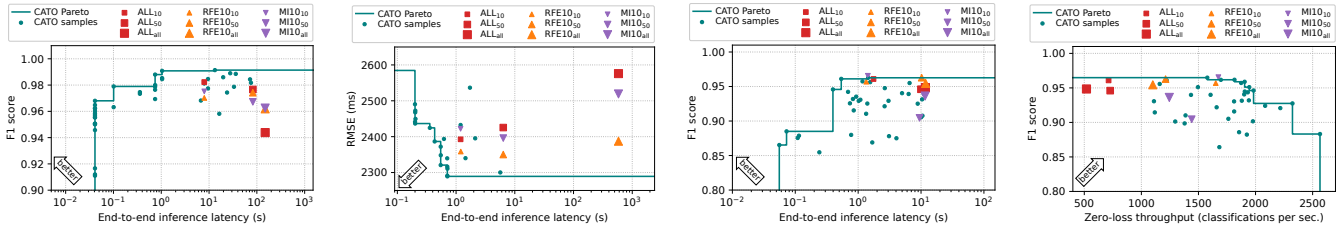
5.2 Model Serving Performance

We first examine the end-to-end inference latency, zero-loss throughput, and predictive performance of CATO-optimized serving pipelines. Note that CATO itself is not a classifier, but a general framework for optimizing ML-based serving pipelines for real-time traffic analysis. Therefore, instead of directly comparing CATO with existing classifiers or models, we evaluate it against optimization strategies commonly used in prior work to build those models. We use the following feature optimization methods and combine them with early inference techniques as our baselines:

- ALL: Use all available features.
- RFE10: Select the top ten features by recursive feature elimination [27]. RFE trains a model using all available features, then iteratively removes the least important feature and retrains until the desired number remains.
- MI10: Select the top ten features based on mutual information [71]. This is a model agnostic algorithm that measures how much information each feature contributes to the target variable and picks the most relevant ones.

Prior traffic analysis solutions typically wait until the end of the connection before making a prediction [7, 9, 15, 40, 62] or use a fixed packet depth for early inference [5, 10, 11, 29, 45, 50, 56]. For example, Peng et al. [54] collects up to the first 10 (including TCP handshake) packets, while recent work like GGFAST [55] use the first 50. For a thorough analysis, we compare against these strategies by running each baseline at packet depths of 10, 50, and all packets. CATO does not assume a predefined optimal packet depth, but searches over the entire feature representation space as part of its optimization process. We choose a maximum packet depth of 50 and run for 50 iterations, which is consistent with common machine learning practices [13, 32]. We show how CATO reacts to different packet depth ranges in Section 5.5.

End-to-End Inference Latency. Figures 5a, 5b, and 5c show the end-to-end inference latency and predictive performance (F1 score or RMSE) for iot-class, vid-start, and app-class, respectively. Each CATO sample represents a candidate point explored during the optimization process, with CATO's Pareto front constructed from the set of non-dominated points. For iot-class and vid-start, all points on CATO's Pareto front dominate the baseline solutions, achieving equal or better predictive performance with lower end-to-end latency. For iot-class, CATO can reduce the inference latency by 11–79× compared to solutions that use the first 10 packets in the connection, 817–2000× compared to those that use the first 50, and over 3600× (from several minutes to under 0.1 seconds) compared to those that wait until the end of the connection. Likewise for vid-start, CATO generates solutions that can infer video startup delays



(a) `iot-class` latency (b) `vid-start` latency (c) `app-class` latency (d) `app-class` throughput

Figure 5: Comparison of F1 score / RMSE vs. end-to-end inference latency / zero-loss throughput (single-core) for `iot-class`, `vid-start`, and `app-class` serving pipelines. CATO identifies multiple solutions on its Pareto front that dominate those found by traditional optimization techniques, and can achieve significantly better systems and predictive performance.

in less than one second (a 2.2–2900× speedup, depending on baseline) while also reducing the mean squared error of its predictions.

Since end-to-end inference latency is largely dominated by packet inter-arrival times, this improvement can be attributed to CATO's ability to find alternative sets of features using the fewest packets necessary without compromising the predictive performance of these popular feature selection methods. For example, RFE10 using the first 10 packets (RFE10_{10}) in `iot-class` achieves an F1 score of 0.970 with an inference latency of 7.9 seconds. However, CATO identifies a different set of features using just the first 3 packets for a better F1 score of 0.979 and an inference latency of 0.1 seconds.

We find a similar pattern for `app-class`, where CATO-optimized pipelines outperform most baseline methods across both objectives. While MI10_{10} and RFE10_{50} achieve slightly higher F1 scores (0.963 and 0.962), CATO produces a solution with a nearly identical F1 score (0.960) and a latency of 0.54 seconds—2.6× and 19× faster than MI10_{10} and RFE10_{50}, respectively. These results reinforce that for traditional feature optimization methods, it is not always clear a priori which feature set at which packet depth results in the best model performance or serving efficiency. Through end-to-end optimization of both objectives over the entire feature representation space, CATO is able to automatically derive and validate the performance of faster and more accurate traffic analysis pipelines.

Zero-Loss Classification Throughput. We compare the predictive performance and classification throughput of solutions found by CATO with those found by the baseline methods for `app-class`. We exclude `iot-class` and `vid-start` due to limitations in replaying the traces at high speeds without repeating flows. For a realistic assessment, we use live traffic from our campus network, but restrict all experiments to a single core to avoid saturating our network's maximum ingress throughput. In an actual deployment scenario, the throughput can be easily scaled up by adding more cores, owing to the per-core scalability of Retina [72]. More details about our throughput experiments can be found in Appendix D.

Figure 5d shows that CATO's solutions outperform the baselines in both throughput and F1 score, with the exception of MI10_{10}. Despite this, CATO successfully identifies the feature representation with the highest overall F1 score and the one with the highest zero-loss throughput. For a decrease in F1 score from 0.96 to 0.93, CATO can increase throughput by 37%. Compared to solutions that wait until the end of the connection, CATO can improve the zero-loss throughput by a factor of 1.6–3.7×, and 1.3–2.7× for those that require the first 50 packets while also achieving higher model performance. Notably, CATO achieves these results after exploring just 50 feature representations out of $2^{67} \times 50 = 7 \times 10^{21}$ (67 candidate features, up to a maximum packet depth of 50).

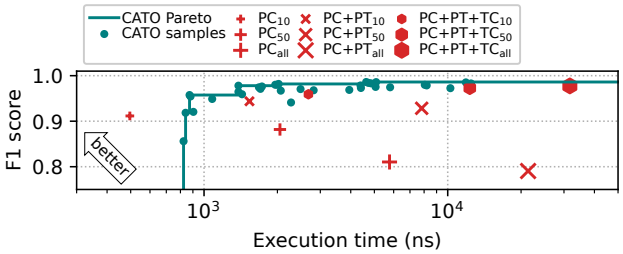


Figure 6: F1 score vs. pipeline execution time for `iot-class` using CATO and Traffic Refinery (in red). While PC_{10} achieves a strong trade-off due to its low cost, CATO more consistently finds solutions closer to the Pareto front.

Comparison with Traffic Refinery. We further compare CATO with Traffic Refinery [14], a recent traffic analysis framework that also facilitates joint evaluation of model performance and system costs. Unlike CATO, Traffic Refinery requires manual exploration of flow features and connection depth. We simulate Traffic Refinery's cost profiler using CATO's execution time cost metric and replicate its built-in packet counter (PC), packet timing (PT), and TCP counter (TC) feature classes. While Traffic Refinery defaults to making an inference after ten seconds into the flow, we evaluate it at packet depths of 10, 50, and all packets for consistency with the above baselines (see Appendix F for more details).

Figure 6 plots F1 score vs. execution time for `iot-class`

using CATO and Traffic Refinery. The results show that Traffic Refinery's macro-aggregation of standard feature classes is less efficient than CATO at finding optimal trade-offs. CATO achieves better accuracy at lower cost across all sampled points except PC_{10}. Even in this case, it is still able to identify an alternative representation with a similar F1 score and a modest 344 ns increase in pipeline execution time. While packet counters alone at a packet depth of 10 happen to perform well for `iot-class` at minimal cost, adding packet timing and TCP state information further improves accuracy but incurs a much higher execution time. Identifying such configurations with Traffic Refinery requires trial-and-error, as it is not clear which combinations perform best at which packet depths. In contrast, CATO more efficiently explores the feature representation space, clustering solutions closer to the Pareto front and more effectively balancing predictive performance and systems cost across a broader range of configurations.

5.3 Optimizer Efficiency

In this section, we evaluate the efficiency of the CATO Optimizer by measuring the quality of its computed Pareto front and the speed at which it converges to the true Pareto front. Since it is infeasible to exhaustively measure all points in $\mathbb{X}$ to obtain the true Pareto front for the default candidate feature set size of 67, we evaluate on a smaller candidate set of six features (Appendix A, Table 4) for the `iot-class` use case in order to obtain the ground truth. We compare the Optimizer with three alternative Pareto-finding algorithms:

- SIMA: Use simulated annealing [38], a metaheuristic for solving optimization problems with a complex search space. We provide details of our multi-objective implementation in Appendix G.
- RAND: Sample a random subset of features at a random packet depth without replacement.
- ITERALL: Use all available features but increment the packet depth on each iteration, starting from one.

We remark that these algorithms differ from the baseline methods described in the previous section in that they attempt to estimate the Pareto front rather than a single point solution. In N search iterations, each of these alternative approaches makes exactly N calls to $\texttt{cost}(x)$ and $\texttt{perf}(x)$.

Pareto Front Quality. We compare the quality of the Pareto front estimated by each algorithm using Hypervolume Indicator (HVI). HVI is a common metric used to compare multi-objective algorithm performance, and measures the area between the estimated Pareto front and the true Pareto front bounded by a reference point [41]. We use pipeline execution time as our chosen systems cost metric and F1 score for model performance. Since F1 score and execution time have different raw value ranges, we normalize the data before computing HVI to assign similar importance to both objectives.

We run each search algorithm for 50 iterations at a maximum packet depth of 50. In Figure 7, we plot the feature representations sampled at each iteration alongside the corresponding Pareto front constructed from the non-dominated points. These sampled points represent candidate solutions explored during optimization and do not directly impact the end-to-end traffic analysis pipeline. While many intermediate samples overlap between algorithms due to inherent randomness, they primarily serve to illustrate the regions explored by each algorithm and are not included in the final solution. For reference, we also show the true Pareto front computed from exhaustively measuring all $2^6 \times 50 = 3{,}200$ feature representations in the search space.

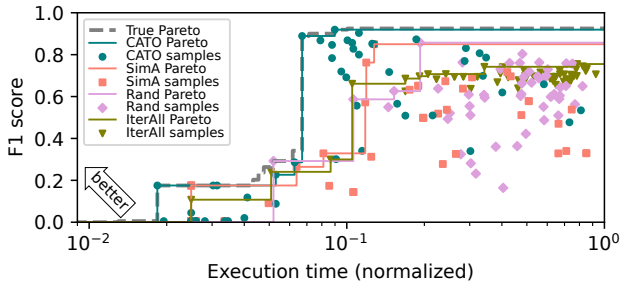


Figure 7: Estimated Pareto fronts after 50 iterations. CATO outperforms other Pareto-finding approaches, especially in regions with high F1 scores and low execution times. The sampled points illustrate the candidate solutions explored by each algorithm during the optimization process, while only the Pareto-optimal points are included in the final output.

We observe that CATO's Pareto front closely approximates the true Pareto front while sampling less than 1.6% of the search space. Using a worst-case reference point (F1 score of 0 and normalized execution time of 1), CATO achieves an HVI of 0.98, compared to 0.88, 0.86, and 0.77 achieved by SIMA, RAND, ITERALL, respectively. We note that CATO does not entirely dominate all alternative search algorithms, especially around F1 scores of 0.3. However, it performs better at higher and lower extremes. This behavior can be attributed to CATO's tendency to explore representations that require very few packets (due to the decay-shaped prior placed over connection depth), while also injecting priors based on each feature's relative importance. If we only consider solutions with an F1 score of at least 0.8, the HVI is 0.95 for CATO, 0.39 for SIMA and RAND, and 0 (no solutions found) for ITERALL.

Convergence Speed. Figure 8 compares the sample efficiency of CATO with alternative search algorithms. We extend the 50 sample explorations commonly used by ML practitioners [13, 32] to 1,500 to examine the convergence rate towards the true Pareto front as most of the search space is explored. We also plot the performance of CATO's baseline BO formulation without dimensionality reduction and prior injection (CATOBASE). We exclude ITERALL from

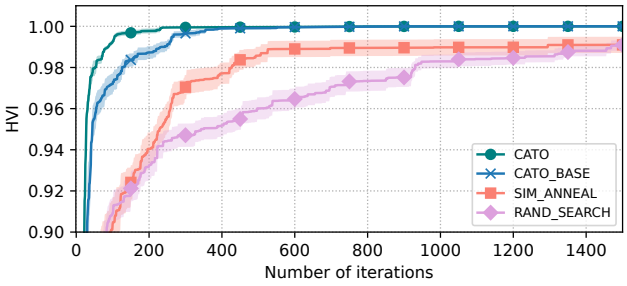


Figure 8: CATO efficiently converges to the true Pareto front. We show the mean and standard error of the HVI with a worst-case reference point across 20 runs.

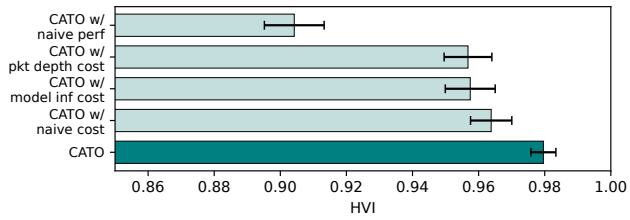


Figure 9: CATO with alternative Profiler metrics. End-to-end measurements prove useful for estimating the Pareto front and validating the performance of identified solutions.

this analysis since more than 50 iterations would exceed the maximum packet depth covered by the ground truth Pareto front. Moreover, we find that the HVI for ITERALL does not show significant improvement beyond this point.

We can see that CATO converges to the true Pareto front fastest, demonstrating a more sample-efficient approach to optimizing ML-based traffic analysis pipelines. CATO surpasses 0.99 HVI (using a worst-case reference point) within 87 iterations on average compared to CATOBASE's 240 iterations, demonstrating a speedup of 2.76×. This speedup can be attributed to the incorporation of priors on the optimization parameters as described in Section 3.3, which helps CATO emphasize more promising regions in the search space. SIMA and RAND are less sample efficient, surpassing 0.99 HVI at 1,295 iterations and 1,469 iterations for a CATO speedup of 14.9× and 16.9×, respectively.

5.4 Ablation Study of the Profiler

We assess the impact of the Profiler on the estimated Pareto front found by CATO. We retain the Optimizer, including dimensionality reduction and prior injection, and perform an ablation study by replacing $\texttt{cost}(x)$ and $\texttt{perf}(x)$ measurements with heuristic metrics. We devise four variants of CATO. The first is CATO W/ NAÏVE COST, which replaces the original cost metric with the sum of the costs of each feature in isolation. This design captures the end-to-end systems costs of individual features, but fails to account for shared processing steps during packet capture and feature extraction. The second is CATO W/ MODEL INF COST, which measures the model inference speed but ignores the cost of packet capture and feature extraction. CATO W/ PKT DEPTH COST directly uses packet depth as the cost. CATO W/ NAÏVE PERF retains the original $\texttt{cost}(x)$ metric but replaces $\texttt{perf}(x)$ with the sum of each feature's mutual information with respect to the target variable. This version does not account for the effects of feature interactions.

We run each variant for 50 Optimizer iterations using the smaller candidate feature set, then measure (using the Profiler) the true $\texttt{perf}(x)$ and $\texttt{cost}(x)$ of each sampled point in a post-processing step to compare HVI. Figure 9 reveals that CATO comes closest to the true Pareto front, demonstrating that there is value to incorporating real model performance and systems costs measurements as feedback for the Optimizer. Furthermore, we note that none of the variants provide a means to *validate* the expected in-network performance of their identified solutions. In particular, CATO W/ NAÏVE PERF and CATO W/ PKT DEPTH COST do not yield meaningful performance metrics. CATO W/ NAÏVE COST and CATO W/ MODEL INF COST are better, but may overestimate or underestimate the true systems cost, respectively. Depending on the concrete definition of $\texttt{cost}(x)$, this could lead to the deployment of an unrealizable model (e.g., by overestimating the throughput or underestimating the latency of the pipeline).

5.5 Microbenchmarks

In this section, we evaluate CATO's robustness to varying search space sizes and its sensitivity with respect to its BO initialization and damping coefficient hyperparameters.

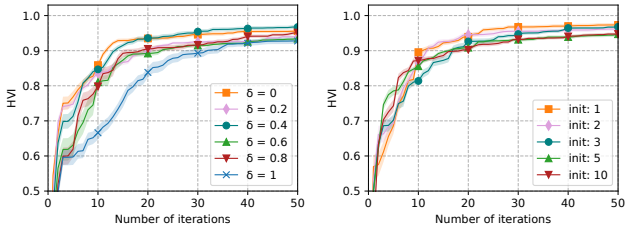
Varying Maximum Connection Depth. Previously, we showed that CATO is highly effective across both large (67) and small (6) candidate feature sets, but limited the search space to a maximum connection depth of 50 packets. Here, we explore the impact of maximum connection depth and the size of the search space on CATO's ability to identify Pareto-optimal feature representations.

In general, limiting the search space inherently restricts the number of possible feature representations, potentially compromising the quality of the estimated Pareto front. Conversely, expanding the search space offers a richer set of feature representations, but makes locating the true Pareto front more challenging. Table 3 shows CATO's performance across different maximum packet depths for `iot-class` using the full 67 candidate features. Since it is difficult to compare HVI across configurations without a ground truth, we report metrics for the estimated Pareto-optimal representations with the highest F1 score and lowest overall execution time.

We can see that restricting the packet depth to very small values (e.g., less than 5) limits CATO's ability to find a so-

Max Depth N	Highest F1 Score n	F1 score	Time (μs)	Lowest Execution Time n	F1 score	Time (μs)
3	3	0.959	1.37	1	0.310	0.2
5	4	0.983	1.30	1	0.520	0.26
10	7	0.994	2.04	1	0.520	0.26
25	7	0.989	2.61	1	0.520	0.26
50	7	0.993	2.10	1	0.461	0.27
100	10	0.990	3.01	1	0.005	0.24
∞	42k	0.984	28.2	52k	0.944	16.5

Table 3: Estimated Pareto-optimal solutions with the highest F1 score and lowest execution time for different maximum packet depths. CATO is able to identify high quality solutions, even when expanding the maximum connection depth.



(a) Damping coefficient δ. (b) BO initialization samples.

Figure 10: Effects of varying the damping coefficient and the number of samples used to initialize the BO surrogate model.

lution that yields an F1 score above 0.99, likely because not many feature sets can achieve such high model performance using only the first few packets in a flow. However, if we expand the search space to include features extracted from *up to* the first 10–100 packets, CATO is still able to identify feature sets that only need the first 7–10 packets to achieve good model performance despite the larger maximum packet depth. However, if the search space over packet depth is unbounded, CATO struggles converge to a feature representation with low cost since the Optimizer has too much flexibility to explore any value between 1 and the maximum number of packets across *all* flows in the training set. These results reinforce that concurrently searching over different features and when to collect those features, rather than predefining a fixed connection depth, can lead to more optimal traffic analysis pipelines. CATO can identify highly efficient and predictive feature representations within a wide range of connection depths, even without prior knowledge of its optimum.

Sensitivity Analysis. We analyze the sensitivity of CATO with respect to its hyperparameters. We again use the smaller candidate feature set to measure HVI against the true Pareto front with a worst-case reference point. Figure 10a shows the impact of varying the damping coefficient δ between 0 and 1. Recall that $\delta = 0$ represents a prior probability equivalent to the normalized mutual information, while $\delta = 1$ represents uniform priors for each feature. Using uniform priors performs the worst with an HVI of 0.93 after 50 iterations. With less damping, CATO converges faster in earlier iterations. We find that a $\delta = 0.4$ results in the highest performance at 50 iterations, while $\delta = 0$ also performs well overall.

In Figure 10b, we vary the number of samples used to initialize the BO surrogate model. Initialization samples are chosen at random but weighted according to the priors. We observe little difference in CATO's performance for small initialization values, but find that initializing with just 1 point empirically results in the highest HVI after 50 iterations. We choose to go with a more conventional value for BO [35], and choose 3 initialization samples by default.

6 Related Work

Efficient Inference. Several systems have been proposed for efficient traffic analysis inference. However, many of these efforts focus only on increasing the speed of the final model inference stage, rather than that of the end-to-end serving pipeline [22, 43, 69]. Such approaches overlook the effects of the packet capture and feature extraction stages, both of which CATO considers in its end-to-end optimization. AC-DC [34] and pForest [16] are similar in that they explicitly consider the preprocessing costs of extracting features. AC-DC performs inference under dynamic memory constraints while pForest targets programmable hardware. Both of these differ from CATO in that they generate pools of models and dynamically switch between them based on inference requirements.

There is a growing body of research that proposes the use of programmable hardware for traffic analysis [9, 16, 33, 36, 59, 64, 67, 68, 74, 75, 80, 82]. For example, N3IC [64] uses binary neural networks to implement traffic analysis models on FPGAs and SmartNICs, while BoS [75] enables RNN inference on programmable switches. These approaches focus on the trade-off between accuracy and efficiency of the *model* under the constraints of dataplane hardware, such as limited memory and lack of support for floating-point operations, multiplications, and loops. Our work, by contrast, focuses on the choice of *traffic* representation—spanning both feature selection and connection depth—to co-optimize systems performance and predictive accuracy. CATO can complement hardware-focused techniques: by applying the CATO Optimizer and extending our Profiler implementation to target programmable hardware, we can identify traffic representations that are cheaper to collect while also validating the end-to-end performance of the hardware pipeline. This combined strategy can further boost the efficiency and predictive performance of ML-based traffic analysis, which we leave for future work.

Balancing Systems and Model Performance. There exist general-purpose serving frameworks that aim to balance efficiency with model performance through techniques like caching [20], adaptive model selection [20, 58], autoscaling [19, 58, 78], and scheduling for resource-quality tradeoffs [42, 79]. Unlike CATO, these frameworks primarily fo-

cus on optimizing resource allocation and are less suited for real-time traffic analysis, where even small inefficiencies in end-to-end systems performance can cause not just delayed results, but also potentially invalidate the model due to packet loss. Additionally, CATO explicitly considers the optimal point in the flow (i.e., connection depth) for making predictions, which previous frameworks do not address. Traffic Refinery [14] considers joint optimization of model and system performance for ML-based network traffic analysis. However, it relies on manual exploration of data representations and focuses only on individual feature costs, whereas CATO performs automated end-to-end optimization of the entire analysis pipeline.

Bayesian Optimization for Traffic Analysis. Bayesian Optimization is a popular technique for compiler optimization [28], FPGA design [52], and hyperparameter tuning [66, 70], but has seen limited use in optimizing ML-based traffic analysis pipelines. Most similar is Homunculus [68], which uses Bayesian optimization to generate ML models for datacenter network applications under the resource constraints of data-plane hardware. Homunculus optimizes over different model architectures and their hyperparameters, but is single-objective and does not consider the effects of different feature sets, connection depths, and their associated systems costs. CATO, on the other hand, uses multi-objective BO over the entire feature representation space to simultaneously optimize serving efficiency and predictive performance.

7 Conclusion

In this work, we introduced CATO, a framework for end-to-end optimization of ML-based traffic analysis pipelines. By leveraging multi-objective Bayesian optimization coupled with a realistic pipeline generator and profiler, CATO efficiently builds and validates serving pipelines that balance both model accuracy and systems performance. Our evaluations on live traffic and offline traces showed that CATO can improve end-to-end inference latency, throughput, and pipeline execution time across diverse traffic analysis tasks, while also maintaining or enhancing model performance. Future work includes broader model selection strategies and extending performance profiling across heterogeneous serving hardware.

Acknowledgements

We thank our shepherd Zhizhen Zhong as well as Tina Wu, Thea Rossman, Qizheng Zhang, Carl Hvarfner, Luigi Nardi, and the anonymous reviewers for their helpful feedback. We thank the Stanford networking team, including Andrej Krevl, Johan van Reijendam, and Will Johnson. This work was supported in part by a Sloan Research Fellowship, the National Science Foundation under Grant Numbers #2319080 and #2124424, the ANR Project No ANR-21-CE94-0001-01 (MINT), and gifts from Google, Inc., Cisco Systems, Inc., and Comcast Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or other funding organizations.

References

[1] Smartcore. https://smartcorelib.org/, 2023.

[2] Web application firewall documentation. https://learn.microsoft.com/en-us/azure/web-application-firewall/, 2023.

[3] Zeek. https://zeek.org/, 2023.

[4] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. Deep learning for network traffic monitoring and analysis (NTMA): A survey. In *Computer Communications*, 2021.

[5] Hasan Faik Alan and Jasleen Kaur. Can android applications be identified using only TCP/IP headers of their launch time traffic? In *ACM Conference on Security and Privacy in Wireless Networks*, 2016.

[6] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. Permutation importance: A corrected feature importance measure. In *Bioinformatics*, 2010.

[7] Zied Aouini and Adrian Pekar. NFStream: A flexible network data analysis framework. In *Computer Networks*, 2022.

[8] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[9] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando M. V. Ramos, and André Madeira. Flowlens: Enabling efficient flow classification for ml-based network security applications. In *Network and Distributed Systems Security Symposium (NDSS)*, 2021.

[10] Laurent Bernaille, Renata Teixeira, Ismael Akodjenou, Augustin Soule, and Kavé Salamatian. Traffic classification on the fly. In *ACM SIGCOMM Computer Communication Review*, 2006.

[11] Laurent Bernaille, Renata Teixeira, and Kavé Salamatian. Early application identification. In *International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2006.

[12] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M. Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. In *Journal of Internet Services and Applications*, 2018.

[13] Xavier Bouthillier and Gaël Varoquaux. Survey of machine-learning experimental methods at NeurIPS2019 and ICLR2020. *Research Report, Inria Saclay Ile de France*, 2020.

[14] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Hyojoon Kim, Renata Teixeira, and Nick Feamster. Traffic refinery: Cost-aware data representation for machine learning on network traffic. In *ACM Measurement and Analysis of Computing Systems*, 2021.

[15] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. In *ACM Measurement and Analysis of Computing Systems*, 2019.

[16] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680v2*, 2022.

[17] Roberto Calandra, Nakul Gopalan, André Seyfarth, Jan Peters, and Marc Peter Deisenroth. Bayesian gait optimization for bipedal locomotion. In *International Conference on Learning and Intelligent Optimization (LION)*, 2014.

[18] Briang Chang, Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Learned load balancing. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2023.

[19] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *ACM Symposium on Cloud Computing*, 2020.

[20] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[21] Alberto Dainotti, Antonio Pescapé, and Carlo Sansone. Early classification of network traffic through multi-classification. In *International Workshop on Traffic Monitoring and Analysis (TMA)*, 2011.

[22] Kayathri Devi Devprasad, Sukumar Ramanujam, and Suresh Babu Rajendran. Context adaptive ensemble classification mechanism with multi-criteria decision making for network intrusion detection. *Concurrency and Computation: Practice and Experience*, 2022.

[23] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of encrypted and VPN traffic using time-related features. In *International Conference on Information Systems Security and Privacy (ICISSP)*, 2016.

[24] Peter I. Frazier. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[25] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. Realtime robust malicious traffic detection via frequency domain analysis. In *ACM SIGSAC Conference on Computer and Communication Security (CCS)*, 2021.

[26] Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. Requet: Real-time QoE detection for encrypted youtube traffic. In *ACM Transactions on Multimedia Computing, Communications*, 2020.

[27] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. In *Machine Learning*, 2002.

[28] Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. BaCO: A fast and portable Bayesian compiler optimization framework. In *ACM Architectural Support for Programming Languages and Operating Systems*, 2023.

[29] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New directions in automated traffic analysis. In *ACM Conference on Computer and Communication Security (CCS)*, 2021.

[30] Nen-Fu Huang, Gin-Yuan Jai, Han-Chieh Chao, Yih-Jou Tzang, and Hong-Yi Chang. Application traffic classification at the early stage by characterizing application rounds. In *Information Sciences*, 2013.

[31] Johann Hugon, Gaetan Nodet, Anthony Busson, and Francesco Bronzino. Towards adaptive ml traffic processing systems. In *Proceedings of the on CoNEXT Student Workshop 2023*, 2023.

[32] Carl Hvarfner, Danny Stoll, Artur Souza, Marius Lindauer, Frank Hutter, and Luigi Nardi. πBO: Augmenting acquisition functions with user beliefs for Bayesian optimization. In *International Conference on Learning Representations (ICLR)*, 2022.

[33] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. Leo: Online ml-based traffic classification at multi-terabit line rate. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[34] Xi Jiang, Shinan Liu, Saloua Naama, Francesco Bronzino, Paul Schmitt, and Nick Feamster. AC-DC: Adaptive ensemble classification for network traffic identification. *arXiv preprint arXiv:2302.11718*, 2023.

[35] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. In *Journal of Global Optimization*, 1998.

[36] Radhakrishna Kamath and Krishna M. Sivalingam. Machine learning based flow classification in DCNs using P4 switches. In *International Conference on Computer Communications and Networks*, 2015.

[37] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. BLINC: Multilevel traffic classification in the dark. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2005.

[38] Scott Kirkpatrick, C. Daniel Gelatt Jr., and Mario P. Vecchi. Optimization by simulated annealing. In *Science*, 1983.

[39] Vengatanathan Krishnamoorthi, Niklas Carlsson, Emir Halepovic, and Eric Petajan. BUFFEST: Predicting buffer conditions and real-time requirements of HTTP(S) adaptive streaming clients. In *ACM Multimedia Systems Conference*, 2017.

[40] Jong-Hyouk Lee and Kamal Singh. SwitchTree: In-network computing and traffic analyses with random forests. In *Neural Computing and Applications*, 2020.

[41] Miqing Li and Xin Yao. Quality evaluation of solution sets in multiobjective optimisation: A survey. In *ACM Computing Surveys*, 2019.

[42] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[43] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *SIGCOMM*, 2019.

[44] Yingqiu Liu, Wei Li, and Yunchun Li. Network traffic classification using K-means clustering. In *International Multi-Symposiums on Computer and Computational Sciences*, 2007.

[45] Manuel Lopez-Martin, Belen Carro, Antonio Sanchez-Esguevillas, and Jaime Lloret. Network traffic classifier with convolutional and recurrent neural networks for internet of things. In *IEEE Access*, 2017.

[46] Mohammad Lotfollahi, Mahdi Jafari Siavoshani, Ramin Shirali Hossein Zade, and Mohammdsadegh Saberian. Deep packet: A novel approach for encrypted traffic classification using deep learning. *Soft Computing*, 2020.

[47] Tarun Mangla, Emir Halepovic, Mostafa Ammar, and Ellen Zegura. Using session modeling to estimate HTTP-based video QoE metrics from encrypted network traffic. In *IEEE Transactions on Network and Service Management*, 2019.

[48] M. Hammad Mazhar and Zubair Shafiq. Real-time video quality of experience monitoring for HTTPS and QUIC. In *IEEE International Conference on Computer Communications*, 2018.

[49] M. Hammad Mazhar and Zubair Shafiq. Real-time video quality of experience monitoring for HTTPS and QUIC. In *IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[50] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, Ahmad-Reza Sadeghi, N. Asokan, and Sasu Tarkoma. IoT sentinel: Automated device-type identification for security enforcement in IoT. In *International Conference on Distributed Computing Systems*, 2017.

[51] Andrew Moore, Denis Zuev, and Michael Crogan. Discriminators for use in flow-based classification. Technical report, 2005.

[52] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. HyperMapper: a practical design space exploration framework. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2019.

[53] Thuy T.T. Nguyen and Grenville Armitage. A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities. In *IEEE Communications Surveys & Tutorials*, 2008.

[54] Lizhi Peng, Bo Yang, and Yuehui Chen. Effective packet number for early stage internet traffic identification. In *Neurocomputing*, 2015.

[55] Julien Piet, Dubem Nwoji, and Vern Paxson. GGFAST: Automating generation of flexible network traffic classifiers. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2023.

[56] Shahbaz Rezaei, Bryce Kroencke, and Xin Liu. Large-scale mobile app identification using deep learning. In *IEEE Access*, 2019.

[57] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. *arXiv preprint arXiv:1708.06376*, 2017.

[58] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[59] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the ai accelerator? In *Morning Workshop on In-Network Computing*, 2018.

[60] Gabriel Gómez Sena and Pablo Belzarena. Early traffic classification using support vector machines. In *International Latin American Networking Conference (LANC)*, 2009.

[61] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. In *Proceedings of IEEE, vol. 104, no. 1*, 2016.

[62] Tal Shapira and Yuval Shavitt. FlowPic: A generic representation for encrypted traffic classification and applications identification. In *IEEE Transactions on Network and Service Management*, 2021.

[63] Jayveer Singh and Manisha Nene. A survey on machine learning techniques for intrusion detection systems. In *Intl. Journal of Advanced Research in Computer and Communication Engineering*, 2013.

[64] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting traffic analysis with neural network interface cards. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[65] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT devices in smart environments using network traffic characteristics. In *IEEE Transactions on Mobile Computing*, 2019.

[66] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *International Conference on Advances in Neural Information Processing Systems*, 2012.

[67] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A data plane architecture for per-packet ML. In *ACM Architectural Support for Programming Languages and Operating Systems*, 2022.

[68] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-generating efficient data-plane ML pipelines for datacenter networks. In *ACM Architectural Support for Programming Languages and Operating Systems*, 2023.

[69] Da Tong, Yun R Qu, and Viktor K Prasanna. High-throughput traffic classification on multi-core processors. In *IEEE International Conference on High Performance Switching and Routing*, 2014.

[70] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *Proceedings of the NeurIPS 2020 Competition and Demonstration Track,*, 2021.

[71] Jorge R. Vergara and Pablo A. Estévez. A review of feature selection methods based on mutual information. In *Neural Computing and Applications*, 2014.

[72] Gerry Wan, Fengchen Gong, Tom Barbette, and Zakir Durumeric. Retina: Analyzing 100 GbE traffic on commodity hardware. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[73] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. In *International Conference on Information Networking*, 2017.

[74] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *ACM Workshop on Hot Topics in Networks*, 2019.

[75] Jinzhu Yan, Haotian Xu, Zhuotao Liu, Qi Li, Ke Xu, Mingwei Xu, and Jianping Wu. Brain-on-switch: Towards advanced intelligent network data plane via nn-driven traffic analysis at line-speed. In *USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[76] Hao Yang, Qin He, Zhenyan Liu, and Qian Zhang. Malicious encryption traffic detection based on NLP. In *Security and Communication Networks*, 2021.

[77] Kun Yang, Nick Feamster, and Samory Kpotufe. Feature extraction for novelty detection in network traffic. *arXiv preprint arXiv:2006.16993v2*, 2021.

[78] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *USENIX Annual Technical Conference*, 2019.

[79] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[80] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Automating in-network machine learning. *arXiv preprint arXiv:2205.08824v1*, 2022.

[81] Weiping Zheng, Jianhao Zhong, Qizhi Zhang, and Gansen Zhao. MTT: An efficient model for encrypted network traffic classification using multi-task transformer. *Applied Intelligence*, 2022.

[82] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An efficient design of intelligent network data plane. In *USENIX Security Symposium*, 2023.

A Candidate Features

Table 4 lists the set of candidate features used in our evaluations. We aggregate common network flow features used throughout networking research for ML-based traffic analysis [3, 15, 16, 51, 64, 68, 72]. These are also commonly supported by open source tools, and are not specific to any particular use case. However, for privacy reasons related to our live traffic experiments, we restrict these to various types of summary statistics to avoid saving packet payloads to disk.

B Dataset Collection

All experiments with live traffic are performed on a server running Ubuntu 20.04, with a dual Xeon Gold 6248R 3GHz CPU, 384 GB of memory, and a 100GbE Mellanox ConnectX-5 NIC. All experiments using offline datasets are performed on a server with a dual Xeon Gold 6154 3GHz CPU and 384 GB of memory, running Ubuntu 20.04.

Web Application Classification. For `app-class`, we classify the following applications: Netflix, Twitch, Zoom, Microsoft Teams, Facebook, Twitter, or "other." Connections are labeled using the SNI from the TLS handshake, and only statistical features listed in Table 4 are collected (see Appendix H for ethical considerations involving live traffic). We

Feature	Description	In mini cand. set
dur	total duration	yes
proto	transport layer protocol	no
s_port	src port	no
d_port	dst port	no
s_load	src → dst bps	yes
d_load	dst → src bps	no
s_pkt_cnt	src → dst packet count	yes
d_pkt_cnt	dst → src packet count	no
tcp_rtt	time between SYN and ACK	no
syn_ack	time between SYN and SYN/ACK	no
ack_dat	time between SYN/ACK and ACK	no
s_bytes_sum	src → dst total bytes	yes
d_bytes_sum	dst → src total bytes	no
s_bytes_mean	src → dst mean packet size	yes
d_bytes_mean	dst → src mean packet size	no
s_bytes_min	src → dst min packet size	no
d_bytes_min	dst → src min packet size	no
s_bytes_max	src → dst max packet size	no
d_bytes_max	dst → src max packet size	no
s_bytes_med	src → dst median packet size	no
d_bytes_med	dst → src median packet size	no
s_bytes_std	src → dst std dev packet size	no
d_bytes_std	dst → src std dev packet size	no
s_iat_sum	src → dst total packet inter-arrival time	no
d_iat_sum	dst → src total packet inter-arrival time	no
s_iat_mean	src → dst mean packet inter-arrival time	yes
d_iat_mean	dst → src mean packet inter-arrival time	no
s_iat_min	src → dst min packet inter-arrival time	no
d_iat_min	dst → src min packet inter-arrival time	no
s_iat_max	src → dst max packet inter-arrival time	no
d_iat_max	dst → src max packet inter-arrival time	no
s_iat_med	src → dst median packet inter-arrival time	no
d_iat_med	dst → src median packet inter-arrival time	no
s_iat_std	src → dst std dev packet inter-arrival time	no
d_iat_std	dst → src std dev packet inter-arrival time	no
s_winsize_sum	src → dst sum of TCP window sizes	no
d_winsize_sum	dst → src sum of TCP window sizes	no
s_winsize_mean	src → dst mean TCP window size	no
d_winsize_mean	dst → src mean TCP window size	no
s_winsize_min	src → dst min TCP window size	no
d_winsize_min	dst → src min TCP window size	no
s_winsize_max	src → dst max TCP window size	no
d_winsize_max	dst → src max TCP window size	no
s_winsize_med	src → dst med TCP window size	no
d_winsize_med	dst → src med TCP window size	no
s_winsize_std	src → dst std dev TCP window size	no
d_winsize_std	dst → src std dev TCP window size	no
s_ttl_sum	src → dst sum of IP TTL values	no
d_ttl_sum	dst → dst sum of IP TTL values	no
s_ttl_mean	src → dst mean TTL	no
d_ttl_mean	dst → src mean TTL	no
s_ttl_min	src → dst min TTL	no
d_ttl_min	dst → src min TTL	no
s_ttl_max	src → dst max TTL	no
d_ttl_max	dst → src max TTL	no
s_ttl_med	src → dst median TTL	no
d_ttl_med	dst → src median TTL	no
s_ttl_std	src → dst std dev TTL	no
d_ttl_std	dst → src std dev TTL	no
cwr_cnt	number of packets with CWR flag set	no
ece_cnt	number of packets with ECE flag set	no
urg_cnt	number of packets with URG flag set	no
ack_cnt	number of packets with ACK flag set	no
psh_cnt	number of packets with PSH flag set	no
rst_cnt	number of packets with RST flag set	no
syn_cnt	number of packets with SYN flag set	no
fin_cnt	number of packets with FIN flag set	no

Table 4: Candidate feature set $\mathcal{F}$ containing 67 commonly used flow features. We indicate the six that are used in the smaller candidate set for ground truth analyses.

use Retina [72] for data collection, which we run for 30 seconds with 12.5% flow sampling, and an additional 10 minutes with 50% flow sampling filtered on the target applications to help collect a more balanced dataset. Flow sampling reduces the effective ingress network throughput while maintaining per-connection consistency, and is done entirely in the NIC using hardware filters [72]. We avoid collecting at full network throughput to ensure that no packets are dropped in the data collection phase. In total, we collected 2M samples of connection data over 50 different packet depths.

IoT Device Recognition. For `iot-class`, we classify one of 28 IoT device types using the UNSW IoT dataset [65]. We use the September 2016 traces in our evaluations, which include approximately 134K connections. Models are trained and evaluated using data from eight days of packet traces.

Video Startup Delay Inference. We use the dataset collected by Bronzino et al. [15] for `vid-start`, focusing exclusively on YouTube traffic, where each video session consists of a single TCP connection. The final dataset comprises 4,287 connections, capturing a wide range of startup delay times. Delay times range from 315 ms to 54 seconds at P99, with the maximum observed delay being 14 minutes.

C Model Training

For DT and RF model training, we use scikit-learn's DecisionTreeClassifier and RandomForestClassifier with default parameters and tune the maximum tree depth from the set {3, 5, 10, 15, 20}. The RandomForestClassifier is configured with 100 estimators. To integrate with the Rust-based packet capture and feature extraction stages, the tuned DT and RF models are retrained in Rust using the SmartCore [1] library.

DNN hyperparameters are tuned over the following values: batch size {16, 32, 64}, learning rate {0.001, 0.01}, dropout rate {0.2, 0.4, 0.6, 0.8}, L2 regularization {0.1, 0.5}, and number of neurons in each hidden layer {4, 8, 16}.

D Measurement Details

End-to-End Inference Latency. We implement two versions of inference latency measurement, depending on the use case. For `app-class`, we record the arrival timestamp of the SYN packet, and subtract it from the timestamp of the final prediction output by the model. Since we do not have access to live traffic from the `iot-class` dataset, we compute the end-to-end inference latency by taking the sum of the pipeline execution time, model inference time, and packet inter-arrival times up to the specified connection depth. The inter-arrival times are calculated from packet timestamps in the traces.

Zero-Loss Throughput. Since we are unable to control the input traffic rate on our live network, we choose to restrict our serving pipelines to a single core to differentiate the performance of each optimization method. This ensures that we

can find an upper bound on the throughput since no solution will be able to saturate the input traffic rate. To measure the zero-loss throughput, we leverage Retina's flow sampling capabilities by starting at the full traffic rate and slowly decreasing the percentage of flows randomly dropped by the NIC until we observe zero packet loss for 30 seconds. We repeat this for multiple trials and take the average zero-loss throughput sustained by the traffic analysis pipeline.

Execution Time. Pipeline execution time is a measure of total CPU time spent in the serving pipeline, excluding time spent waiting for packets to arrive. We measure this by inserting calls to query the Read Time-Stamp Counter register at the start and end of each packet processing step and the model inference stage.

E Optimization Wall-Clock Time

Wall-clock time depends heavily on the application use-case, model type, number of samples explored, and the concrete definitions of $\texttt{cost}(x)$ and $\texttt{perf}(x)$. For reference, Table 5 reports the breakdown in time elapsed for CATO to compute the Pareto fronts depicted in Figure 5d and Figure 7, which target different use cases and system cost metrics. As expected, execution time is mostly consumed by the Profiler, which, on each iteration, generates a fresh serving pipeline, trains and evaluates the model, and measures the end-to-end systems costs. We deem this a worthwhile trade-off because the time spent by the Profiler to validate the systems cost of each sampled solution ensures that the resulting Pareto-optimal pipeline can meet real-time performance requirements. The BO-guided sampling that determines the next feature representation for the Profiler to evaluate adds between 1.4 and 55 seconds per iteration depending on the search space.

F Reproducing Traffic Refinery

We reproduce key components of the Traffic Refinery [14] framework for evaluation. Traffic Refinery defines several feature classes that contain features commonly used in ML-based traffic analysis. Specifically, we replicate the PacketCounter (PC), PacketTiming (PT), and TCPCounter (TC) feature classes using subsets of our candidate feature set. PC includes all packet and byte counters, PT includes all packet inter-arrival statistics, and TC includes all flag counters, window size statistics, and RTT. We simulate using Traffic Refinery by manually aggregating feature classes, varying the packet depths, and measuring the predictive performance and pipeline execution time using CATO's Profiler. While Traffic Refinery also has the ability to profile state and storage costs, we focus on execution time in our evaluation since it is a shared cost metric in both frameworks.

Use case / # cand. features **Systems cost metric:**	`app-class` / 67 zero-loss throughput	`iot-class` / 6 processing time
Preprocessing	22.4 s	4.1 s
Opt. Iteration (50×)		
BO sample	55.5 s	1.4 s
Pipeline generation	53.1 s	46.5 s
Measure $\texttt{perf}(x)$	29.6 s	26.4 s
Measure $\texttt{cost}(x)$	546.7 s	70.3 s
Total elapsed	9.5 h	2 h

Table 5: CATO optimization wall-clock times. BO is well-suited for expensive-to-evaluate objective functions, such as $\texttt{perf}(x)$ and $\texttt{cost}(x)$.

G Simulated Annealing Details

We describe our simulated annealing [38] implementation SIMA from Section 5.3. SIMA starts with a random feature representation x as the current "best" point. On each iteration i, it samples a neighbor point x_i and measures $\texttt{cost}(x_i)$ and $\texttt{perf}(x_i)$. Neighbors are sampled by randomly perturbing either the feature set or the packet depth with equal probability:

- Feature set perturbation: Add, remove, or replace a feature at random.
- Packet depth perturbation: Move up to some maximum step size away from the current packet depth, with the maximum step size decreasing linearly from the maximum packet depth as more samples are explored. This allows for more exploration earlier in the search.

Since our optimization is multi-objective, we adjust the standard simulated annealing neighbor acceptance criterion as follows: If the neighboring point dominates the current point across both objectives, it is accepted as the new current point. Otherwise, it is still accepted with probability $\mathbb{P}(x, x_i, T_i) = \exp(f(x) - f(x_i))/T_i$, where $f(x)$ is an equal weighted combination of $\texttt{perf}(x)$ and $\texttt{cost}(x)$, and T_i is the temperature at iteration i. Consistent with simulated annealing algorithms, the temperature gradually decreases as the search space is explored. This mechanism allows for non-dominating solutions to still be accepted with higher probability at the beginning of the search process, preventing SIMA from getting trapped in local minima. We empirically tune SIMA with different initial temperatures and cooling schedules, and choose $T_0 = 1$ and $T_{i+1} = 0.99T_i$.

Figure 8 (Section 5.3) reveals that while SIMA is less sample efficient than CATO, it is generally more efficient than RAND. However, RAND catches up after approximately 1500 sample explorations, likely due to SIMA's reduced ability to explore new feature representations as the temperature decreases.

H Ethical Considerations

As part of our experiments with high-speed network traffic, we evaluated the performance of models against live campus

network traffic. The candidate flow features we use included only aggregate flow statistics and an application name derived from the SNI field in TLS handshakes. We never captured or analyzed client IP addresses, viewed any individual flows or connection records, stored any packets to disk, or investigated human behavior; our IRB has ruled that this type of analysis does not constitute human subjects research. Nonetheless, we took steps to ensure the security and privacy of campus users. All live traffic experiments were isolated to a single hardened server that was deployed in partnership with our campus networking and security teams in order to not increase the attack surface for users.

Resolving Packets from Counters: Enabling Multi-scale Network Traffic Super Resolution via Composable Large Traffic Model

Xizheng Wang[1,2], Libin Liu[2], Li Chen[2], Dan Li[1], Yukai Miao[2], Yu Bai[2]

[1]*Tsinghua University* [2]*Zhongguancun Laboratory*

Abstract

Realistic fine-grained traffic traces are valuable to numerous applications in both academia and industry. However, obtaining them directly from devices is significantly challenging, while coarse-grained counters are readily available on almost all network devices. None of existing work can restore fine-grained traffic traces from counters, which we call network traffic super-resolution (TSR). To this end, we propose ZOOMSYNTH, the first TSR system that can achieve packet-level trace synthesis with counter traces as input. Following the basic structure of the TSR task, we design the Granular Traffic Transformer (GTT) model and the Composable Large Traffic Model (CLTM). CLTM is a tree of GTT models, and the GTT models in each layer perform upscaling on a particular granularity, which allows each GTT model to capture the traffic characteristics at this resolution. Using CLTM, we synthesize fine-grained traces from counters. We also leverage a rule-following model to comprehend counter rules (*e.g.* ACLs) when available, guiding the generations of fine-grained traces. We implement ZOOMSYNTH and perform extensive evaluations. Results show that, with only second-level counter traces, ZOOMSYNTH achieves synthesis quality comparable to existing solutions that takes packet-level traces as input. CLTM can also be fine-tuned to support downstream tasks. For example, ZOOMSYNTH with fine-tuned CLTM outperforms the existing solution by 27.5% and 9.8% in anomaly detection and service recognition tasks, respectively. To promote future research, we release the pre-trained CLTM-1.8B model weights along with its source code.

1 Introduction

Fine-grained traffic traces are valuable to numerous applications in both academia and industry. Packet-level traces can benefit performance evaluation of systems and algorithms in all layers of the networking stack, from packet scheduling to congestion control. Realistic traffic traces are also useful in optimizing various tasks, such as traffic engineering [48], network telemetry [53, 54], anomaly detection [75], and service recognition [27]. However, obtaining fine-grained traffic traces from devices is challenging due to factors such as business confidentiality, device processing/capacity limits, and privacy constraints [77].

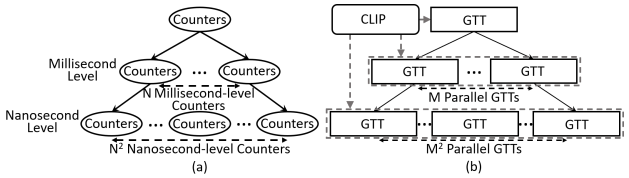


Figure 1: CLTM is a tree of Granular Traffic Transformers (GTTs).

Meanwhile, counters are readily available on almost all network devices and can be collected by network management systems (NMSs) [26, 43] (§2.2). On any network interface, we can easily find counters that accumulate byte-count or packet-count statistics. Counters are usually deeply embedded in the chip registers of the networking hardware; thus, they are high-performance, accurate, and reliable. Networking engineers frequently use them to monitor network behaviors or diagnose failures. But the problem with counters is that they are coarse–grained — the frequency at which NMSs collect them is typically in tens of seconds or even minutes. Without involving deep modification using SDK of the networking chip, a typical operating system of a networking device can only report counters on the scale of seconds. This is far from enough for the collection of packet-level fine-grained traces.

Given the abundance of counter statistics in NMSs and the scarcity of fine-grained traffic traces, we seek to answer a natural research question: *Can we restore fine-grained traffic traces from coarse-grained counters?* Although there have been continuous efforts in the synthesis of packet traces [28, 42, 50, 51, 71, 74, 77], none of them can recreate packet-level traffic from coarse-grained counter statistics.

Recent advances in generative artificial intelligence give us hope: for computer vision, image super-resolution (ISR) is a similar and well-studied task [35, 49]. ISR is the task of recreating high-resolution images (*e.g.* 1080p) from low-resolution ones (*e.g.* 360p), and it has many existing algorithms and models that achieve high-quality results [78, 79].

However, our experiments show that ISR algorithms and models cannot be applied directly to network traffic super-resolution (TSR) (§2.3). We summarize the unique challenges of TSR:

C1 **Input Format**: ISR and TSR work on different domains

of data. A TSR system should take in a time-series (counters), and output a time-series (packets), while an ISR system deals with images (grids of pixels).

C2 **Extreme Upscaling Ratio**: The required upscaling ratio of TSR is many orders of magnitude larger than that of ISR. Packet-level traces have timestamps at resolutions of nanoseconds. To recreate a packet trace from a per-second counter trace, the upscaling ratio is 10^9. In contrast, the maximum ISR upscaling ratio of the popular Stable Diffusion model is only 16 [21].

C3 **Generality**: Network traffic in different network environments, such as the Internet, data centers, and access networks, exhibits diverse characteristics. For TSR, even when the coarse-grained traffic statistics remain consistent, the features of the corresponding fine-grained traffic should be different.

In addition, we define three more design requirements that serve to broaden our scope of application.

R1 **Multi-scale Synthesis**: Some downstream tasks do not require nanosecond-level timestamps, and a coarser-grained trace would suffice. For example, traffic engineering systems do not need traces with granularity of nanoseconds but will benefit if we can upscale a trace's resolution from 1 hour to 1 minute. Thus, we intend to generate traces with different upscaling ratios.

R2 **Understanding Counter Rules**: Due to the finite number of registers in network devices, operators often use user-defined rules (*e.g.* ACL) to specify which flows to track. Understanding the semantic meaning of these rules can guide the TSR to generate traces adhering to the rules. The rules can be accessed by operators from their configuration databases or network devices [76], and shared with researchers anonymously [72].

R3 **Real-time Synthesis**: Recreating high-resolution traffic from counters in real time can enable various downstream tasks such as network telemetry, failure diagnosis, and network digital twins [25].

Clearly, tackling these challenges and requirements simultaneously is ambitious, and an added difficulty is that we also need to design a new base model architecture for TSR, given that the existing models for ISR are unsatisfactory. We draw inspiration from the basic structure of the problem: A coarse-grained trace is the summary of many fine-grained traces. For example, the trace of a second-level counter is the *summary* of ten 100ms-level counters. This summarizing relationship forms a tree structure (Fig. 1), and we design the base model for TSR in the same way, which we name the Composable Large Traffic Model (CLTM).

The core of CLTM is a tree of Granular Traffic Transformers (GTTs). Each layer of the tree consists of the same GTT modules only look at the traffic characteristics at this particular granularity. Concentrating on a particular scaling phase (e.g. 1s to 100ms) better captures the characteristics of traffic at this granularity. For instance, with the same $10\times$ upscaling factor, ten 100ms-level counters generated from a second-level counter may have relatively average values, whereas the subsequent 10ms-level counters may show a more uneven distribution.

With CLTM, we design and implement ZoomSynth, the first network traffic synthesis system that supports TSR. ZoomSynth effectively tackles the aforementioned challenges and meets the specified requirements through a strategic combination of innovative ideas in addition to CLTM. Firstly, the tree of GTTs in CLTM addresses **C1**, **C2**, and **R1**, simultaneously. For **C3**, to adapt to new synthesis scenarios, we incorporate the Low-Rank Adaptation Model (LoRA) [40]. This adaptation model borrows the concept from domain adaptation in large language models (LLMs). Furthermore, we develop a rule-following model based on Contrastive Language-Image Pre-training (CLIP) [59], linking counter rules with the generation of traffic traces, to understand counter rules (**R2**). Finally, to achieve real-time generation (**R3**), we introduce pipeline-parallelism to accelerate CLTM's inference.

In summary, we make the following contributions.

- We illustrate the practical need for TSR and the availability of counters, underscoring the importance of resolving fine-grained traces from coarse-grained counters (§2).
- We design and implement ZoomSynth[1], the first TSR system that facilitates multi-scale TSR based on counters, with counter rules available as optional input (§3, §6).
- We propose CLTM, a novel base model architecture for TSR, which breaks down the TSR process into multiple phases, each employing a GTT tailored to the specific upscaling stage (§4).
- We evaluate ZoomSynth with extensive testbed experiments (§7) and confirm it meets the design goals. Notably, across all distributional metrics and traces [6, 7, 9, 22, 31, 55, 57, 63], ZoomSynth achieves packet trace synthesis with up to 25.8% JSD and 20.1% EMD improvements compared to the state-of-the-art schemes [46, 77]. ZoomSynth can also achieve real-time generation with over one billion packets per second with pipelined inference.

This work raises no ethical concerns. All traces used in this paper contain no personally identifiable information (PII).

2 Background and Motivations

In this section, we first introduce the applications of TSR (§2.1). Then, we demonstrate the wide availability of counters (§2.2) and show that existing approaches cannot be applied to TSR (§2.3). Finally, we motivate the design choices of ZoomSynth for TSR (§2.4).

[1] https://github.com/wxzisk/ZoomSynth_NSDI2025

2.1 Applications of TSR

We believe that any application that requires realistic fine-grained traffic traces can benefit from TSR, because TSR can turn counter traces into fine-grained packet traces, and counter traces are much easier to collect. Previous works [44, 50, 71, 74, 77] have explored a range of applications, including telemetry algorithms, the evaluation of machine learning models [77], and service recognition [46].

Accurate and real-time TSR can also enable new applications. One example is failure diagnosis using counters. Previously, counters could only offer operators a low-resolution view of network state. With TSR, operators can *expand* the counter trace into a high-likelihood packet trace, which can help understand and diagnose network events. TSR is also a potential enabling technology for network digital twins (NDTs) [25]. A major roadblock for NDT is the large amount of data that needs to be transferred from the physical network to the NDT system to realistically recreate the traffic in the digital twin, especially when physical networks today have interfaces $\geq$100Gbps. Using TSR, NDT systems can collect only the coarse-grained data from the physical network, and restore high-resolution traffic.

2.2 Availability of Counters

Counters are almost available on all network devices and network operating systems (NOS) and software. We conclude this by surveying the device configuration manuals of Cisco, one of the mainstream vendors, and widely used open-source NOS and software, such as software for open networking in the cloud (SONiC) [20], Linux kernel, and data plane development kit (DPDK) [10].

In the survey, we find 41 out of 42 categories of router products of Cisco listed at its official site [5] exactly declare supporting packet counters in their configuration manuals. Only the virtual router emulator states that some hardware counters are not supported by it [8]. We summarize the router products, configuration commands to display counters, and the corresponding online configuration manuals, in Table 6 in appendix §.5. Also, the vendor NOS, such as Cisco IOS XR [1, 2] and Cisco IOS XE [3, 4], that these routers run support the standard management protocols, such as SNMP [43] and NetFlow [26], which is commonly used to collect counters by the network management systems.

Besides, existing open-source NOS and software [10, 12, 20] support multiple types of counters as well. Taking SONiC as an example, its switch abstraction interface (SAI) defines eight types of counters [17] in its router interface API [19], which include the ingress or egress byte counters, ingress or egress packet counters, byte or packet counters for packets having errors on router ingress, and byte or packet counters for packets having errors on router egress. And Linux kernel implements 25 types of counters including packet and byte counters [12], while DPDK supports packet and byte counters in its packet framework library [15, 18].

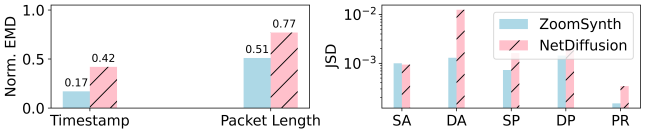


Figure 2: NetDiffusion's performance for TSR to achieve 10^9 end-to-end upscaling.

2.3 Applicability of ISR to TSR

Recent advances in ISR [35, 49] address a similar research question to TSR, aiming to restore high-resolution images (*e.g.*, 1080p) from low-resolution images (*e.g.*, 360p). Many ISR algorithms and models have achieved high-quality results [78, 79]. This prompts us to investigate whether existing ISR algorithms and models can be applied directly to TSR.

We notice a recent pioneering work that applies ISR to traffic synthesis [45]. NetDiffusion utilizes Stable Diffusion 1.5 [65] as the base model, which is recognized as one of the state-of-the-art ISR models. We deploy the open-source implementation of NetDiffusion [13] and evaluate its performance for TSR, *i.e.*, synthesizing packet-level traces from counters. To ensure consistency, we download the model and follow the same fine-tuning process as NetDiffusion, while employing the same datasets as those used by ZOOMSYNTH. Fig. 2 illustrates the experimental results in terms of Jensen-Shannon Divergence (JSD) [56] and Earth Mover's Distance (EMD) [66], and we also show the performance of ZOOMSYNTH. For these two metrics, lower is better, and we observe that NetDiffusion has worse performance than ZOOMSYNTH.

We dive into the implementation of NetDiffusion and identify two primary reasons. First, there is a mismatch between input formats (or data representation) required by the Stable Diffusion model and the inherent nature of traffic data, which is naturally represented as time series. NetDiffusion needs to fold the time series into matrices, which may induce the ISR model to capture unwanted or non-existent relationship between data points. Second, NetDiffusion directly uses the pre-trained model based on image datasets as the basic model and only fine-tunes it for traffic synthesis. Thus, the direct application of an ISR model for TSR, without sufficient customization to the characteristics of traffic data, results in unsatisfactory performance.

2.4 ZOOMSYNTH Design Motivations

We first summarize the challenges and requirements in §1 into five design goals, and then we motivate the design choices of ZOOMSYNTH to achieve these goals.

Design goals. Based on §1, we summarize the goals for ZOOMSYNTH.

G1 **Counter-to-packet Synthesis**: This goal ensures practical applicability, given that many downstream tasks necessitate packet-level traces and coarse-grained counters are widely available.

G2 **Multi-scale Synthesis**: In addition to packet traces

(nanosecond trace), ZoomSynth should generate traffic traces with varying upscaling ratios.

G3 **Counter-rule-based Synthesis**: ZoomSynth should comprehend the semantic meaning of these rules, such as ACL policies, and generate traces that adhere to them.

G4 **Generality**: Traffic traces in different network environments have different characteristics. ZoomSynth should adapt to the features of these varying environments and generate traces with user-defined granularity.

G5 **Real-time Synthesis**: Real-time synthesis enables ZoomSynth to be used in real-time tasks. We define real-time synthesis as follows: for a counter collected every n seconds ($n \geq 1$), ZoomSynth should generate the corresponding packet-level trace within n seconds.

To achieve the goals, our five design choices for ZoomSynth are as follows:

1. GTT-based TSR: With the unsatisfactory performance of ISR models for TSR, we seek a model which is more suitable to handle time-series data. We choose Transformer [70] for ZoomSynth (§4.1), because Transformer can capture long-range dependencies in time-series sequences, and is well-suited for modeling time-series traffic traces where temporal dependencies span over extended periods [67, 68].

We do not use a single Transformer model for TSR: each Transformer model is responsible for only one upscaling granularity (*e.g.*, 1s to 100ms), and it captures the characteristics of traffic at this particular granularity. We name this type of models the Granular Traffic Transformer (GTT). We use GTT $^{k}_{\delta}$ to denote a GTT with upscaling factor k and generation resolution δ. For example, a GTT $^{10}_{100\text{ms}}$ can transform a trace with 1s resolution to one with 100ms resolution.

2. Tree-structured CLTM Model: A coarse-grained counter summarizes the underlying fine-grained counters, forming a classic tree structure at different granularities, as illustrated in Fig. 1(a). With this observation, we then compose GTTs at different granularities into a tree, which we name the Composable Large Traffic Model (CLTM).

CLTM can achieve high upscaling ratios (**G1**). CLTM contains a tree of GTTs, illustrated in Fig. 1(b). This hierarchical structure allows the upscaling ratios to be multiplied across GTTs in distinct layers of CLTM. Consequently, CLTM can achieve exponential increases in upscaling ratios by incorporating additional layers of GTTs. This enables CLTM to attain high upscaling ratios.

CLTM supports multi-scale traffic synthesis (**G2**). Within the CLTM tree, each layer of GTTs is dedicated to a specific granularity. Each layer can generate traffic traces with the targeted granularity, enabling multi-scale traffic synthesis.

3. Counter Rule Understanding with CLIP: **G3** requires us to use counter rules (*e.g.*, text strings of ACL policies) to guide the generation of traffic traces, which is a multimodality task of two domains: text and time-series. We learn from mainstream multimodal models [32, 52], and use CLIP [59] to develop a conversion model to link counter rules with traffic traces. This ensures that synthetic traffic aligns with specified rules in every layer of the CLTM (§4.2).

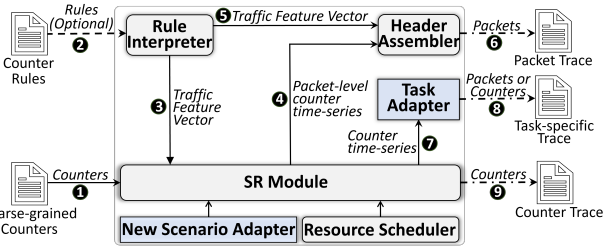


Figure 3: ZoomSynth end-to-end workflow (New Scenario Adaptor and Task Adaptor only run in the training phase). Note that counter rules are not mandatory input but optional.

4. Generality with a LoRA-based Approach: To generalize ZoomSynth, we seek a method that allows users to use their own data to *fine-tune* CLTM, since it is impossible to collect data in all scenarios. Thus, for **G4**, we leverage a LoRA-based approach, borrowing from the LLM literature [40], to facilitate lightweight fine-tuning of CLTM for adapting to new scenarios (appendix §.1.5).

5. Pipeline-parallel Inference: To achieve **G5**, we introduce pipeline paralellism and execute GTTs in different layers of CLTM concurrently. We dynamically allocate computational resources to different layers by scheduling GTTs automatically to ensure efficient inference. We also develop a prefetching technique to further improve the performance (§4.3).

3 ZoomSynth Overview

We present the overview of ZoomSynth in Fig. 3. ZoomSynth is a TSR system designed for fine-grained traffic synthesis using counter traces and counter rules.

3.1 Architecture and Components

ZoomSynth has six key modules, shown in Fig. 3, and we describe them as follows:

- **SR Module**: The SR Module runs the CLTM model. It takes as input a coarse-grained counter time-series. Through the application of the SR module, each counter within the time series undergoes an upscaling process, resulting in a more fine-grained counter time series. Users' required upscaling ratio can be achieved by composing the GTT models to the target number of layers. We set the upscaling factor of each layer to a fixed value of 10 and justify this in appendix §.1.2.
- **Rule Interpreter**: The Rule Interpreter runs a CLIP-based rule-following model. It takes counter rules as input and outputs a feature vector representing the traffic traces complying with the rules. As shown in Fig. 3, the output vector of the Rule Interpreter is fed into each layer of CLTM to help align the time-series features of counters, while in

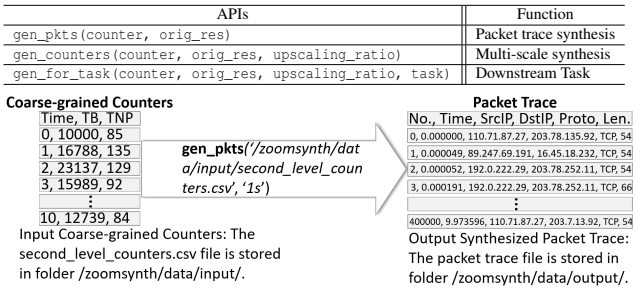


Figure 4: APIs of ZOOMSYNTH's applications and an example for packet trace synthesis from second-level counters using the API `gen_pkts(·)` of ZOOMSYNTH (TB: Total Bytes, TNP: Total Number of Packets).

the header generation process, it is used by the Header Assembler to constrain the scope of the packet headers.

- **Header Assembler**: The Header Assembler runs a GPT-2 based header generation model. As shown in Fig. 3, it takes the outputs from both CLTM and Rule Interpreter as input and generates packet headers to synthesize packet traces.
- **Resource Scheduler**: This module allocates resources to run the above modules. We use pipeline parallelism (a type of model parallelism) to accelerate the inference of CLTM. For the other models, running Rule Interpreter, Header Assembler and Task Adapter is trivial as their model is small and do not need model parallelism, while New Scenario Adaptor only runs offline and does not affect the inference performance of real-time traffic synthesis.
- **Task Adaptor**: This module runs task-specific model with domain knowledge. It can take fine-grained traces with various granularity as input, and leverages task-specific model to align the traces to the domain of the downstream task. Taking an ML-based anomaly detection task as an example, the Task Adaptor uses the ML-based anomaly detection model trained with real traffic data to align the type labels of the fine-grained traces.
- **New Scenario Adaptor**: This module aims to make ZOOMSYNTH adapt to new scenarios efficiently. It leverages a LoRA-based approach to update the CLTM with a small amount of traffic traces of the new scenarios, while purely retraining the models of Header Assembler and Task Adapter as they are small and quick to train. We explore the efficiency of this module in appendix §.4.1.

3.2 Workflow

User of ZOOMSYNTH can programmatically determine their desired resolution by configuring the CLTM. In this way, ZOOMSYNTH supports arbitrary upscaling ratios and can also act as *base model* for many downstream tasks. For each application, ZOOMSYNTH needs to adapt its relative components to synthesize required traces from coarse-grained counters. ZOOMSYNTH provides APIs summarized in Fig. 4 to run the applications. We summarize the workflow of ZOOMSYNTH in Fig. 3 and describe its applications as follows.

Packet Trace Synthesis: The finest-grained traces that ZOOMSYNTH can generate are packet trace synthesis with nanosecond resolution. ZOOMSYNTH utilizes Rule Interpreter, SR Module, and Header Assembler to compose the assembly line for packet trace synthesis. It takes the ❶ counters or both ❶ counters and ❷ counter rules as input, and processes them with ❸ ❺ the Rule Interpreter, ❹ SR Module, and ❻ Header Assembler in succession and outputs packet traces. To run these modules, users can use the `gen_pkts(·)` API, where the parameter `counter` is the path to the file of counters and rules and the parameter `orig_res` is the granularity of the input counters.

Taking Fig. 4 as an example to illustrate packet trace synthesis based on second-level counters by using `gen_pkts(·)` API of ZOOMSYNTH. When invoking the API, users need to specify the input file of the coarse-grained counters with its granularity, ZOOMSYNTH outputs the packet trace after running the API.

Multi-scale Fine-grained Trace Synthesis: Fine-grained counter traces are counter time-series on a finer scale than the input counter time-series. They follow the same counter rules and do not contain packet header information. Therefore, ZOOMSYNTH only takes the ❶ counters or both ❶ counters and ❷ counter rules as inputs, and adopts ❸ the Rule Interpreter and ❾ the SR Module to synthesize fine-grained counter traces. Users can adopt the API `gen_counters(·)` to run the application, where the parameter `upscaling_ratio` is the target end-to-end upscaling ratio.

Downstream Task: Specific downstream network tasks generally require task-specific fine-grained traffic traces. Before synthetic traces are used for a specific task, ZOOMSYNTH leverages Task Adapter to align them to task-specific characteristics with domain knowledge. Therefore, ZOOMSYNTH takes the ❶ counters or both ❶ counters and ❷ counter rules as inputs, and adopts ❸ Rule Interpreter, ❼ SR Module, and ❽ Task Adapter to synthesize counter or packet traces needed by downstream tasks. ZOOMSYNTH provides the API `gen_for_task(·)` for users to specify their input file of coarse-grained counters, counters' original resolution, target end-to-end upscaling ratio, and the downstream task model. In §5.2, we implement three downstream tasks based on ZOOMSYNTH, and evaluate their performance in §7.2.

4 Composable Large Traffic Model

CLTM is composed of GTT models and CLIP-based rule-following model and is the key for enabling multi-scale TSR. In CLTM, GTT is organized as a tree structure, by combining it with the rule-following model, CLTM empowers rule-based synthesis. CLTM supports real-time synthesis with pipelined inference. In the following, we illustrate its design in detail.

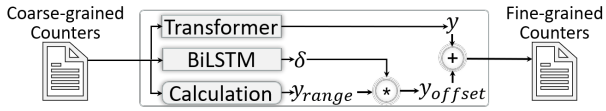


Figure 5: The structure of GTT.

4.1 GTT Model

GTT model consists of two key modules: a Transformer model [70] and a bidirectional LSTM (BiLSTM) [38]. GTT uses Transformer to learn temporal characteristics and leverages BiLSTM to deal with extreme values of counter time series. For each stage of CLTM, a GTT $\frac{k}{\delta}$ make $k\times$ upscaling for counters at a δ resolution. We describe more details about the training data preparation for GTT and the mechanism to determine the upscaling factor k in appendix §.1.1 and appendix §.1.2, respectively.

Handling Extreme Values: The counter time-series displays many extreme values, which are caused by many hidden factors, such as network stack, network functions, and scheduling policies. We observe that a Transformer cannot handle extreme values efficiently, since it tends to evenly distribute packets and byte counts, making upscaled counters tend to favor the means. To address this issue, we use a BiLSTM model to learn the distribution of extreme values, which is borrowed from extreme value prediction methods [33]. GTT integrates Transformer and BiLSTM, as shown in Fig. 5, and uses BiLSTM to learn the distribution frequency of extreme values and their deviations from the means. Unlike Transformer, which directly uses coarse-grained and the corresponding fine-grained counter time-series as training data, we leverage the same input time-series but uses the deviations from the mean of the fine-grained counter time-series as output to BiLSTM in the training phase.

As shown in Fig. 5, in the inference phase, when GTT receives the counter time-series, it feeds the input into both the Transformer and BiLSTM, and calculates the range of the counter time-series. GTT takes the offset ratios (δ) output by BiLSTM to multiply by the calculated range (y_{range}) to obtain the offsets (y_{offset}), and then add them with the output (y) of Transformer to produce the final output.

In order to make GTT models learn counter time-series distributions, we adopt both Mean Square Error (MSE) and EMD in the loss function, while guiding it to satisfy the counter equality constraints described as follows.

Counter Equality Constraints: Clearly, the counters obtained from the fine-grained counter time-series outputted by a GTT should align with the input coarse-grained counters, which relation we call counter equality constraints, denoted as $E(C_{s-1}, \hat{C}_s)$.

$$E(C_{s-1}, \hat{C}_s) = 0,$$
$$\text{where } E(C_{s-1}, \hat{C}_s) = C_{s-1} - F(\hat{C}_s).$$

Here, C_{s-1} represents the realistic input counter time-series of upscaling stage s, and $\hat{C}_s$ is the corresponding counter time-series outputted by the GTT model based on C_{s-1}. F is the counter filtering function.

Loss Function of GTT: We integrate MSE and EMD into our loss function, since MSE can make the means of GTT's output closely align with the target values of the ground truth, while EMD facilitates learning the intrinsic structure of the target distribution. By minimizing both MSE and EMD, the loss function guides GTTs to capture the counter time-series characteristics at each upscaling stage. Thus, the combined loss function can be expressed as:

$$L_{combine} = MSE(\hat{C}_s, C_s) + \lambda EMD(\hat{C}_s, C_s),$$

where $\hat{C}_s$ is the output of a GTT and C_s is the corresponding ground truth at the upscaling stage s, and λ is a hyperparameter to balance MSE and EMD metrics.

Besides finding model parameters that minimize $L_{combine}$ across the training data, GTTs also need to adhere to the counter equality constraints. Formally, we translate it to solving an optimization problem:

$$\min(L_{comibine}),$$
$$\text{s.t. } E(C_{s-1}, \hat{C}_s) = 0.$$

Inspired by Zoom2Net's design for knowledge augmented loss [37], in order to make GTTs learn and respect the constraints during the training phase, we employ the augmented Lagrangian method [34] to establish an augmented Lagrangian loss function:

$$L = L_{combine} + \sigma E(C_{s-1}, \hat{C}_s),$$

where σ is a hyperparameter to balance $L_{combine}$ and $E(C_{s-1}, \hat{C}_s)$. Training GTTs with the loss function L enables them to capture the consistency between the input and output counter time-series, as enforced by the counter equality constraints. Different from Zoom2Net [37], we do not need to consider inequality constraints, since counters within the same time-series do not have such constraints.

4.2 Composing GTTs and Rule-following Model into CLTM

ZoomSynth automatically composes GTTs in CLTM to the target number of stages based on upscaling ratio parameter in its API summarized in Fig. 4. For instance, when users specify the end-to-end upscaling ratio to 10^9 for a second-level counter time-series, ZoomSynth automatically composes GTT $\frac{10}{100\text{ms}}$, GTT $\frac{10}{10\text{ms}}$, ..., and GTT $\frac{10}{1\text{ns}}$ into CLTM in a hierarchical manner. The number of GTTs increases by a factor of k in each subsequent layer, which guarantees that each GTT processes the same size of counter time-series.

Rule-following Model: The counters are the summarization of traffic traces processed following specific counter rules. Properly understanding the counter rules and applying them to the TSR process benefits the synthesis process when restoring fine-grained trace from counters. This is similar to text-based image synthesis, which utilizes CLIP [59] to align the embeddings of images and their corresponding textural descriptions, thereby establishing an association between them. Inspired by

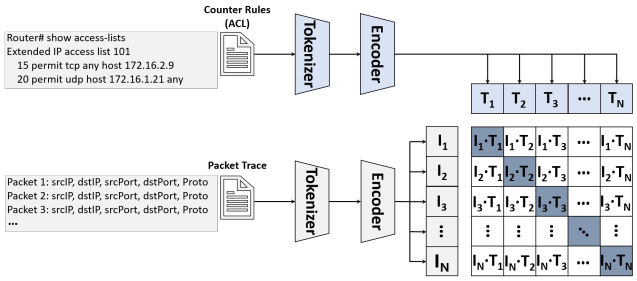


Figure 6: The pre-training process of rule-following model.

this, we employ CLIP to construct a conversion model, called the rule-following model, that bridges counter rules and traffic traces. The rule-following model is fine-tuned using counter rules and traffic traces to understand their relationships in a shared latent space.

Fig. 6 shows the pre-training process of rule-following model. The two Transformer-based encoders with 63M parameters encode counter rules and packet trace to the vectors $T=(T_1,T_2,...,T_N)$ and $I=(I_1,I_2,...,I_N)$, respectively, and map them to the same latent space. Here, we set N to 256. In the training phase, we minimize the cosine similarity of T and I and obtain a similarity matrix of counter rules and packet trace. In the inference phase, we encode the counter rules with the encoder to its vector representation T and convert T to another vector representation using the similarity matrix.

In order to prepare the training data for the rule-following model, we process the original pcap data with specific counter rules and generate packet traces which adhere to them. After finishing the training, we compose the model into CLTM as shown in Fig. 1. We compose GTTs and the pre-trained rule-following model into CLTM for training and inference to achieve multi-scale synthesis based on both counters and counter rules.

4.3 Pipelined Inference of CLTM

Fig. 7 showcases the task scheduling of ZOOMSYNTH on 4 GPUs for pipelined inference of CLTM, which is for packet trace synthesis from second-level counters. A task is defined as a GTT model for processing a fixed size of counter time-series. We use a block to represent a task in Fig. 7, and the number within the block indicates the specific GTT model a task runs. For instance, task belongs to stage 1 of CLTM run GTT $^{10}_{100ms}$. The number of tasks increases by multiplying by the upscaling factor k along with the increase of CLTM's stages. Here, we assume k=10.

ZOOMSYNTH follows two simple rules to schedule tasks on GPUs. (1) Prioritize tasks running GTT models with coarse-grained generation resolutions, because finer-grained tasks depend on them. (2) Prioritize tasks whose GTTs have input data affinity with preceding tasks running on the same GPU. As shown in Fig. 7, ZOOMSYNTH can fully utilize all GPUs after the first stage.

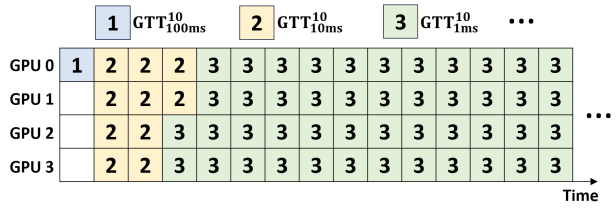


Figure 7: The task scheduling of ZOOMSYNTH on 4 GPUs for pipelined inference of CLTM. Each block represents a task, which is a GTT model for processing a fixed size of counter time-series. The number within each block represents the stage of CLTM the task belongs to. The ratio for the number of tasks between neighbouring CLTM stages is 1:10 assuming upscaling factor k=10.

5 Enabling Downstream Applications

CLTM enables downstream applications of ZOOMSYNTH, and we describe them in this section.

5.1 Header Generation

The final step of packet traces generation is to provide headers for the packets. Similarly to the existing work [62, 77], we use an embedding model to encode the header fields of the packet. Unlike previous work, the header generation model only takes care of the synthesis of five tuples (*i.e.*, source IP, destination IP, source port, destination port and protocol), since CLTM has provided the other information of packets, such as packet timestamp and size.

Packet Header Encoding: We use an embedding model similar to IP2Vec [62] to encode the categorical fields of the packet header, such as IP address, port, and protocol. The embedding is trained on packet traces collected from public datasets. Considering that the IP, port, and protocol fields in our datasets are finite, we utilize an embedding layer with a vocabulary size of 50000 and embedding dimension of 128 to serve as our header encoder. Users can retrain it to cover any additional fields with their new datasets.

Header Generation Model: Building upon the success of previous studies [30, 58] that utilize GPT-2 [60] for packet header generation, we adopt this model to reconstruct packet headers using the finest-grained counter time series from CLTM. We retrain the model using public packet traces. We detail the model training process in appendix §.2.

During the inference phase, the header generation model populates the header fields of packets using the information extracted from the raw training packet traces, alongside the timestamps and sizes derived from the synthesized finest-grained counter time-series. This ensures that the generated packet headers adhere to the traffic semantics and preserve the traffic distribution characteristics. However, this approach has limitations with respect to the generalization of packet headers, which we discuss in §9.

5.2 Task Adaptor

The Task Adaptor fine-tunes CLTM for downstream tasks, allowing the end-to-end upscaling model to be trained with a loss function that is more conducive to downstream tasks. This is similar to *fine-tuning* in the LLM literature [41]. As for most downstream tasks, packet header traces are needed, since the discrete nature of packet header information guarantees that the Task Adaptor does not entail predicting extreme values when synthesizing header fields. As a result, we do not need BiLSTM here. We use a Transformer model for Task Adaptor.

The effectiveness of synthetic traffic used for downstream tasks is the key criterion for evaluating the Task Adaptor. For this, we implement three classic downstream tasks like NetShare [77], including anomaly detection, sketch-based telemetry, and service recognition, and we use the same machine learning (ML) model as NetShare [77].

Anomaly Detection: Existing ML models for this task require packet traces with fields that include packet arrival time, packet size, and five tuples. Based on ZOOMSYNTH, we use five models [77] for anomaly detection: Decision Tree (DT), Logistic Regression (LR), Random Forest (RF), Gradient Boosting (GB), and Multi-layer Perceptron (MLP).

Sketch-based Telemetry: We implement a Count-Min Sketch (CMS) [53] and use it to estimate the proportion of heavy-hitters within the traffic. Sketch-based telemetry commonly uses traffic synthesis as a precursor task, producing a large volume of traffic with consistent characteristics.

Service Recognition: The service recognition task is a more complex ML-based task, different from the above two categories of tasks. It relies on more deep features of traffic to recognize services, instead of looking at IP addresses, port numbers, and protocol types. We adopt the same five models for anomaly detection for this task, and utilize data from CIDDS to train them for service recognition. The services are categorized into three types: email, file copying, and others.

6 Implementation

We implement a prototype of ZOOMSYNTH in Python 3.10. We use PyTorch [16] library to implement all the models in ∼950 lines of code (LOC), except for the header generation model, which we implement using the codes from Hugging Face. And for parsing datasets into packets and multi-scale counters, we implement scripts in ∼230 LOC. Table 5 in appendix §.3.1 lists the hyper-parameter setups of ZOOMSYNTH for each model. Specifically, besides the APIs summarized in Fig. 4, we also implement 6 APIs to support operations of CLTM. Based on these APIs, users can adjust the number of GTTs in each stage of CLTM by considering their hardware and TSR tasks. For the three downstream tasks, we also utilize the PyTorch library to implement the models for anomaly detection and service recognition in ∼420 LOC and implement CMS for sketch-based telemetry in ∼170 LOC.

We implement the Resource Scheduler in ∼130 LOC, which adopts `nvidia-smi` (provided by the GPU driver) to monitor GPU resources and schedule tasks to GPUs.

Datasets: We aggregate seven open-source datasets available online, namely TON (IoT traffic data) [57], CA (data of a cyber defense competition) [6], CIDDS (enterprise traffic) [7], DC (data center traffic) [9], UGR16 (traffic data from Spanish ISP) [55], CAIDA (anonymized trace on backbone network) [22], and MAWI (packet traces from WIDE backbone) [31]. The resulting dataset have a combined size of 16GB, and MAWI and CAIDA predominantly constitute this hybrid dataset, with each accounting for 45% of the data. We obtain counter traces of varying resolutions from this dataset, and the counter traces has a total size of >1.7TB.

Model Size: With k=10, input resolution 1s, and output resolution 1ns (packet-level), the required CLTM has 1.8 billion parameters. We call this model CLTM-1.8B, and release a pre-trained version[1] using the above datasets, accompanied with the source code for its training. We use this model in the evaluations, unless otherwise specified.

7 Evaluation

In this section, we seek to understand: 1) With only counter traces, can ZOOMSYNTH achieve a similar performance in traffic generation as existing traffic synthesizers that take packet-level trace as input? 2) Can the synthesized ZOOMSYNTH traces satisfy the requirements of downstream tasks? 3) Can ZOOMSYNTH achieve real-time synthesis? 4) How does the counter based on the counting bloom filter (CBF) affect TSR? 5) How does each model in ZOOMSYNTH contribute to its performance? We answer the first four questions positively. We summarize the evaluation results below.

Summary of Results:

- For packet-level trace synthesis, ZOOMSYNTH, using only counters, achieves a similar performance as NetShare that takes packet-level trace as input. ZOOMSYNTH achieves up to 25.8% and 20.1% improvements in terms of JSD and EMD, respectively. And across all four datasets, ZOOMSYNTH achieves only 30.6% reduction in JSD.
- ZOOMSYNTH relaxes the requirements for data collection for downstream tasks. With only counter traces, ZOOMSYNTH with fine-tuned CLTM outperforms NetShare by 27.5% and 9.8% in anomaly detection and service recognition tasks, respectively.
- For a CLTM-1.8B on a server with 8× A100 GPUs, for a per-second counter trace, ZOOMSYNTH can generate at most 10^9 packets in 0.966 second. This exceeds the maximum forwarding capacity of modern core network devices [8, 23, 24], proving its real-time synthesis capacity.
- We observe that CBF-based counters have negative impact on TSR performance, and find that ZOOMSYNTH still outperforms NetDiffusion and Zoom2Net, achieving im-

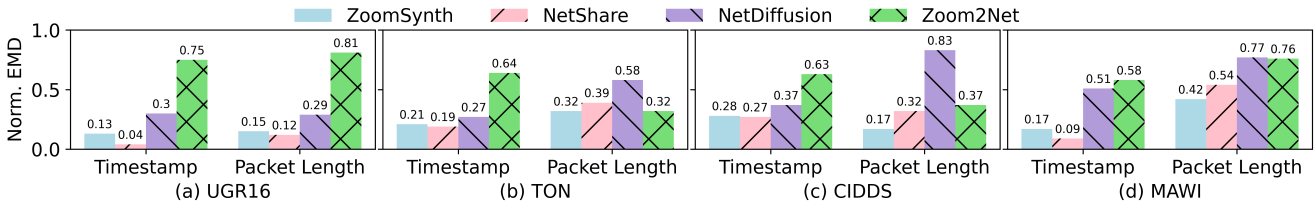


Figure 8: Earth Mover's Distance (EMD) between real and synthetic distributions on UGR16, TON, CIDDS, and MAWI, where the end-to-end upscaling ratio is 10^9.

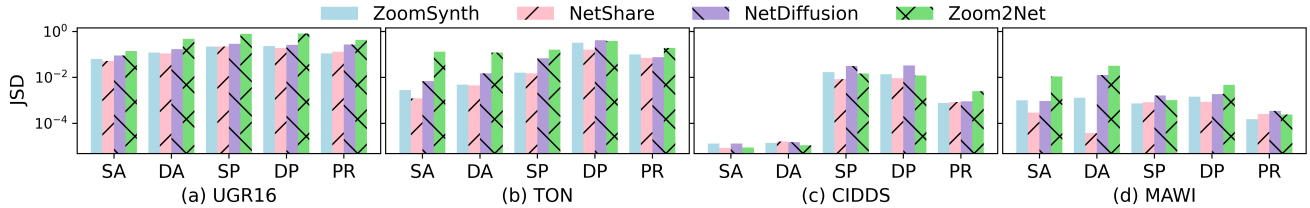


Figure 9: Jensen-Shannon Divergence (JSD) between real and synthetic distributions on UGR16, TON, CIDDS, and MAWI, where the end-to-end upscaling ratio is 10^9.

provements in EMD by 58.3% and 76.0%, and in JSD by 52.3% and 80.1%, respectively.

- We confirm the effectiveness of the header generation and rule-following models in ZOOMSYNTH.

Testbed: We use a machine equipped with 8× NVIDIA A100 GPUs, 2× 64-core 2.90GHz Intel Xeon Platinum CPUs, and 2TB DDR4 RAM. The machine runs Ubuntu 22.04 LTS and PyTorch 2.1.0.

Comparison Targets: We compare ZOOMSYNTH to three existing traffic synthesizers, each of which is the state-of-the-art work in their application scenarios. (1) *NetDiffusion* [45]: This scheme originally uses a stable diffusion model [65] to synthesize the traffic trace from text, such as *generating TCP traffic*. In the TSR scenario, we use it to synthesize traffic trace from counters. (2) *NetShare* [77]: This system must take packet traces as input, and uses the DoppelGANger model [50] with domain-specific insights to generate packet traces. It cannot take counter traces as input. (3) *Zoom2Net*: We extend Zoom2Net [37] for TSR. Following [37], this scheme uses a single Transformer model [70] to complete all tasks for TSR including up-scaling and packet header generation. In addition, we use the same loss function as ZOOMSYNTH to incorporate the domain knowledge in Zoom2Net. Comparison of ZOOMSYNTH with Zoom2Net can demonstrate the performance benefit of ZOOMSYNTH's other design choices. The hyperparameters of Zoom2Net are the same as GTT.

Metrics: We evaluate the fidelity of synthetic traffic traces by computing distance metrics between real and synthetic distributions of packet header fields. Following the common practice in prior work [74, 77], we utilize Earth Mover's Distance (EMD) (also called Wasserstein-1 distance) for continuous fields, such as packet length and timestamp, and Jensen-Shannon Divergence (JSD) for discrete fields, such as source address (SA), destination address (DA), source port (SP), destination port (DP), and protocol (PR).

7.1 Counter-to-Packet Synthesis

Accuracy. To assess the generation quality of timestamp field and packet length field, we employ EMD to compare the generated packet traces with the actual ones, with smaller values indicating closer resemblance. Fig. 8 presents the EMD metrics across four datasets. With only second-level counters, ZOOMSYNTH achieves performance comparable to NetShare (using packet-level traces as input), and significantly outperforms Zoom2Net and NetDiffusion. Specifically, ZOOMSYNTH achieves EMD reductions of 69.5% and 48.4% in the four datasets on average compared to Zoom2Net and NetDiffusion, respectively. In TON, CIDDS, and MAWI datasets, our method surpasses NetShare in terms of packet length's EMD. We further look at the percentiles of deviations for the 1st and 99th synthetic packet lengths generated by ZOOMSYNTH and NetShare in appendix §.4.2, and find that NetShare tries to fit the extreme values, making it perform well in generating packet lengths around the extreme values and also causing it to tend to overfit the variations of the packet length time-series, and thus leading to worse EMD.

For categorical fields such as IP address/port number/protocol in packet headers, we use JSD, a metric that assesses the distribution characteristics of discrete data within the whole set, and a smaller JSD is better. Similarly, we calculate the JSD between the synthetic data and the real data for each scheme, which is shown in Fig. 9. We can see ZOOMSYNTH achieves 49.6% and 35.6% JSD reductions compared to Zoom2Net and NetDiffusion, respectively, and it is slightly inferior to the NetShare scheme, which uses packet-level traces as input.

Error Accumulation. To show how errors accumulate in the multi-level TSR process of ZOOMSYNTH, we assess the results generated at multiple granularities. Counter-level results encompass three dimensions of information: time interval, byte count, and packet count. Since the time interval at the

Table 1: Performance comparison of ZOOMSYNTH, NetShare, NetDiffusion, and Zoom2Net on three downstream tasks.

Downstream Task	Anomaly Detection					Sketch-based Telemetry			Service Recognition				
	MLP	DT	RF	GB	LR	TON_SA	TON_DP	CAIDA_SP	MLP	DT	RF	GB	LR
Real Traces	0.99	1.0	1.0	1.0	0.8	-	-	-	1.0	1.0	1.0	1.0	0.87
ZOOMSYNTH	0.88	0.91	0.81	0.87	0.78	0.22	0.64	0.57	0.87	0.89	0.84	0.84	0.75
NetShare	0.58	0.78	0.64	0.77	0.60	0.14	0.62	0.53	0.83	0.79	0.85	0.78	0.60
NetDiffusion	0.43	0.49	0.43	0.51	0.36	0.29	0.78	0.73	0.60	0.59	0.63	0.55	0.49
Zoom2Net	0.31	0.42	0.32	0.35	0.32	0.41	0.94	0.82	0.58	0.54	0.61	0.49	0.43

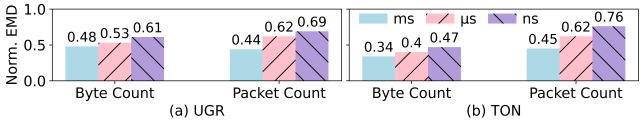


Figure 10: Error accumulation on counters of packets and bytes as the TSR processes.

counter level is fixed that represents the sampling period, we just calculate the EMD for the byte and packet counts. Due to how the EMD is calculated, there are magnitude differences in the EMD across different granularities. Therefore, when comparing the EMD values horizontally, it is necessary to perform normalization. Fig. 10 shows the normalized results. When generating microsecond counters from millisecond counters in the TON dataset, it is evident that even when the EMD values for millisecond data are divided by a thousand (reflecting the granularity difference), they are still significantly lower than the EMD for the microsecond-level results. A similar pattern is observed when nanosecond counters are generated from microsecond counters.

We conclude that, although errors accumulate in multi-level generation, the growth of accumulated error is slow, which has limited impact on overall accuracy, as we observe in Figs. 8 and 9.

7.2 Downstream Task Performance

We evaluate the performance of the three downstream tasks manufactured by ZOOMSYNTH in the following.

Anomaly Detection: We train DT, LR, RF, GB, and MLP with both real and synthetic traces and compare their accuracy with real traces. Table 1 shows that models trained using ZOOMSYNTH's synthetic traces achieve superior accuracy compared to NetShare, especially for DT model, its accuracy is 27.5% higher than that of NetShare, while they improve 93.2% and 149.2% on average accuracy compared to NetDiffusion and Zoom2Net, respectively.

Sketch-based Telemetry: We use E_{real} and E_{syn} to represent the number of errors in identifying heavy-hitters using CMS for real and synthetic data, respectively. If the features of the generated data are closer to those of the real data, these two values are also closer. Therefore, we use the relative error ratio $\frac{|E_{real}-E_{syn}|}{E_{real}}$ to evaluate the similarity between real data and synthetic data, and a lower value means better performance. We calculate the relative error of the heavy-hitters using TON and CAIDA datasets. Table 1 shows that the synthetic traces from ZOOMSYNTH have a performance similar to that of NetShare, with only 23.6% higher average relative error, and reduce the average relative errors with 21.3% and 49.6%, respectively, compared to NetDiffusion and Zoom2Net.

Table 2: The inference time of ZOOMSYNTH for achieving different end-to-end upscaling ratios from second-level counters. The upscaling factor k=10 and the results are obtained using TON dataset.

End-to-end Upscaling Ratio	10	10^2	10^3	10^5	10^9
Inference Time (Seconds)	0.855	0.938	0.951	0.965	0.966

Service Recognition: We compare the trained service recognition models using data from the real dataset and the synthetic ones of the compared schemes. Table 1 presents the results that the synthetic traces of ZOOMSYNTH better represent the characteristics of the real ones. For instance, the models trained with the data from ZOOMSYNTH achieve 9.8%, 46.9%, and 59.7% higher accuracy on average than the models trained with the synthetic data from NetShare, NetDiffusion, and Zoom2Net, respectively.

7.3 Real-time Synthesis

To assess whether ZOOMSYNTH can empower real-time downstream tasks, such as NDT [25], we measure the inference time under various end-to-end upscaling ratios. And to avoid measuring errors that result from inference with insufficient counter time series data, we employ counters of 1-second duration to generate packets and test whether the inference time exceeds 1 second. For example, when evaluating with an upscaling factor of 10^6, we use 1000× ms-level counters and record the end-to-end inference time. We conduct the experiments using CLTM-1.8B and 8× A100 GPUs. Table 2 shows that ZOOMSYNTH can achieve the target upscaling performance goal from second-level counters within 1 second for any end-to-end upscaling ratios, demonstrating that ZOOMSYNTH achieves real-time synthesis for various upscaling ratios.

7.4 Using CBF-based Counters

Due to limited memory and computing resources, many network devices adopt Counting Bloom Filters (CBF) [36] to implement packet or byte counters for obtaining approximate results. To see its impact on TSR performance, we implement a CBF-based counter with an array size of 300 and 2 hash functions, while using packet length as the hash key. Subsequently, we use the CBF-based counter to process the packet traces to obtain the counter time-series data, and then we use

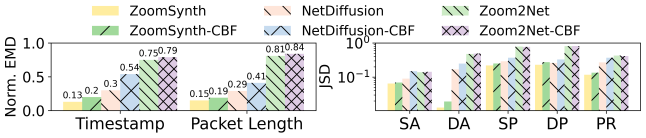


Figure 11: Normalized EMD and JSD between real and synthetic packet traces on UGR dataset using real counters and CBF counters as input, respectively.

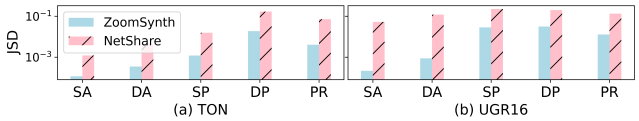


Figure 12: Performance comparison for packet header generation between NetShare and ZOOMSYNTH.

the data to train GTTs. Fig. 11 shows normalized EMD and JSD between real and synthetic packet traces on UGR dataset using real counters and CBF counters as input, respectively. Note that the packet and byte counters obtained using CBF on UGR dataset exceed the actual values by 7% and 12%, respectively. We can see ZOOMSYNTH outperforms NetDiffusion and Zoom2Net, improving EMD by 58.3% and 76.0%, and JSD by 52.3% and 80.1%, respectively. Besides, both ZOOMSYNTH and NetDiffusion have a significant increase in EMD for timestamp (53.8% for ZOOMSYNTH and 80.0% for NetDiffusion) and packet length (26.7% for ZOOMSYNTH and 41.4% for NetDiffusion), indicating that they are more sensitive to the accuracy of CBF.

7.5 Header Generation Model

To solely evaluate the quality of header generation, we use real nanosecond-level counter information as input, as this represents the packet prototype stripped of its header information. Fig. 12 provides an assessment of the capability of the header generation model. Our approach demonstrates an order of magnitude lower JSD values across various fields, proving that our header generation model significantly outperforms NetShare. This also underscores that the effectiveness of header generation is greatly dependent on the TSR results of the counters. We do not compare ZOOMSYNTH against NetDiffusion and Zoom2Net, since they do not have dedicated header generation modules.

7.6 Rule-following Model

We investigate the impact of counter rules on CLTM's header generation process. To do this, we construct multi-scale counter time-series and packet traces by filtering the original traffic traces using the ACL rules and use these data to train CLTM. Subsequently, we gather the packet outputs of ZOOMSYNTH with and without counter rules as input, respectively. Then we quantify the percentage of packets that adhere to the counter rules among all synthetic packets. For instance, when considering a rule like "Deny TCP", if counter rule is absent from the input, ZOOMSYNTH would use CLTM to recreate counters and use the header generation model to generate packet headers purely following the characteristics

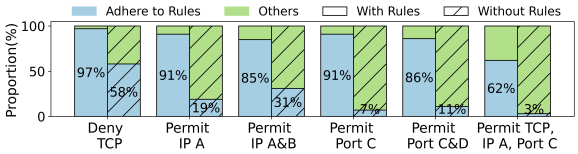


Figure 13: Comparison for the proportion of traffic which adheres to the counter rules with and without them as input.

Table 3: Summary of existing work.

Existing Work	Input	Output	Generality
DYNAMO [28], NS-2 [14]	Config Para.	Packets	✗
Swing [71], Tmix [73], LitGen [64]	Model Para.	Packets	✗
STAN [74]	-	Flows	✓
Redzovic et al. [61]	-	Sizes, Times	✓
DoppelGANger [50], NetShare [77]	Packets	Packets	✓
NetDiffusion [44, 46], PAC-GPT [47]	Prompts	Packets	✓
ZOOMSYNTH	Counters, Counter Rules	Packets, or Counters	✓

of the overall traffic flow.

We conduct this experiment on the CAIDA dataset, and use six counter rules as input to generate packet traces, respectively. Fig. 13 shows the comparison results for the proportion of traffic which adheres to the counter rules with and without counter rules as input. The results demonstrate that using counter rules as input can significantly benefit packet trace synthesis. For example, when the counter rule representing "Deny TCP" is not fed into the model, yet the counters represent traffic filtered by the rules, the proportion of TCP traffic in the final packet trace reaches 42%. In contrast, when we incorporate the "Deny TCP" rule into the packet header generation process, the final packet trace only has 3% TCP traffic. We can filter out the remaining 3% directly in the post-processing process without affecting the whole packet trace, because the proportion of the remaining TCP traffic among all traffic is small enough.

8 Related Work

There has been a large amount of work on traffic synthesis. In Table 3, we compare ZOOMSYNTH against them across three dimensions.

Traffic Synthesis Input: Existing work varies in their inputs. Simulation-/emulation-driven approaches demand detailed configurations, such as workloads and clients, to simulate network traffic in virtual environments. Model-driven approaches require distribution parameters derived from realistic traffic to establish statistical or structural models. ML-based models have diverse needs: some, like STAN [74] and Redzovic et al. [61], require no input, whereas others, such as DoppelGANger [50] and NetShare [77], need packet-level traffic data. In contrast, NetDiffusion [44, 46] and PAC-GPT [47] accept text prompts. Unlike these approaches, ZOOMSYNTH leverages counters to generate fine-grained traffic traces, achieving TSR not natively supported by them.

Traffic Synthesis Output: In addition to the inputs, the outputs of existing approaches also vary. Most approaches can produce packet-level traces, but some of them differ in

the fields generated from the packet headers. Specifically, simulation-/emulation-driven approaches acquire packets via direct packet capture within their environments. STAN [74] is limited to generating flow-level traffic, while Redzovic et al. [61] can only produce the header fields of packet sizes and interarrival times. Besides producing packet-level traces, ZOOMSYNTH further enables multi-scale synthesis of fine-grained counters directly from coarse-grained ones, offering flexibility for supporting various network applications.

Generality: In this regard, simulation-/emulation-driven and model-driven approaches are easy to analyze: they both rely on extracting specific features from realistic traffic to configure simulators, emulators, or models manually, making them inherently tied to specific workloads and thus limiting their adaptability for generalization. To overcome the issue, ZOOMSYNTH supports fast new scenario adaptation.

Cross-Domain Techniques: ZOOMSYNTH applies some techniques from other domains to traffic synthesis. ZOOMSYNTH adapts CLIP [59], a model used to align image and text embeddings in image synthesis, to understand the relationship between counter rules and packet traces. LoRA [40], originally used to fine-tune LLMs for new scenarios, is employed in training GTT layers of CLTM, tailored for the traffic synthesis domain. Moreover, ZOOMSYNTH chooses GPT-2 [11] as the header generation model, considering its generation efficiency.

9 Limitations and Future Work

Although ZOOMSYNTH boosts many network applications by enabling TSR, it faces several limitations that require further investigation. We discuss these limitations in the following.

Lowering Device Fingerprinting Risks: Sharing raw counter datasets from devices can increase the risks of device fingerprinting. Counters vary in type, with some accumulating values over time and others resetting at regular intervals. Public access to these counters could allow adversaries to identify devices according to their types. To reduce risk, operators can process counter data into a uniform format, thereby obscuring the original types.

Extending ZOOMSYNTH to Cases Needing Diverse Inputs: ZOOMSYNTH has limitations in accurately synthesizing traffic for cases that require more than just counters as input. For example, generating traffic for video or audio applications, using only counters may not ensure that the traffic aligns with the specific characteristics of these applications. Future research could explore methods to synthesize traces that capture the characteristics of specific applications, incorporating more inputs such as application descriptions along with counters.

Mitigating Adversarial Misuse Risks: There is a potential for adversaries to exploit ZOOMSYNTH for malicious purposes, such as using synthesized normal traffic patterns to evade detection by security devices like IDS and IPS. Although ZOOMSYNTH focuses on TSR, the potential for misuse presents a significant challenge. This highlights the need for future research on preventing adversarial use.

Reducing Adversarial Effects in Traffic Synthesis: ML-based traffic synthesizers are vulnerable to adversarial traffic [29], which can lead to inaccurate output. Adversaries may tamper with training data, introducing deceptive patterns that result in unrealistic traces. To protect against this, the training datasets for GTTs should be carefully filtered to exclude adversarial samples. Identifying adversarial traffic is a well-established research area [39, 69] and is beyond the scope of our work.

Enhancing Support for More Downstream Tasks: ZOOMSYNTH currently has limited support for downstream tasks requiring precise stateful protocol behaviors, such as congestion control and switch buffer monitoring, in line with existing work [45, 47, 50, 74, 77]. Future work will focus on enhancing packet generation with advanced stateful features (*e.g.* sequence numbers) and improving inter-arrival sequence modeling through better protocol and application behavior understanding.

Enhancing Generalization in Header Generation: ZOOMSYNTH has generalization issue in header generation, as it limits header field values to those found in the training data. This reduces diversity compared to realistic traffic. To enhance generality while maintaining semantic conformity, future research could involve training a packet header model with domain-specific semantic knowledge.

10 Conclusion

We present ZOOMSYNTH, the first TSR system to restore fine-grained traffic traces from coarse-grained counters. We design GTTs to capture traffic characteristics at particular granularities, and compose CLTM as a combination of a tree of GTTs and a rule-following model. ZOOMSYNTH uses CLTM and a header generation model to achieve packet-trace synthesis, and can be adapted to new scenarios with LoRA. We prototype ZOOMSYNTH with PyTorch and evaluate it on a real testbed using public datasets. ZOOMSYNTH can satisfy the requirements for three downstream tasks. Compared to NetShare that takes packet-level trace as input, ZOOMSYNTH can achieve comparable accuracy in terms of JSD and EMD, while obtaining more than 42.1% accuracy improvements compared to NetDiffusion.

Acknowledgments

We thank the anonymous NSDI reviewers for their constructive comments. Libin Liu and Li Chen are the corresponding authors of this paper. This work was supported by Zhongguancun Laboratory and the Beijing Outstanding Young Scientist Program (No. JWZQ20240101008), the National Natural Science Foundation of China under Grant U23B2001.

References

[1] Implementing SNMP on the Cisco IOS XR Software. https://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r3-9/system_management/configuration/guide/yc39xr12k_chapter10.html, 2011.

[2] Configuring NetFlow on Cisco IOS XR Software. https://www.cisco.com/c/dam/en/us/td/docs/iosxr/ncs5xx/netflow/63x/b-netflow-cg-63x-ncs5xx.html, 2021.

[3] NetFlow Configuration Guide for Cisco IOS XE. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/netflow/configuration/xe-3s/nf-xe-3s-book/cfg-nflow-data-expt-xe.html, 2022.

[4] SNMP Configuration Guide for Cisco IOS XE. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/snmp/configuration/xe-16-6/snmp-xe-16-6-book/nm-snmp-cfg-snmp-support.html, 2022.

[5] All Router Products of Cisco. https://www.cisco.com/c/en/us/support/routers/index.html, 2024.

[6] Capture files from Mid-Atlantic CCDC. https://www.netresec.com/?page=MACCDC, 2024.

[7] CIDDS - Coburg Intrusion Detection Data Sets. https://www.hs-coburg.de/forschung/forschungsprojekte-oeffentlich/informationstechnologie/cidds-coburg-intrusion-detection-data-sets.html, 2024.

[8] Cisco 8000 Hardware Emulator Datasheet. https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000-emulator/cisco8000-hardware-emulator-datasheet.html, 2024.

[9] Data Set for IMC 2010 Data Center Measurement. https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2024.

[10] DPDK. https://www.dpdk.org/, 2024.

[11] Hugging Face gpt2. https://github.com/AUTOMATIC1111/stable-diffusion-webui, 2024.

[12] Linux Network Interface Statistics. https://www.kernel.org/doc/html/latest/networking/statistics.html, 2024.

[13] NetDiffusion Generator. https://github.com/noise-lab/NetDiffusion_Generator, 2024.

[14] ns-2. https://nsnam.sourceforge.net/wiki/index.php/Main_Page, 2024.

[15] Packet Framework Library of DPDK. https://doc.dpdk.org/guides/prog_guide/packet_framework.html, 2024.

[16] PyTorch Library. https://pytorch.org/, 2024.

[17] Router Interface Counters in SONiC. https://sonic.software/sai/group__SAIROUTERINTF.html, 2024.

[18] rte_table_action.h. https://doc.dpdk.org/api/rte__table__action_8h_source.html, 2024.

[19] SAI - Router Interface Specific API Definitions. https://sonic.software/sai/struct__sai__router__interface__api__t.html, 2024.

[20] SONiC. https://sonicfoundation.dev/, 2024.

[21] Stable Diffusion Web UI. https://github.com/AUTOMATIC1111/stable-diffusion-webui, 2024.

[22] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/catalog/datasets/passive_dataset/, 2024.

[23] Cisco 500 Series WPAN Industrial Routers. https://www.cisco.com/c/en/us/support/routers/500-series-wpan-industrial-routers/series.html, 2025.

[24] NCS 540 Series Routers. https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5xx/interfaces/24xx/b-interfaces-hardware-component-cg-24xx-ncs540/config-ethernet-interfaces.html, 2025.

[25] Paul Almasan, Miquel Ferriol-Galmés, Jordi Paillisse, José Suárez-Varela, Diego Perino, Diego López, Antonio Agustin Pastor Perales, Paul Harvey, Laurent Ciavaglia, Leon Wong, et al. Digital Twin Network: Opportunities and Challenges. *arXiv preprint arXiv:2201.01144*, 2022.

[26] Claise Benoit, Sadasivan Ganesh, Valluri Vamsi, and Djernaes Martin. Cisco systems netflow services export version 9. *RFC 3954*, 2004.

[27] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Hyojoon Kim, Renata Teixeira, and Nick Feamster. Traffic Refinery: Cost-aware Data Representation for Machine Learning on Network Traffic. *Proc. ACM on Measurement and Analysis of Computing Systems*, 5(3):1–24, 2021.

[28] Tobias Bühler, Roland Schmid, Sandro Lutz, and Laurent Vanbever. Generating Representative, Live Network Traffic out of Millions of Code Repositories. In *Proc. ACM HotNets*, 2022.

[29] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proc. IEEE SP*, 2017.

[30] Yuxuan Chen, Rongpeng Li, Zhifeng Zhao, Chenghui Peng, Jianjun Wu, Ekram Hossain, and Honggang Zhang. Netgpt: A native-ai network architecture beyond provisioning personalized generative services. *arXiv preprint arXiv:2307.06148*, 2023.

[31] Kenjiro Cho. MAWI Working Group Traffic Archive. *https://mawi.wide.ad.jp/mawi/*, 2023.

[32] W Dai, J Li, D Li, AMH Tiong, J Zhao, W Wang, B Li, P Fung, and S Hoi. InstructBLIP: Towards General-purpose Vision-language Models with Instruction Tuning. In *Proc. NeurIPS*, 2023.

[33] Daizong Ding, Mi Zhang, Xudong Pan, Min Yang, and Xiangnan He. Modeling Extreme Events in Time Series Prediction. In *Proc. ACM SIGKDD*, 2019.

[34] Franck Djeumou, Cyrus Neary, Eric Goubault, Sylvie Putot, and Ufuk Topcu. Neural Networks with Physics-Informed Architectures and Constraints for Dynamical Systems Modeling. In *Proc. Machine Learning Research*, 2022.

[35] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image Super-Resolution using Deep Convolutional Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307, 2015.

[36] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[37] Fengchen Gong, Divya Raghunathan, Aarti Gupta, and Maria Apostolaki. Zoom2Net: Constrained Network Telemetry Imputation. In *Proc. ACM SIGCOMM*, 2024.

[38] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *Proc. IEEE ICASSP*, 2013.

[39] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New Directions in Automated Traffic Analysis. In *Proc. ACM CCS*, 2021.

[40] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*, 2021.

[41] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. 2023.

[42] Shuodi Hui, Huandong Wang, Zhenhua Wang, Xinghao Yang, Zhongjin Liu, Depeng Jin, and Yong Li. Knowledge Enhanced GAN for IoT Traffic Generation. In *Proc. ACM WWW*, 2022.

[43] Case Jeffrey, Mundy Russ, Partain David, and Stewart Bob. Introduction and applicability statements for internet-standard management framework. *RFC 3410*, 2002.

[44] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. NetDiffusion: Network Data Augmentation Through Protocol-Constrained Traffic Generation. *arXiv preprint arXiv:2310.08543*, 2023.

[45] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. NetDiffusion: Network Data Augmentation through Protocol-Constrained Traffic Generation. *Proc. of the ACM on Measurement and Analysis of Computing Systems*, 8(1):1–32, 2024.

[46] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Generative, High-Fidelity Network Traces. In *Proc. ACM HotNets*, 2023.

[47] Danial Khosh Kholgh and Panos Kostakos. PAC-GPT: A Novel Approach to Generating Synthetic Network Traffic with GPT-3. *IEEE Access*, 2023.

[48] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *Proc. USENIX NSDI*, 2018.

[49] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-Realistic Single Image Super-Resolution using a Generative Adversarial Network. In *Proc. IEEE CVPR*, 2017.

[50] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. Using GANs for Sharing Networked Time Series Data: Challenges, Initial Promise, and Open Questions. In *Proc. ACM IMC*, 2020.

[51] Zinan Lin, Hao Liang, Giulia Fanti, and Vyas Sekar. RareGAN: Generating Samples for Rare Classes. In *Proc. AAAI*, 2022.

[52] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual Instruction Tuning. In *Proc. NeurIPS*, 2023.

[53] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-based Monitoring in Software Switches. In *Proc. ACM SIGCOMM*, 2019.

[54] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with Univmon. In *Proc. ACM SIGCOMM*, 2016.

[55] Gabriel Maciá-Fernández, José Camacho, Roberto Magán-Carrión, Pedro García-Teodoro, and Roberto Therón. UGR'16: A New Dataset for the Evaluation of Cyclostationarity-based network IDSs. *Computers & Security*, 73:411–424, 2018.

[56] ML Menéndez, JA Pardo, L Pardo, and MC Pardo. The jensen-shannon divergence. *Journal of the Franklin Institute*, 334(2):307–318, 1997.

[57] Nour Moustafa. A New Distributed Architecture for Evaluating AI-based Security Systems at the Edge: Network TON_IoT Datasets. *Sustainable Cities and Society*, 72:102994, 2021.

[58] Jian Qu, Xiaobo Ma, and Jianfeng Li. Trafficgpt: Breaking the token barrier for efficient long traffic analysis and generation. *arXiv preprint arXiv:2403.05822*, 2024.

[59] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning Transferable Visual Models from Natural Language Supervision. In *Proc. PMLR ICML*, 2021.

[60] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[61] Hasan Redžović, Aleksandra Smiljanić, and Milan Bjelica. IP Traffic Generator Based on Hidden Markov Models. *parameters*, 1(2):1.

[62] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. Flow-based Network Traffic Generation using Generative Adversarial Networks. *Computers & Security*, 82:156–172, 2019.

[63] Markus Ring, Sarah Wunderlich, Dominik Grüdl, Dieter Landes, and Andreas Hotho. Flow-based Benchmark Data Sets for Intrusion Detection. In *Proc. ECCWS*, 2017.

[64] Chloé Rolland, Julien Ridoux, and Bruno Baynat. LiTGen, a Lightweight Traffic Generator: Application to P2P and Mail Wireless Traffic. In *Proc. International Conference on Passive and Active Network Measurement*, pages 52–62, 2007.

[65] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution Image Synthesis with Latent Diffusion Models. In *Proc. IEEE CVPR*, 2022.

[66] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40:99–121, 2000.

[67] Amin Shabani, Amir Abdi, Lili Meng, and Tristan Sylvain. Scaleformer: Iterative Multi-scale Refining Transformers for Time Series Forecasting. In *Proc. ICLR*, 2023.

[68] Binh Tang and David S Matteson. Probabilistic Transformer for Time Series Analysis. In *Proc. NeurIPS*, 2021.

[69] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. BotFinder: Finding Bots in Network Traffic without Deep Packet Inspection. In *Proc. ACM CoNEXT*, 2012.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Proc. NeurIPS*, 2017.

[71] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and Responsive Network Traffic Generation. In *Proc. ACM SIGCOMM*, 2006.

[72] Yuejie Wang, Qiutong Men, Yao Xiao, Yongting Chen, and Guyue Liu. ConfMask: Enabling Privacy-Preserving Configuration Sharing via Anonymization. In *Proc. ACM SIGCOMM*, 2024.

[73] Michele C Weigle, Prashanth Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F Donelson Smith. Tmix: A Tool for Generating Realistic TCP Application Workloads in NS-2. *ACM SIGCOMM Computer Communication Review*, 36(3):65–76, 2006.

[74] Shengzhe Xu, Manish Marwah, Martin Arlitt, and Naren Ramakrishnan. STAN: Synthetic Network Traffic Generation with Generative Neural Models. In *Proc. International Workshop on Deployable Machine Learning for Security Defense*, 2021.

[75] Kun Yang, Samory Kpotufe, and Nick Feamster. A Comparative Study of Network Traffic Representations for Novelty Detection. *arXiv preprint arXiv:2006.16993*, 2020.

[76] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proc. ACM SIGCOMM*, 2020.

[77] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical GAN-based Synthetic IP Header Trace Generation using NetShare. In *Proc. ACM SIGCOMM*, 2022.

[78] Zongsheng Yue, Jianyi Wang, and Chen Change Loy. ResShift: Efficient Diffusion Model for Image Super-Resolution by Residual Shifting. In *Proc. NeurIPS*, 2023.

[79] Yulun Zhang, Kunpeng Li, Kai Li, Lichen Wang, Bineng Zhong, and Yun Fu. Image Super-Resolution using Very Deep Residual Channel Attention Networks. In *Proc. ECCV*, 2018.

Appendix

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

.1 Additional Details of CLTM

.1.1 Training Data Preparation for GTT Models

We need traffic data with various resolutions to train GTT $^{k}_{\delta}$ models. To obtain them, we aggregate raw pcap data from public datasets into counter time-series at various granularities, according to k. We have introduced all the public datasets we use in §6. The number of needed training data couple is $\frac{\theta}{k}$, where the overall end-to-end upscaling raito for CLTM is θ. Taking packet synthesis from second-level counters (overall end-to-end upscaling ratio is 10^9) and k=10 as an example, a total nine couples of counter time-series at different granularities are needed, including (1s, 100ms), (100ms, 10ms), ..., and (10ns, 1ns). These counter time-series is structured as `pandas.DataFrame` with the following columns: time, packet count, and byte count. Moreover, CLTM increases GTTs within each stage by k to process different partitions of counter time-series to improve scalability, as the data size increases by k along with the upscaling stages. Considering the temporal dependency between the neighbouring partitions, we intentionally introduce overlap between them, which enhances the ability of GTTs to capture long-term time-series characteristics.

.1.2 Choosing Upscaling Factor k

As analyzed in §2.4, each GTT concentrates on a particular upscaling stage with a fixed upscaling factor k, which is important for the tradeoff between end-to-end traffic synthesis accuracy and model training cost. A GTT with small k can achieve satisfactory end-to-end synthesis accuracy, yet may incur extra training costs as involving more GTTs for more upscaling steps.

The choices of k are different across different datasets. Generally, a GTT having smaller upscaling factor can have better TSR performance. For example, GTT $^{10}_{1ms}$ outperforms GTT $^{100}_{1ms}$. For a fixed end-to-end upscaling ratio, however, smaller k would increase CLTM stages. Each stage may have a degree of synthesis errors, which may accumulate along with upscaling stages. In addition, the performance gains obtained by choosing GTT $^{10}_{1ms}$ over GTT $^{100}_{1ms}$ are not consistent across the datasets, since they exhibit various characteristics at different granularities. Therefore, we employ experimental methods to determine the value of k.

We use TON [57] and CIDDS [7] datasets as an example to show how we experimentally choose the value of k. We restore packet trace from second-level counters and make k equal to 5, 10, 10^2, and 10^3, respectively. These choices of k are enough for us to determine the k values for the two datasets. For each k value, we calculate the normalized EMD and JSD between real and synthetic packet traces. Fig. 14 shows the results. We can see, for the two datasets, the normalized EMD and JSD results have different trends along with the increase of k. For TON dataset, the normalized EMD and JSD results increase along with k, and they are similar for k=5 and k=10, and increase significantly when k>10. However, for CIDDS dataset, from k=5 to k=10^2, both EMD and JSD results of the most packet header fields decrease along with k, and then they increase largely when k=10^3, compared to k=10^2.

Table 4 shows the corresponding end-to-end model training time for different values of k. The measurements are done on the testbed described in §7. We only show the training time using TON dataset, since it is consistent for all the datasets that more CLTM stages with more GTT models would lead to longer training time. Taking accuracy and training time into account, we recommend to set k as 10 and 10^2 for the TON and CIDDS datasets, respectively. Other datasets adopt the same method to choose their k.

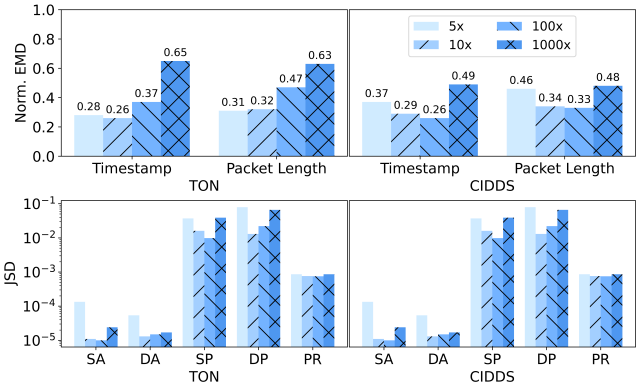


Figure 14: Normalized EMD and JSD between real and synthetic packet traces for different choices of k on TON and CIDDS datasets (SA: source address, DA: destination address, SP: source port, DP: destination port, PR: protocol).

Table 4: The end-to-end training time of ZOOMSYNTH for different values of k using TON dataset.

k	5	10	10^2	10^3
Training Time (GPU Hours)	99.15	61.91	6.77	6.31

.1.3 Training CLTM

In the training phase of CLTM, we combine the output vector of pre-trained rule-following model and the vector of counter time-series at a particular granularity and feed their combinations into GTT to train it. Specifically, when no counter rules are available, its input to GTT is a zero vector. To combine them, we directly copy and extend the output vector of pre-trained rule-following model into 2 dimensions, ensuring that the two vectors are of same shape before concatenating them.

.1.4 Handling Sparsity with Early Stopping

We observe some intermediate fine-grained counter time-series exhibit sparsity. For instance, when upscaling a counter

Table 5: Hyperparameters of ZOOMSYNTH models.

Models	Param.	Value
GTT	Transformer Layers	15
	Multi-Headers	4
	Hidden Size	256
	Dropout	0.1
	Learn Ratio	0.001 - 0.01 (vary with iterations)
	Batch Size	10 - 64 (vary with data volumes)
	λ in Loss Func.	0.1 - 1.5 (vary with iterations)
	σ in Loss Func.	0 - 1 (vary with iterations)
Header Gen. Model	Vocab Size	50000
	Embedding Dim.	128
	Generation Model	GPT2-Small
Rule-following Model	Text Encoder	Transformer-63M
	Similarity Matrix	256*256
Downstream Task Adaptor	Header Generation	Transformer-38M
	Task-specific Model	3-layers DT or MLP

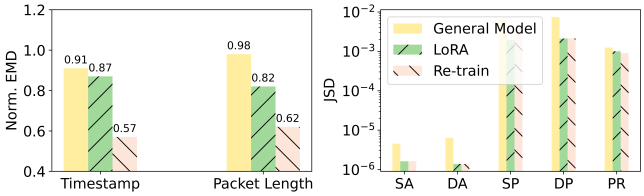


Figure 15: EMD and JSD between real and synthetic distributions on CIDDS using four kind of models: specific model trained on CIDDS, general model trained on combined dataset, LoRA on general model, re-training general model.

time-series from 10ms to 1ms, there are 8 upscaled 1ms-counters whose values are 0; during the inference phase, for these counters, ZOOMSYNTH will stop the upscaling process of the subsequent GTT models in CLTM. In addition, if both the packet count of the other 2 1ms-counters are 1, this represents the 1ms-counter has reached the finest granularity and its byte count is the packet size, we can stop its upscaling process as well. By classifying the sparsity of counter time-series during the upscaling process, ZOOMSYNTH accelerates the inference of CLTM with an early stop mechanism, avoiding useless upscalings.

.1.5 *Fast New Scenario Adaptation*

CLTM is designed to be a general foundation model for TSR and we train it using the common traffic traces, which prevents it from overfitting to the specific features of any specific dataset, while directing the model to focus on the underlying and universal traffic features across various scenarios. However, we also admit that traffic characteristics can vary significantly for different scenarios, and can also change over time. Therefore, we design an approach for adapting CLTM to the new scenarios with low overhead.

We leverage the LoRA-based approach to empower CLTM to adapt to new scenarios quickly. This is a systematic application of LoRA [40] for traffic synthesis building on its success in fine-tuning of LLMs. The core of LoRA is training the low-rank parameter matrices of the whole model with a small amount of data from new scenarios during the re-training phase. After finishing the training, it injects the re-trained parameters to the original model and replaces the corresponding ones. We use the same approach to train the low-rank parameter matrices of CLTM.

We only choose the GTT model layer of CLTM for finer-grained counter synthesis to perform re-training with LoRA approach, which further accelerates the adaptation process. We observe that across various scenarios, coarse-grained traffic statistics remain consistent, the features of the corresponding fine-grained traffic are different. Therefore, as shown in Fig. 1, for GTTs within the upper layers of CLTM, they are similar to perform uniform upscalings, while GTTs within the lower layer vary in capturing the traffic fine-grained features. Based on the observation and analysis, we should re-train GTTs for finer-grained counter synthesis. For instance, we choose GTT $^{10}_{10ns}$ and GTT $^{10}_{1ns}$ in the last two stages of CLTM to be retrained for packet synthesis from second-level counter time-series.

.2 Training Header Generation Model

For preparing the training dataset, we process the raw packet trace to the same granularity of counter time series. We use the processed counter time series and packet trace to train the header generation model. The training method is that we mask header fields randomly and use the model to regenerate the relative fields. We retrain the header generation model on our hybrid dataset, with the training process taking approximately 8 days.

.3 Implementation Details of ZOOMSYNTH

.3.1 *Hyperparameters of* ZOOMSYNTH *Models*

Table 5 lists the hyper-parameter setups of ZOOMSYNTH for each model.

.3.2 *Operation APIs of CLTM*

The six APIs to support operations of CLTM introduced in §6 include `def_TSR_ratio(·)` for defining the end-to-end upscaling ratio, `def_GTT_factor(·)` for choosing the factor k of the GTT, `def_GTT_num(·)` for deciding the number of GTTs in CLTM, `def_GTT_model(·)` for configuring model hyperparameters, `def_training_iter(·)` for setting model training parameters.

.4 Additional Experimental Results

.4.1 *Adapting to New Scenarios*

To simulate adapting to a new scenario, we design two datasets: the D_0 dataset combines UGR16, TON and MAWI datasets; and the D_+ dataset which contains only the CIDDS dataset whose size is 5% of that of D_0. We then train three models: 1) the *General Model* is a CLTM trained on D_0; 2) with the *General Model* as base, we train the LoRA layers on D_+, which is our recommended procedure to adapt CLTM to new scenarios; we denote the combined model of *General Model* and LoRA layers as *LoRA*; 3) Using the *General Model* as a starting point, we run 50 iterations of training using D_+, and we call the resulting model *Re-train Model*.

Fig. 15 shows the results. We observe that the *Re-train Model* achieves the best performance, and *LoRA* can greatly

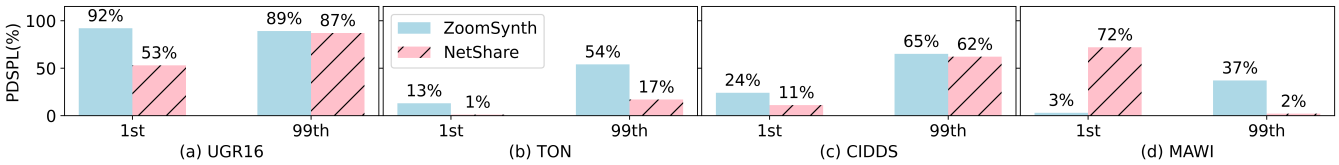


Figure 16: Proportions of deviations for synthetic packet lengths at the 1st and 99th percentile generated by ZOOMSYNTH and NetShare on the UGR16, TON, CIDDS, and MAWI datasets (PDSPL: Proportions of Deviations for the Synthetic Packet Lengths).

improve the *General Model*'s performance in the new scenario. *LoRA* can improve the generation quality of timestamps and packet lengths by 4.3% and 15%, respectively. For header generation, *LoRA* achieves roughly the same performance as the *Re-train Model*. Since re-training the complete CLTM model takes up significantly more resource and time than training a few LoRA layers, we believe adapting to new scenarios with *LoRA* is an attractive alternative for users with limited time, data, and resources.

.4.2 Deviation Proportions of Synthetic Packet Lengths

To further explore the generation performance of ZOOMSYNTH and NetShare in synthesizing packet lengths, we look at the proportions of deviation between the synthetic and realistic packet lengths. This metric, referred to as the proportion of deviations for synthetic packet lengths (PDSPL), is defined as follows:

$$PDSPL_i = \frac{|PL_i^{syn} - PL_i^{real}|}{PL_i^{real}} \times 100\%.$$

Here, PL_i^{real} denotes the packet length at the ith percentile of the sorted realistic packet length time-series, arranged from smallest to largest, while PL_i^{syn} represents the corresponding value of the sorted synthetic packet length time-series. This metric measures how closely the synthetic packet lengths match the realistic ones across different percentiles.

Fig. 16 shows the proportions of deviations for the 1st and 99th percentile synthetic packet lengths generated by ZOOMSYNTH and NetShare on the UGR16, TON, CIDDS, and MAWI datasets. We can see NetShare exhibits lower deviations at both the 1st and 99th percentiles, except for the 1st percentile packet length on the MAWI dataset. We analyze that this is because GAN-based models are sensitive to fluctuations and extreme values in time-series data [46], making NetShare perform better in generating packet lengths around extreme values. However, this sensitivity also causes NetShare to tend to overfit the variations of the packet length time-series, leading to worse EMD performance compared to ZOOMSYNTH, as we observe in Fig. 8.

.5 Cisco Router Products Supporting Counters

Table 6 shows Cisco's router products which support counters, their configuration commands to display counters, and the corresponding online configuration manuals. These routers are the 41 out of 42 ones from 500 series to 12000 series listed at Cisco's official site [5]. Their configuration manuals state that they support counters, which can be obtained via command line interface (CLI) with the corresponding configuration commands. The router product which does not appear in Table 6 is 8000 series virtual router emulator, whose data sheet introduces that some hardware counters are not supported by it [8]. It is not that it does not support counters at all.

We list the links of the online configuration manuals referenced in Table 6 below.

[1] https://www.cisco.com/c/en/us/support/routers/500-series-wpan-industrial-routers/series.html

[2] https://www.cisco.com/c/en/us/td/docs/routers/ncs5xx/ncs520/configuration/guide/CE/17-1-1/b-ce-xe-17-1-1-ncs520/using-ethernet-operations-administration-and-maintenance.html

[3] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5xx/interfaces/24xx/b-interfaces-hardware-component-cg-24xx-ncs540/config-ethernet-interfaces.html

[4] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs560/interfaces/24xx/b-interfaces-hardware-component-cg-24xx-ncs560/configuring_ethernet_interfaces.html

[5] https://www.cisco.com/c/en/us/td/docs/routers/access/800/829/software/configuration/guide/b_IR800config/b_cellular.html

[6] https://www.cisco.com/c/en/us/td/docs/routers/access/800M/software/800MSCG/routconf.html

[7] https://www.cisco.com/c/en/us/td/docs/routers/access/900/software/configuration/guide/900SCG/routconf.html

[8] https://www.cisco.com/c/en/us/td/docs/routers/asr903/software/guide/chassis/17-1-1/b-config-guide-xe-17-1-1-asr900/configuring_ethernet_interfaces.html

[9] https://www.cisco.com/c/en/us/td/docs/wireless/asr_901/Configuration/Guide/b_asr901-scg/b_asr901-scg_chapter_01010.html

Table 6: Cisco router products which support counters, their configuration commands to display counters, and the corresponding online manuals.

No.	Router Products	Config Commands	Config Manuals
1	500 Series WPAN Industrial Routers	show wpan packet-count	[1]
2	Network Convergence System 500 Series Routers	show ethernet oam statistics	[2]
3	Network Convergence System 540 Series Routers	show interface [name]	[3]
4	Network Convergence System 560 Series Routers		[4]
5	800 Series Industrial Integrated Services Routers		[5]
6	800 Series Routers		[6]
7	900 Series Integrated Services Routers		[7]
8	ASR 900 Series Aggregation Services Routers		[8]
9	ASR 901 10G Series Aggregation Services Routers		[9]
10	ASR 901 Series Aggregation Services Routers		
11	ASR 920 Series Aggregation Services Router		[10]
12	1000 Series Connected Grid Routers		[11]
13	Cisco 1000 Series Integrated Services Routers		[12]
14	ASR 1000 Series Aggregation Services Routers		[13]
15	Cloud Services Router 1000V Series	show platform packet-trace statistics	[14]
16	Catalyst IR1100 Rugged Series Routers	show interface [name]	[15]
17	Catalyst IR1800 Rugged Series Routers		[16]
18	1900 Series Integrated Services Routers		[17]
19	2000 Series Connected Grid Routers		[18]
20	2900 Series Integrated Services Routers		[17]
21	3900 Series Integrated Services Routers		
22	3000 Series Industrial Compute Gateways	show iox detail	[19]
23	4000 Series Integrated Services Routers	show interface [name]	[20]
24	5000 Series Enterprise Network Compute System	show ip traffic interface [name]	[21]
25	Network Convergence System 5000 Series	show interface [name]	[22]
26	Network Convergence System 5500 Series		[23]
27	Network Convergence System 5700 Series		[24]
28	5900 Series Embedded Services Routers		[25]
29	Network Convergence System 6000 Series Routers		[26]
30	ESR6300 Embedded Series Routers		[27]
31	8000 Series Routers	sh int [name]	[28]
32	Catalyst 8000V Edge Software	show platform packet-trace statistics	[29]
33	Cisco Catalyst 8200 Series Edge Platforms	show interface [name]	[30]
34	Cisco Catalyst 8300 Series Edge Platforms		
35	Cisco Catalyst 8200 Series Edge uCPE		[31]
36	Cisco Catalyst 8300 Series Edge uCPE		
37	Catalyst 8500 Series Edge Platforms	show platform packet-trace statistics	[32]
38	Catalyst 8500L Series Edge Platforms		
39	ASR 9000 Series Aggregation Services Routers	show interface [name]	[33]
40	IOS XRv 9000 Router	show controller dpa statistics global	[34]
41	XR 12000 Series Router	show interface [name]	[35]

[10] https://www.cisco.com/c/en/us/td/docs/routers/asr920/configuration/guide/chassis/17-1-1/b-Chassis-Guide-xe-17-1-asr920/m-eth_im_config-asr920.html

[11] https://www.cisco.com/c/en/us/td/docs/routers/connectedgrid/cgr1000/1_0/software/configuration/guide/wifi/WiFi_Book/wifi_interface.html

[12] https://www.cisco.com/c/en/us/td/docs/routers/access/isr1100/software/configuration/xe-17/isr1100-sw-config-xe-17/troubleshooting.html

[13] https://www.cisco.com/c/en/us/td/docs/routers/asr1000/configuration/guide/chassis/asr1000-software-config-guide/bdi-asr.html

[14] https://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/configuration/b_CSR1000v_Configuration_Guide/packet_trace.html

[15] https://www.cisco.com/c/en/us/td/docs/

routers/access/1101/software/configuration/guide/b-cisco-ir1101-scg/m-basic-router-cli-configuration.html

[16] https://www.cisco.com/c/en/us/td/docs/routers/access/IR1800/software/b-cisco-ir1800-scg/m-basic-router-cli-config.html

[17] https://www.cisco.com/c/en/us/td/docs/routers/access/1900/software/configuration/guide/Software_Configuration/routconf.html

[18] https://www.cisco.com/c/en/us/td/docs/routers/access/2000/CGR2010/software/configuration/guide/CGR_2010/routconf.html

[19] https://www.cisco.com/c/en/us/td/docs/routers/ic3000/deployment/guide/b_IC3000_deployment_guide/b_IC3000_deployment_guide_chapter_0100.html

[20] https://www.cisco.com/c/en/us/td/docs/routers/access/4400/software/configuration/xe-16-6/isr4400swcfg-xe-16-6-book/basicconfg.html

[21] https://www.cisco.com/c/en/us/td/docs/routers/sdwan/configuration/system-interface/ios-xe-17/systems-interfaces-book-xe-sdwan/configure-interfaces.html

[22] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5000/interfaces/711x/configuration/guide/b-interfaces-hardware-component-cg-ncs5000-711x/advanced-configuration-modification-of-management-ethernet-interface.html

[23] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5500/interfaces/24xx/configuration/guide/b-interfaces-hardware-component-cg-ncs5500-24xx/configuring-ethernet-interfaces.html

[24] https://www.cisco.com/c/en/us/products/collateral/routers/network-convergence-system-5500-series/network-convergence-sys-wp.html

[25] https://www.cisco.com/c/en/us/td/docs/solutions/GGSG-Engineering/15-4-3M/config-guide/Configuration-Guide/Config.html

[26] https://www.cisco.com/c/en/us/td/docs/routers/ncs6000/software/ncs6k-7-6/b-interfaces-hardware-component-cg-ncs6000-76x/configuring-ethernet-interfaces.html

[27] https://www.cisco.com/c/en/us/td/docs/routers/embedded/6300/software/config/b_ESR6300_config/b_ESR6300_config_chapter_010.html

[28] https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000/Interfaces/24xx/configuration/guide/b-interfaces-config-guide-cisco8k-r24xx/m-configuring-ethernet-interfaces-8k.html

[29] https://www.cisco.com/c/en/us/td/docs/routers/C8000V/Configuration/c8000v-edge-software-configuration-guide/packet_trace.html

[30] https://www.cisco.com/c/en/us/td/docs/routers/cloud_edge/c8300/software_config/cat8300swcfg-xe-17-book/isr9000swcfg-xe-16-12-book_chapter_010.html

[31] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/interface/command/ir-cr-book/ir-s4.html

[32] https://www.cisco.com/c/en/us/td/docs/routers/cloud_edge/c8500/software-configuration-guide/c8500-software-config-guide/packet_trace.html

[33] https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/24xx/interfaces/configuration/guide/b-interfaces-hardware-component-cg-asr9000-24xx/configuring-ethernet-interfaces.html

[34] https://www.cisco.com/c/en/us/td/docs/routers/virtual-routers/configuration/guide/b-xrv9k-cg/m-cisco-ios-xrv-9000-data-plane-specific-features.html

[35] https://www.cisco.com/c/en/us/products/collateral/routers/xr-12000-series-router/product_data_sheet0900aecd803f856f.html

BFTBrain: Adaptive BFT Consensus with Reinforcement Learning

Chenyuan Wu
University of Pennsylvania

Haoyun Qin
University of Pennsylvania

Mohammad Javad Amiri
Stony Brook University

Boon Thau Loo
University of Pennsylvania

Dahlia Malkhi
UC Santa Barbara

Ryan Marcus
University of Pennsylvania

Abstract

This paper presents BFTBrain, a reinforcement learning (RL) based Byzantine fault-tolerant (BFT) system that provides significant operational benefits: a plug-and-play system suitable for a broad set of hardware and network configurations, and adjusts effectively in real-time to changing fault scenarios and workloads. BFTBrain adapts to system conditions and application needs by switching between a set of BFT protocols in real-time. Two main advances contribute to BFTBrain's agility and performance. First, BFTBrain is based on a systematic, thorough modeling of metrics that correlate the performance of the studied BFT protocols with varying fault scenarios and workloads. These metrics are fed as features to BFTBrain's RL engine in order to choose the best-performing BFT protocols in real-time. Second, BFTBrain coordinates RL in a decentralized manner which is resilient to adversarial data pollution, where nodes share local metering values and reach the same learning output by consensus. As a result, in addition to providing significant operational benefits, BFTBrain improves throughput over fixed protocols by 18% to 119% under dynamic conditions and outperforms state-of-the-art learning based approaches by 44% to 154%.

1 Introduction

Byzantine fault-tolerant (BFT) consensus protocols are the core engines powering the state machine replication (SMR) paradigm, ensuring that non-faulty replicas execute client requests in the same order, despite the existence of f Byzantine replicas. The ability to tolerate up to f arbitrary failures makes BFT protocols a key component in various distributed systems, including permissioned blockchains [6, 48], distributed file systems [19, 22], locking services [23], firewalls [33, 34], key-value stores [27, 35], and SCADA systems [9, 45].

While various BFT protocols have been proposed (e.g., PBFT [18, 19], Zyzzyva [46], CheapBFT [43], Prime [4], SBFT [40] and HotStuff-2 [52]), there is no one-size-fits-all solution. The performance ranking of BFT protocols varies significantly depending on client workloads, network configurations, and application needs. For example, protocols that reduce message complexity by increasing communication phases exhibit better throughput but worse latency. In addition, adversarial behaviors in the system also affect the best-performing protocol choice. The lack of a clear "winner" among BFT protocols makes it difficult for application developers to choose one and may invalidate their choice if workloads or attacks change. This is exacerbated in blockchain systems where application workloads and potential attacks are diverse and dynamic [37].

To address this challenge, two prior systems, Abstract [8, 38] and ADAPT [10] have been proposed. Both systems combine multiple protocols under a single framework and switch between protocols adaptively in real-time. Specifically, Abstract creates an adaptive framework whereby there is a predefined switching order of BFT protocols. If one protocol does not make sufficient progress, it is aborted and the next protocol in the predefined order is selected. Such an approach lacks flexibility and is unlikely to work in all scenarios.

ADAPT enhances protocol selection using supervised learning, but it faces operational limitations. First, it relies on a single replica to collect data, train the machine learning model, and then distribute the decision to all other replicas. In a Byzantine environment, such centralized control is not realistic and runs counter to the entire BFT approach. Second, its requirement for prolonged upfront data collection and supervised learning hinders its ability to adapt to unforeseen conditions and workloads. Lastly, as our experiments will demonstrate, ADAPT's incomplete feature space design renders it ineffective under diverse fault scenarios.

To embrace the idea of a multi-protocol BFT engine while considerably enhancing its practicality, we propose *BFTBrain*, a reinforcement learning (RL) based BFT system. At a high level, given a performance metric to optimize, BFTBrain smartly switches between a set of BFT protocols at run-time under dynamic workloads. BFTBrain is *practical* in two aspects — it not only maintains adaptivity under various adversarial behaviors, but also provides significant *operational benefits* when deployed on different hardware and system configurations: instead of manually choosing from multiple alter-

native systems for deployment, or running a prolonged data collection process prior to the deployment, BFTBrain only requires running one system that automatically re-configures itself to implement a top-performing protocol in real-time.

To leverage RL, BFTBrain measures in real-time performance metrics obtained locally by replicas. Beyond standard metrics like commit throughput, BFTBrain employs fine-grained metrics that offer deeper performance insights, such as the ratio of requests that are committed in the fast path of dual-path protocols, the number of received messages per slot, and the interval between consecutive leader proposals. These new metrics are measured in a distributed manner and serve as features for BFTBrain's RL engine. By modeling the selection of a BFT protocol as a contextual multi-armed bandit problem, the RL engine strategically tests different protocols at run-time to learn which ones are well-suited to the current system conditions. BFTBrain coordinates RL in a decentralized manner, where nodes share local features/rewards by consensus and reach the same learning output, achieving resilience to adversarial data pollution. Our extensive evaluation shows that BFTBrain significantly outperforms fixed protocols and existing adaptive systems in dynamic workloads and faults.

Specifically, this paper makes the following contributions.

BFT based on reinforcement learning. To our best knowledge, BFTBrain is the first BFT system that utilizes reinforcement learning to achieve run-time adaptability. It does not rely on a lengthy data collection process prior to every deployment and effectively adapts to unforeseen system conditions.

Adaptivity to fault scenarios. BFTBrain is the first BFT consensus system that not only automatically adapts to dynamic user workloads under a broad set of hardware and system configurations, but also adapts to various fault scenarios.

Dissecting BFT performance. Through a suite of workload and fault parameters, we conducted a large-scale study examining the relationship between protocol algorithmic design and performance, encompassing a wide range of well-known BFT protocols, including PBFT, Zyzzyva, CheapBFT, Prime, SBFT, and HotStuff-2. Our experiments highlight the vast state space, rendering existing solutions impractical.

Prototype and experimental evaluation. We developed a prototype of BFTBrain and integrated it with the six BFT protocols listed above. Our CloudLab evaluation demonstrates that BFTBrain improves throughput by 18%-119% over fixed protocols under dynamic conditions and outperforms the state-of-the-art learning-based approach by 44%-154% depending on how many data are polluted by adversaries.

2 Landscape of BFT Performance

To motivate BFTBrain, we first conduct an in-depth performance study to examine how various BFT protocols perform under different conditions. The key takeaway is that no single BFT protocol is always "better" or "worse" than others, but rather that different protocols perform better/worse than others under different circumstances.

Table 1: Our observed best-performing BFT protocols under different conditions. The advantage over the second-best protocol is shown in the last column.

f	# of absentees	request size	proposal slowness	best protocol
1	0	4KB	0ms	Zyzzyva (15.6%)
4	0	4KB	0ms	Zyzzyva (34.3%)
4	0	100KB	0ms	CheapBFT (8.5%)
4	4	4KB	0ms	CheapBFT (13.1%)
4	0	0KB	20ms	HotStuff-2 (45.4%)
4	0	1KB	20ms	HotStuff-2 (44.8%)
4	0	0KB	100ms	Prime (16.9%)
1	0	0KB	20ms	Prime (71.5%)

2.1 Comparing Representative BFT Protocols

We picked six representative BFT protocols as our exploration targets: PBFT [18], Zyzzyva [46], CheapBFT [43], Prime [4], SBFT [40], and HotStuff-2 [52]. All six protocols are *leader-based*, working in the partial synchrony settings with networks of $n = 3f + 1$ nodes[1]. Detailed background on these protocols is provided in Appendix A. These protocols and the benchmarking tools were implemented under a common software framework Bedrock [5]; hence, our evaluation below focuses on the impact of algorithmic logic of these protocols on performance, rather than the effect of the implementation details of a specific system.

Our experiments were conducted on CloudLab [28] where each replica is an `xl170` bare metal server. We ran these six protocols under diverse workloads and fault scenarios, and compared their average throughput during 120 second runs that were stably reproduced 10 times. For a fair comparison, the common internal parameters (e.g., batch size and view-change timer) of all six protocols were configured with the same values. For simplicity and facilitating protocol switching, as described in later sections, we added f extra replicas to the original CheapBFT acting as *active* replicas (see [43]). This approach affects the hardware resource consumption and scalability of the original CheapBFT, but does not change its performance under our setups. We also emulated the overhead of the trusted subsystem CASH by injecting 60 μs delay for creating and verifying message certificates.

Table 1 summarizes our evaluation results. Importantly, it demonstrates that **no single protocol dominates in all conditions**. The first four columns indicate the workloads and fault scenarios: system size, number of non-responsive nodes, client request size, and slowness of leader proposals; a more detailed description is provided in Section 4.2. The last column shows the best-performing protocol under each condition as well as its relative percentage advantage in throughput over the second-best protocol; the throughput of all six protocols under each condition is provided in Appendix D.1.

It is worth underscoring that a "slowness-attack", capturing the interval between two consecutive leader proposals, ap-

[1] CheapBFT requires only $2f + 1$ with the help of trusted hardware, but was evaluated with $3f + 1$ for simplicity. Correctness is still guaranteed.

pears in many studies [4, 8, 23]. It captures a Byzantine attack where a malicious leader deliberately postpones its proposal before the view-change timer expires. In addition, it could also happen naturally in a heterogeneous deployment, where a leader has weaker compute or network resources compared to other replicas. Below, we navigate through the table and explain the insights behind the results.

Row 1-3. The first three rows are in an ideal world where every replica is benign and responsive, with no obvious slowness in leader proposals, while varying the network and request sizes. When $n = 4$ ($f = 1$) and the request size is 4KB, Zyzzyva outperforms the next best protocol CheapBFT by 15.6%. When increasing the network size to $n = 13$ ($f = 4$), Zyzzyva outperforms CheapBFT by 34.3%. However, when the request size increases to 100KB, a flip of ranking occurs: CheapBFT becomes the best-performing protocol and outperforms the second best protocol HotStuff-2 by 8.5%, which then slightly outperforms Zyzzyva. The comparison of rows 2 and 3 suggests that different protocols have different "sweet spots" depending on the request size and quorum size. When requests are small, optimistically waiting for $3f + 1$ votes is reasonable, but when requests become larger, waiting for the slowest f nodes to vote on a leader proposal takes a long time, especially when f is large. In the latter scenario, protocols that only need $2f + 1$ replicas to vote on the leader proposal perform better, even at the cost of an extra phase of exchanging small hashes and extra computation. Note that due to the separation of transaction dissemination from consensus logic, only the leader proposals contain the actual requests.

Two of the protocols we studied require a $3f + 1$ quorum in their fast paths: Zyzzyva and SBFT. Across rows 1 to 3 in Table 1, Zyzzyva's performance leads or almost equals SBFT since it has fewer phases of communication. However, we found that with weaker clients, SBFT outperforms Zyzzyva in some cases. More specifically, we reran row 1 on a different hardware setup, where the client machine (which hosts multiple client threads) is configured with fewer CPU cores and higher network latency: this is done by limiting the available CPU cores on a 10-core machine to 6 using `taskset` and injecting an extra 20ms RTT. In this new setup, SBFT outperforms Zyzzyva by 8.5%. These two protocols demonstrate a design trade-off between the number of phases and the choice of commit collector. Since SBFT moves the collector role from clients to certain replicas, it is beneficial when clients have weak network connectivity and compute power.

Row 4. Row 4 demonstrates the effect of non-responsive replicas, referred to as absentees in the table. When certain replicas are non-responsive, the performance of dual-path protocols is adversely impacted, since a slow path is activated only after a timer expiration when failing to gather $3f + 1$ votes on the fast path. Conversely, single-path protocols are less impacted and even have better performance due to fewer message validations and less bookkeeping. In this setting, CheapBFT is the best-performing protocol in our evaluation, which outperforms the next best protocol HotStuff-2 by 13.1% since it has fewer phases of communication. Due to their slow paths, Zyzzyva and SBFT become the bottom-performing protocols. It is worth mentioning that in our evaluation, HotStuff-2 is equipped with a reputation-based leader rotation mechanism, Carousel [24], which tracks active replica participation via their signed votes during the committed chain prefix in order to select the next leaders. For a HotStuff-2 implementation without leader reputation mechanisms, CheapBFT will outperform it by an even larger margin, since it suffers from a non-responsive leader periodically.

Row 5-8. Row 5-8 evaluate different degrees of proposal slowness, representing a Byzantine world where malicious leaders might deliberately slow down the system. When the slowness is as low as 20ms and f is large, HotStuff-2 outperforms all other protocols by up to 45.4%. This is due to HotStuff-2's *routine* leader rotation, which is made possible with low cost by its linear responsive view-change and alleviates the impact of slow nodes elected as leaders. Although Prime also replaces slow leaders proactively, it has more phases and higher communication complexity than HotStuff-2 (i.e., 6 phases and quadratic complexity compared to 2 phases and linear complexity), resulting in worse performance. However, when the slowness further increases to 100ms, or when the network size reduces from 13 to 4 ($f = 4$ to $f = 1$), the ranking flips: Prime outperforms HotStuff-2 by 16.9%-71.5%. This happens since Prime replaces any deliberately slow leader with a *stable* benign leader. In Prime, each node measures the actual turnaround time to the leader, which is independent of the system load, and compares it with the acceptable turnaround time, which is a function of the RTT between correct servers. In HotStuff-2, when the network is small or slowness is high, slow nodes are being rotated in as leaders routinely, causing considerable slowness. This outweighs the benefits of HotStuff-2's simpler and linear communication pattern, resulting in sub-optimal performance.

These experiments demonstrate a complex trade-off between design principles of different BFT protocols. Thus, when conditions change dynamically, no single protocol outperforms others in all scenarios.

2.2 The Case for Reinforcement Learning

Since no single BFT protocol is dominant in all scenarios, one could imagine building heuristics or supervised learning models that map conditions to the best-performing protocol, and switching protocols at run-time according to the current perceived conditions. However, such approaches suffer from several drawbacks.

Condition space size. The condition space is too large to search. Table 1 only presents a sample of the space. The complete condition space we monitor consists of 6 dimensions (State 1 and 2 in Section 4.2), where each dimension is either a continuous or discrete variable. Further, each point in the condition space has multiple protocols to experiment with.

Even with coarse-grained sampling and an automated toolkit, it took us more than a week to experimentally explore just a small subset of the condition space. Unfortunately, building good heuristics and supervised models requires *complete* data, which are hard to obtain.

Hardware and time dependence. The mapping from conditions to the best-performing protocol depends on the underlying hardware and system configuration: when changing from `xl170` to `m510` instances on CloudLab, or changing from a 13-node network to a 4-node network, the mapping changes. Additionally, we observe that even with the same hardware instance type but across different launches on shared facilities like CloudLab, the sweet spots (in terms of request size) of Zyzzyva and CheapBFT change due to subtle differences in physical machines and network resource availability. Worse yet, network conditions can change over time, rendering any pre-computed mapping on a specific network less useful.

Growing protocol space. When new BFT protocols emerge (e.g., HotStuff-2) or changes are introduced to existing implementations (e.g., DiemBFT-v1 to v4 [1]), any precomputed mapping would need to be recomputed. In other words, one would need to re-collect data and re-craft the heuristics, or re-collect data and retrain the supervised model virtually every time a new BFT consensus protocol is proposed.

Reinforcement learning addresses this daunting and complex task, and has shown superior performance in other learned systems [20, 53, 54, 62]. Unlike supervised learning, which assumes training data is complete and requires a separate lengthy data collection process prior to deployment, one can simply plug and play an RL-based system — it learns from its mistakes and optimizes long-term rewards through trials in an online fashion. With reinforcement learning, BFTBrain can optimize itself to whatever client workloads, faults, hardware, system configurations and BFT protocols present, providing adaptivity and significant operational benefits.

3 BFTBrain Overview

We provide an overview of BFTBrain, first outlining its system model followed by its overall reinforcement learning based design.

3.1 System Model

In BFTBrain, we assume a system consists of a fixed set of $n = 3f + 1$ *nodes* and a finite population of *clients*, where up to f nodes and any number of clients might experience Byzantine faults. Each node serves two roles simultaneously, *validator* and *learning agent*. A validator is responsible for totally ordering the blocks, while a learning agent coordinates online data collection, trains machine learning models, and periodically instructs the companion validator at run-time to replace the current BFT protocol.

When a node is faulty, it can behave arbitrarily in any of its roles. The faulty validators may exhibit standard malicious behavior such as double voting and message suppression or equivocation. We assume a strong adversary that can coordinate the faulty validators to compromise the replicated service. However, we do assume the adversary cannot break the cryptographic techniques. Given our use of machine learning, faulty learning agents could exhibit additional malicious behaviors related to learning, such as being non-responsive (e.g., refuse to exchange its locally measured data with other learning agents), as well as presenting *arbitrarily manipulated* local data points (i.e., local features and rewards) in order to disrupt the machine learning models on other agents.

Network communication is point-to-point and authenticated. BFTBrain adopts the *partial synchrony model* [29], where there is a known bound Δ and an unknown Global Stabilization Time (GST), such that after GST, all transmissions between two correct nodes arrive within time Δ. For two different roles on the same node, we assume their communication in-between is always synchronous.

3.2 Design Overview

At a high level, BFTBrain contains three key components: (1) a reinforcement learning algorithm (i.e., the core of the learning agent) that guides the choice of BFT protocols according to the perceived underlying dynamic environment, (2) a coordination protocol that collects data distributedly at run-time, and (3) a switching mechanism that allows BFTBrain to seamlessly transition from one BFT protocol to another while ensuring safety and liveness.

BFTBrain operates in *epochs*, where each epoch is marked by the completion of k blocks. Here, k is a predefined constant hyper-parameter. Within one epoch, the protocol remains unchanged. When the learning agent finds a protocol candidate, it instructs the validator to use that protocol for the next epoch. We next introduce BFTBrain's key components and workflow.

Learning agent. BFTBrain's learning agent models the problem of selecting a BFT protocol as a contextual multi-armed bandit (CMAB) problem [67]: periodically, BFTBrain examines the most recent state of the workload and faults in the system (*context*), and then selects one of many BFT protocols (*arms*) in our protocol pool. After making the selection, it observes the performance of the newly selected protocol (*reward*). To be successful, BFTBrain must balance the *exploration* of new, untested protocols with *exploiting* past experience to maximize performance. That is, without a careful balance of exploration and exploitation, BFTBrain risks failing to discover an optimal protocol (too much exploitation), or performing no better than random (too much exploration). We select this CMAB formulation (as opposed to full reinforcement learning) because CMABs are exceptionally well-studied, enable faster convergence, and many asymptotically-optimal algorithms exist to solve them [2, 21]. Details about the learning algorithms are provided in Section 4.

Since BFTBrain operates in a Byzantine environment, a centralized learning agent cannot be trusted. In BFTBrain, each validator process has a companion learning agent run-

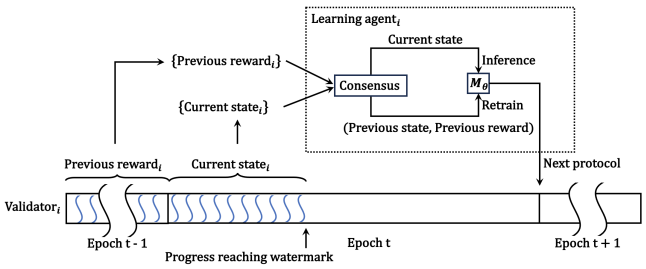


Figure 1: Overview of BFTBrain. For readability, we only present the internals of one node i.

ning on the same node, and accepts instructions only from its companion learning agent. The learning agents themselves also form a state machine replication. Specifically, they start with the same *initial state*, i.e., the same random seed of machine learning models. For the same epoch t, as we will show later, different learning agents agree on the same *sequence of operations*, i.e., training data points where each data point consists of context and reward. With deterministic training, benign learning agents host the same parameters for their machine learning models. As a result, if different learning agents perceive the same context for epoch $t+1$, they will render the same decision (i.e., choice of protocol) for epoch $t+1$.

Distributed online data collection. In a Byzantine environment, no centralized entity could be trusted to collect training data. Therefore, the learning agents in BFTBrain also participate in a protocol that coordinates distributed data collection in an online fashion. At a high level, for every epoch, each learning agent monitors its local context and reward at runtime, then exchanges them with other agents via a separate instance of BFT consensus independent of the consensus that validators are running. For each epoch, agents form agreement over an aggregation of contexts and rewards that include input from at least a quorum of two-thirds of agents. The consensus algorithm used for forming this agreement is left open for the system designer (note that it is invoked only once per epoch, hence does not need to have high throughput). Once an agreed quorum of local contexts and rewards is obtained, each learning agent can apply the same robustness filter to the quorum in order to get a global context and reward, constituting a training data point. Details about this learning-coordination protocol are provided in Section 5.

Switching BFT protocols. After a BFT protocol is selected by the learning agent, the switching mechanism allows each validator to make use of this protocol for the next epoch. Our switching mechanism is an improvement over Abstract [8], which aborts a BFT instance if a certain progress condition is not met. An epoch in BFTBrain is equivalent to a *Backup* instance in Abstract. Due to space limits, details on how we switch BFT protocols are provided in Appendix B.

BFTBrain workflow overview. Figure 1 presents an overview of BFTBrain, where each node in the system follows the same workflow as depicted. In the middle of epoch t, when the number of executed blocks reaches a certain watermark, the validator on node i notifies its local learning agent. The learning agent featurizes its current local state (i.e., context) observed in epoch t, and uses it to approximate the next local state $state_i^{t+1}$ for epoch $t+1$. Each agent exchanges $state_i^{t+1}$ and its locally measured reward of previous epoch $reward_i^{t-1}$ with other agents via the learning-coordination protocol. Therefore, each agent obtains the same global state $state^{t+1}$ and reward $reward^{t-1}$. Subsequently, each agent adds the $(state^{t-1}, protocol^{t-1}, reward^{t-1})$ triplet to its experience buffer, and retrains its predictive model M_θ based on its experience buffer as well as the chosen algorithm to solve CMAB. Once retrained, the predictive model M_θ inferences the performance of each protocol candidate under $state^{t+1}$, and selects $protocol^{t+1}$ that is predicted to have the best performance. The learning agent then informs the validator to switch to $protocol^{t+1}$ for epoch $t+1$, the reward of which is then measured locally upon reaching the end of epoch $t+1$. The validator only starts epoch $t+1$ once it receives a decision for that epoch from the companion agent.

The learning agent is designed in a way such that the BFT system is not delayed due to learning. First, within one epoch, when the model undergoes retraining and inference, the parallel validator process still commits blocks simultaneously. Second, with a lightweight model design and limiting the size of the experience buffer, model training and inference can be viewed as a synchronous process. In other words, with a reasonable epoch length, the learning agent can complete protocol selection before the validator finishes its current epoch, without impeding the start of the next epoch.

4 Learning Algorithms

This section delves into BFTBrain's learning algorithms. We first formalize the learning problem and explain the use of Thompson sampling. The state and action space design is then outlined, followed by the predictive model description.

4.1 Problem Formulation

BFTBrain formulates the learning problem as a contextual multi-armed bandit (CMAB) problem, where an agent periodically makes decisions in a sequence of epochs. In epoch t, the agent selects an action a_t in its protocol pool based on a provided state s_t, and then receives a reward r_t. The agent's goal is to select actions in a way that minimizes *regret*, i.e., the difference between the reward sum associated with an optimal selection strategy and the reward sum associated with the chosen actions. CMABs assume that epochs are independent from each other, and that the optimal action depends only on the state s_t. BFTBrain fits this assumption, since the current choice of protocol does not affect the pattern of workloads and faults in future epochs. Using this CMAB formulation, BFTBrain allows users to specify any performance metric (e.g., throughput or latency) as the reward function to optimize.

Thompson sampling. Amongst different CMAB algorithms, we select Thompson sampling [2, 21] for its simplicity: at the start of each epoch, the learning agent trains a model based on current experience, and then selects the best action as predicted by the model. In Thompson sampling, instead of selecting the model parameters that are most likely given the training data as used by supervised learning, it *samples model parameters proportionally to their likelihood given the training data*. More formally, we can define maximum likelihood estimation as finding the model parameters θ that maximize likelihood given experience E: $\arg\max_\theta P(\theta \mid E)$ (assuming a uniform prior). Instead of maximizing likelihood, Thompson sampling simply samples from the distribution $P(\theta \mid E)$. As a result, if we have a lot of data suggesting that our model weights should be in a certain part of the parameter space, our sampled parameters are likely to be in that part of the space. Conversely, if we have only a small amount of data suggesting that our model weights should be in a certain part of the parameter space, we may or may not sample parameters in that part of the space during any given epoch.

4.2 State and Action Space

We next list factors that affect the performance of BFT protocols, broadly grouped into workloads, faults, and hardware/system configurations categories, jointly constituting the state space. Within each epoch, each learning agent leverages a window of the last w executed requests to featurize such factors, where w is a constant hyper-parameter.

State 1: Workloads (W). The first category consists of factors that are influenced by application and client dynamics.

W1: Request size. The request size is dependent on the application workload, where some requests contain little data while others are more involved and require updating files with large chunks of data. Although all our candidate protocols separate request dissemination from sequencing (i.e., only the leader proposals contain the actual requests while the remaining messages contain the hash of requests), as described in Section 2, request size is still an important factor impacting the performance of different protocols in different ways. We use the average request size to represent this feature.

W2: Reply size. Depending on the application, request and reply size can be asymmetric. Reply size also impacts different protocols in different ways, but with a distinct boundary from that of request size. For instance, in our experiments, when all other factors remain the same, CheapBFT performs best for either 0KB/4KB (request size/reply size) or 40KB/0KB, while Zyzzyva is best with 4KB/0KB. Furthermore, different protocols have different sensitivity to an increase in reply size. For instance, SBFT reduces the per-client linear reply cost to just one message by adding an execution aggregation phase. We use the average reply size to represent this feature.

W3: Load on system. The load on the BFT system is dependent on the number of clients and the rate at which clients send new requests. Specifically, each honest client allows a *quota* of outstanding unacknowledged requests before issuing new ones, controlling the rate at which requests are generated relative to the system's capacity to process them.

In our experiments, we observed that lowering the load affects different protocols in the same direction, but to different extents: for example, there is a sharper drop in the throughput of Zyzzyva and SBFT compared with other protocols. The reason is that lower load increases batching delay, which has a larger impact in terms of both latency and throughput on protocols with fewer phases (i.e., Zyzzyva and SBFT in our protocol pool). BFTBrain derives the per-client sending rate according to the request timestamps, and uses the aggregated client sending rate to represent this feature.

W4: Execution overhead. Execution overhead captures the computational cost of request execution, which impacts the system in two ways. First, it directly affects the execution latency in state machine replication. Second, it indirectly affects other components of the BFT protocol that are also compute-intensive. For instance, requests with high execution overhead compete for CPU resources that are otherwise used to sign and verify messages, especially when machines have limited compute capacity or a small number of cores. Higher compute load results in excessive context switching, and potentially pushes the system towards being compute-bound instead of network-bound. We use the CPU cycles consumed by the executor thread to represent this feature.

State 2: Faults (F). The next category of factors is tied to faulty behaviors. BFT protocols make different assumptions about "steady state" and "common faults", and hence, each protocol is often optimized for specific fault scenarios. The features below enable BFTBrain to tell what type of fault scenarios the system is experiencing and choose the most promising protocol accordingly.

Note that BFTBrain does not aim to defend against *transient* or *broad-spectrum* faults. The reason is, transient faults are already handled timely at the protocol level and require no protocol switching. An example would be a crashed leader or a malicious leader which equivocates, such that no progress is made and a view-change will be triggered to replace the leader. For broad-spectrum faults, effective and orthogonal solutions already exist. Examples of broad-spectrum faults include network flooding which can be resolved via resource isolation [8], and malformed client requests which can be handled by enforcing client signatures (instead of MACs) [23].

F1: Absence from participation. Measuring absence is tricky in a Byzantine environment, since information reported by malicious nodes could deviate from their actual participation. Moreover, a collusion of f malicious participants could taint validator participation simply by *excluding* some alive benign validators (up to f) and progressing without them. For example, a malicious leader could deliberately avoid sending leader proposals to them, while the f malicious validators work together with the remaining $f+1$ benign ones to make sure all requests are committed successfully on these $2f+1$ valida-

tors. We refer to such excluded, non-faulty validators as being placed *in-dark*. In-dark validators could further be excluded in other protocol phases, in addition to leader proposals, by f malicious validators. Since no state transition is ever triggered on in-dark validators, they remain in the initial state and are thus absent from participation. Although they will timeout and complain, since fewer than $f+1$ validators complain, view-change is not triggered to replace the malicious leader and they are in-dark continuously.

All protocols tolerate the absence of up to f validators by design, but the performance of different protocols is impacted differently. As illustrated in Section 2, dual-path protocols (i.e., Zyzzyva, SBFT) are adversely impacted since the more expensive slow paths are initiated, while single-path protocols (i.e., PBFT, CheapBFT, Prime, HotStuff-2) could be positively impacted due to less resource consumption.

The learning agents in BFTBrain featurize absence by utilizing information that is already collected locally during protocol execution. First, *fast path ratio* captures the percentage of slots committed in the fast path over the total number of committed slots. For single-path protocols, all slots are committed in the slow path. Second, for each slot, the agent sums the number of (valid) distinct messages from each sender, deriving the *number of received messages per slot*. Note that this feature does not incur more messages to be sent or received; it simply counts messages as they arrive and pass preliminary processing (de-serializing and sender verification) before they can be excluded from protocol steps like voting. In fact, these two features could capture a sufficiently big subset of abnormal behaviors, encompassing not only omission failures but also leader equivocation and network partitions.

F2: Slowness of proposal. In leader-based BFT protocols, every slot is initiated by a leader proposal, which significantly affects the system's end-to-end performance. In the case of a faulty leader, validators use a timer to trigger view-change, which will replace the leader, hence guaranteeing liveness. However, a malicious leader can deliberately slow down its proposals without being replaced, resulting in poor latency and throughput. Slowness may not necessarily be a deliberate act by a malicious leader, it could result simply from a weak or overloaded leader, albeit to a lighter extent.

Section 2 illustrates the effect of this factor. Protocols with routine or proactive leader replacement (i.e., HotStuff-2, Prime) maintain good performance under such slowness, but perform sub-optimally in other normal cases. To featurize this factor, each node in BFTBrain timestamps every leader proposal received, and measures the average time interval between receiving two consecutive proposals.

State 3: Hardware and system configurations. The last category comprises hardware and system configurations. Hardware-level factors include network settings that affect latency and bandwidth, and also machine configurations such as CPU frequency and the number of cores. We assume the network latency and bandwidth follow certain non-shifting distributions that capture noises in the network. System-level configurations include the number of nodes in the system and the geo-distribution of the nodes. Compared to State 1 and 2, both hardware and system configurations are fairly static and do not change rapidly at the timescale of a consensus deployment. Thus, BFTBrain does not explicitly featurize these factors because (1) the impact of these factors is implicitly encoded in the predictive model trained online, and (2) CMABs will converge even without any explicit features [12], the purpose of which is to accelerate convergence so that the optimal action is reached before the world changes.

Actions. BFTBrain's action space consists of the same set of leader-based protocols that are studied in Section 2, namely, PBFT, Zyzzyva, CheapBFT, Prime, SBFT, and HotStuff-2. As we will show later, since BFTBrain builds a separate model for each candidate protocol, features of the protocol design are encoded in the model itself. Thus, BFTBrain does not require protocol-specific feature engineering for encoding the action space, making it easy to incorporate new protocols into the BFTBrain framework. Although BFTBrain focuses on choosing the best-performing protocol, configuring protocol internal parameters (e.g., values of different timers and the interval for rotating leaders) intelligently at run-time can be an interesting future extension.

4.3 Predictive Model

Each learning agent hosts some predictive models, which follow the *value based* RL approach: given the featurized current state, predict the performance (i.e., reward) of each action (i.e., protocol). The simplest implementation would be to build a single predictive model for all protocols, but this has a major drawback. While features *W1-W4* in the workloads category are completely independent from the previous action, the featurized absence-from-participation $F1$ and slowness-of-proposal $F2$ have a "one-step dependency": the current observed $state^{t+1}_{F1,F2}$ is dependent on the previous $protocol^t$. When the workload and fault scenario shift, such one-step dependency might prevent convergence to the new optimal protocol. For instance, if BFTBrain has converged to protocols whose leader has lower parallelism (e.g., Prime due to message aggregation in global ordering), the measured slowness of proposals will be higher than other protocols, regardless of whether a fault is actually happening or not. In other words, the interpretation of the slowness-of-proposal feature changes based on the previous action. If the model learns that a high slowness-of-proposal is bad for Zyzzyva, then once Prime is chosen, slowness-of-proposal will always seem high, and Zyzzyva may not ever be selected again.

Luckily, unlike in the general case of Markov decision processes, the dependency observed here is limited to a single time step. That is, the prediction of the next best action is independent *given the immediately prior action*. To solve this issue rooted in one-step dependency, the learning agent trains a separate model for each possible (previous protocol,

protocol) pair, and divides the experience buffer into several smaller *buckets* according to the (previous protocol, protocol) pair as well. In terms of bandit theory, assuming there are K protocols in our action space, BFTBrain's approach is equivalent to playing K bandit games where each game has K arms. In each game, the current observed $state^{t+1}_{F1,F2}$ is independent from the previous action $protocol^t$.

It is worth mentioning that this transformation does not completely remove the one-step dependency. The action made at epoch t will determine which of the K bandit games is being played at epoch $t+1$. A multi-armed bandit algorithm will not be able to take advantage of the fact that some of the K bandits may have significantly better reward distributions than others. Thus, the convergence bound on regret of bandit algorithms will not apply to our scenario. However, since each of the K bandits will be played an unbounded number of times eventually (assuming the probability of any action never fully reaches 0), regret is still bounded in the limit.

Specifically, for each possible $(protocol^t, protocol^{t+1})$ pair, BFTBrain uses a lightweight random forest [15] as the predictive model, which is trained exclusively on the corresponding experience bucket. The model takes the featurized state as input, and outputs the predicted performance for the corresponding candidate action $protocol^{t+1}$. Thus, at inference time, given a known previous protocol and the current state, the learning agent enumerates K models to get the predicted performance for each candidate protocol, and then chooses the candidate with the best predicted performance to be carried out. Once there is a tie on the best predicted performance, we break the tie randomly to avoid local maxima. When an experience bucket is empty, BFTBrain prioritizes exploring this bucket by choosing the corresponding candidate protocol to be carried out.

Integration with Thompson sampling. Integrating a predictive model with Thompson sampling requires the ability to sample model parameters from $P(\theta \mid E)$ — the distribution of model parameters given the current experience. The simplest technique (which has been shown to work well in practice [55]) is to train the model as usual, but only on a bootstrap [14] of the training data. In other words, the predictive model is trained using $|E|$ random samples drawn with replacement from experience E, inducing the desired sampling properties. BFTBrain uses this bootstrapping technique on each experience bucket and predictive model for its simplicity.

Overhead of learning. First, BFTBrain's training overhead is not larger than the strawman of building a single predictive model, since in every epoch, only one model which corresponds to the updated bucket needs to be retrained. For such a bucket, the time complexity for training a single random forest is $O(n \log n)$, where n is the number of data points. Thus, given the same total population of data, it even incurs less training overhead than the strawman solution, since the bucket contains fewer data points than the single unified experience buffer. Second, BFTBrain's inference overhead is $O(K)$, where K is the number of candidate protocols. Lastly, BFTBrain has the same memory overhead as the strawman for storing training data. However, it incurs $O(K^2)$ memory overhead for storing the models. Since random forest is a very lightweight model as compared to deep neural networks, such model storage overhead is negligible.

5 Learning Coordination

The goal of the *learning-coordination* mechanism is to form an agreement at each epoch on a *report quorum* that includes local metrics collected from $2f+1$ nodes.

Specifically, learning coordination is performed in every epoch t. After executing w requests (a hyper-parameter) in epoch t, each node i gathers local performance indicators p_i^{t-1} measured during epoch $t-1$, featurizes the next state f_i^{t+1}, and broadcasts both metrics inside a report message. To ensure that at least $f+1$ metrics in the report quorum are honest measurements, it is important that the metrics reported by honest nodes are measured by **themselves**. That is, if a node j has been placed in-dark (defined in Section 4.2) or temporarily slows down during epoch t, it may not have executed w requests by itself. Rather, node j will have recovered the consensus state through a *state-transfer* from other nodes. In this case, j should avoid reporting the state features it has copied from others, and likewise, avoid reporting performance indicators collected from partial or no execution. Therefore, node j will not report any metrics for epoch t. Note that in addition to the f benign nodes being placed in-dark, in the meantime, the f Byzantine nodes that contributed to committing requests can refuse to report their metrics. Hence, there may not be enough $2f+1$ nodes reporting for the epoch.

In order for nodes to agree on a quorum of (valid) reports to be used as input for the learning engine, any "blackbox" validated Byzantine consensus primitive (VBC) seeded with a leader collecting reports from $2f+1$ nodes can be utilized. Specifically, for each epoch t, the leader of VBC initiates VBC-PROPOSE($(t, reportQC^t)$, P) once it receives valid report messages (p_i^{t-1}, f_i^{t+1}) where both fields are non-null from $2f+1$ nodes, or when a timer expires. Here, P is an external validity predicate that checks if $reportQC^t$ includes at least $f+1$ distinct reports.

Each node participating in VBC gates voting for a leader proposal it receives by applying the validity predicate P to it. Once a quorum of reports $reportQC^t$ is decided by VBC, if it includes sufficient $2f+1$ reports, each node takes the median value of each field in order to obtain a robust global performance measurement p^{t-1} and state feature f^{t+1}, thereby triggering the retraining and inference process. Taking the median value from an aggregated set of metrics guarantees that despite f arbitrarily manipulated values from Byzantine nodes, the global value taken is between two honest measurement values. Otherwise, if $reportQC^t$ does not include sufficient reports, each node retains the decision from the

previous epoch instead of deriving any new learning decision, and complains about the leader in VBC as well as the leader in the current protocol used by the node for committing client requests. Note that since VBC is a separate consensus instance, the leader of VBC can be different from the leader of the current protocol in BFTBrain. Either of them acting maliciously can result in insufficient reports being collected.

For completeness, Appendix C.1 presents in Algorithm 1 a detailed learning coordination protocol where VBC is implemented using PBFT. Due to space limitations, the safety, liveness, and robustness guarantees of BFTBrain's learning agents are discussed and proved in Appendix C.2.

6 Implementation

We implemented BFTBrain in Java using Bedrock [5], a unified platform for BFT protocol implementation and experimentation. The Bedrock platform consists of four main components: the core unit, the state machine manager, the plugin manager, and the coordination unit. The core unit defines entities (e.g., clients and validators), maintains the application logic, enables users to specify different workloads and benchmarks, and track the execution of requests via sequence numbers and views. The state machine manager parses Bedrock's domain-specific language (DSL) for rapidly prototyping BFT protocols, and defines the states and transitions of the specific BFT protocol for each entity. The plugin manager enables implementing protocol-specific behaviors that cannot be captured by Bedrock's DSL, while the coordination unit manages the run-time execution of Bedrock.

All protocol candidates in BFTBrain's action space utilize Bedrock. On top of it, we implemented a workload and fault generator, which allows users to specify time-varying dynamics inside a `YAML` configuration file, enabling parameterized random sampling and scheduling of predefined sequences of events. To enable switching among BFT protocols, we implemented a new state machine manager which loads all plugins required by BFTBrain when the system boots, and uniquely tags each protocol state and transition such that different protocols do not interfere. This approach has negligible overhead since such loading and tagging only happens when the system boots, and lookup for states and transitions at run-time takes a (small) constant time. We implemented the learning agent separately in Python using `scikit-learn`, which communicates with the companion validator process in Bedrock via gRPC. Our implementation is publicly available[2].

7 Evaluation

Our evaluation aims to answer the following questions:

1. How fast can BFTBrain converge to the best-performing protocol under static conditions without pre-training?
2. How does BFTBrain compare to fixed protocols and existing learning-based approaches in dynamic environments?
3. How does the hardware setup affect the performance of fixed protocols? How does BFTBrain compare to existing learning approaches under different hardware setups?
4. How robust is BFTBrain and how does it compare to existing learning approaches under adversarial data pollution?
5. What learning overhead does BFTBrain introduce?

In the rest of this section, we present the experimental setup and then answer each of the above questions.

7.1 Experimental Setup

Testbed. Unless otherwise specified, our testbed consists of several bare-metal xl170 machines on CloudLab, each with a 10-core Intel E5-2640v4 processor at 2.4 GHz, 64GB ECC Memory, and two dual-port Mellanox ConnectX-4 25 GB NIC. Each server is connected via a 10Gbps control link and a 25Gbps experimental link to Mellanox 2410 switches in groups of 40 servers. Our experiments are conducted using the experimental link.

System configuration. Our experiments are carried over networks of two sizes: $n = 4$ and $n = 13$. In all experiments, we run multiple client threads on a separate xl170 server, where each client follows the standard closed-loop buffer design, i.e., allows at most 100 outstanding unacknowledged requests before issuing new ones. The number of clients is a parameter in our workload space. We use batch sizes of 10 requests throughout the experiments. Every experiment of BFTBrain starts with PBFT as its initial protocol. We use throughput as the reward function for BFTBrain to optimize.

7.2 Convergence under Static Conditions

Our first set of experiments evaluates how quickly can BFTBrain converge to the best-performing protocol under static conditions. We picked three representative settings from Table 1 where the size of the network is small: row 1, a variant of row 4 where $f = 1$, and row 8. Under each setting, we ran all six fixed protocols and BFTBrain for 10 minutes on a LAN. We note that the other rows in Table 1 yielded similar conclusions and are omitted for brevity.

Table 2 summarizes our results, listing for each static condition the average throughput (tps) of each protocol in the last 20 epochs. The last column reports the convergence time of BFTBrain, defined as the time BFTBrain spent to reach the stable peak throughput. Although each setting has a different best-performing protocol, BFTBrain always learns to select it within 0.81-5.39 minutes[3]. As we will show soon, the convergence is significantly accelerated the second time BFTBrain encounters the same conditions.

Obviously, in this experiment, BFTBrain is not able to surpass the best-performing protocol in any given setting. Additionally, as shown in the table, it experiences a marginal increase in switching overhead when compared to the leading protocol in each specific setting. Nevertheless, owing to its adaptability, BFTBrain does deliver the best average and best

[2] https://github.com/JeffersonQin/BFTBrain

[3] Most RL systems quantify convergence time in minutes.

Table 2: Throughput of protocols and the convergence time of BFTBrain under various static conditions. The highest throughput in each row is highlighted in **blue**.

Condition	Average Throughput (tps)							Conv. Time (minutes)
	PBFT	Zyzzyva	CheapBFT	Prime	SBFT	HotStuff-2	BFTBrain	
Row 1 (LAN)	9133	**13664**	11822	4601	11067	6882	13100	0.81
Row 4 (LAN)*	10303	1025	**12297**	3749	2920	5156	11803	2.08
Row 8 (LAN)	989	988	989	**4527**	989	2640	4329	5.39
Row 1 (WAN)	5325	9503	**12201**	1639	8261	2882	11101	1.58
Average	6438	6295	9327	3629	5809	4390	**10083**	2.47
Worst	989	988	989	1639	989	2640	**4329**	5.39

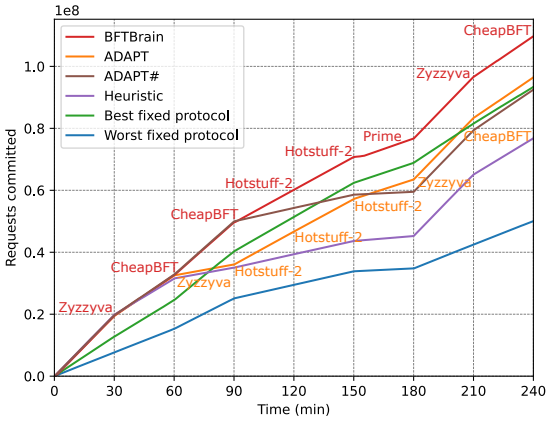


Figure 2: Adaptivity of BFTBrain under changing conditions. The vertical dashed lines indicate when conditions change. Labels indicate the dominant protocol that BFTBrain and ADAPT choose under each condition.

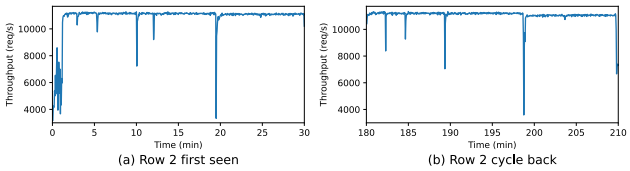


Figure 3: BFTBrain's throughput (a) during minutes 0-30, and (b) during minutes 180-210. In both periods, it encounters the system conditions captured in row 2 of Table 1.

worst-case performance across all settings in the table. In contrast, fixed protocols exhibit subpar average and worst-case throughput, underscoring our initial premise that no single fixed protocol excels under all circumstances.

7.3 Adaptivity under Changing Conditions

Our next set of experiments evaluate BFTBrain under dynamic conditions, demonstrating clear performance benefits.

Cycle back conditions. In the initial series of experiments, we selected rows 2-7 from Table 1, each characterized by an identical network size ($f = 4$). We ran the settings of each row for 30 minutes each, employing a round-robin approach to switch to the next row and repeating the cycle from the beginning after reaching row 7. We compared BFTBrain against five baselines: the fixed protocols with the best and worst performance (based on the number of committed requests throughout the entire experiment), ADAPT, ADAPT#, and an expert heuristic. Details on the last three are explained below.

To be faithful to ADAPT's design [10], we excluded all features capturing faults (i.e., State 2 in Section 4.2), and pre-trained ADAPT with complete data that we collected in these changing conditions when running BFTBrain for hours spanning multiple protocols. To study the effect of *unseen conditions* in supervised learning, we implemented another baseline named ADAPT#. In ADAPT#, we used the same set of complete features as BFTBrain, but pre-trained it on partial data that we collected in these changing conditions, i.e., excluding data corresponding to the settings of rows 5-7. In both ADAPT and ADAPT#, for fair comparison, we used the same set of BFT protocols as BFTBrain in their action space. The expert heuristic we used is designed based on insights gleaned from Table 1, namely: if proposal slowness is greater than 20ms, use Prime; otherwise, use Zyzzyva.

Figure 2 shows the cumulative number of committed requests over time, where the slope of each line indicates its current throughput. Throughout the 4-hour experiment, BFTBrain successfully converges to the best-performing protocols whenever the conditions change. In terms of the number of requests committed, BFTBrain demonstrates 18% improvement over the best fixed protocol (i.e., HotStuff-2), 119% over the worst fixed protocol (i.e., PBFT), 14% over ADAPT, 19% over ADAPT#, and 43% over heuristic.

As expected, both ADAPT and ADAPT# exhibit similar performance to BFTBrain for the first 60 minutes, as the system is operating without faults and their features and training data encompass these conditions. However, both methods encounter challenges when the conditions shift to rows 4-7. ADAPT struggles since it fails to recognize the changes in conditions, as faults are not captured in its feature space. While ADAPT# does detect the changes due to its expanded feature set, these conditions were absent during its pre-training phase.

Two things are worthy of noting. First, BFTBrain not only outperforms both supervised learning based approaches, but also completely removes the cumbersome data collection and pre-training process prior to deployment. Second, in common stable conditions corresponding to row 2, BFTBrain

outperforms HotStuff-2 by 57% while being able to switch to HotStuff-2 when conditions become advantageous for it.

Figure 3 further demonstrates how BFTBrain's throughput changes over time: (a) during minutes 0-30, and (b) during minutes 180-210. In both periods, it encounters the conditions captured in row 2 of Table 1. Obviously, when the system condition cycles back to what it has seen before, BFTBrain converges to the best-performing protocol much faster than the first time (2s vs. 70s). The blips in both throughput plots indicate that certain sub-optimal protocols have been chosen for a few epochs. These are the explorations made by the Thompson sampling algorithm when it samples the less likely model parameters. When conditions are dynamic, such sampling is crucial to avoid being stuck at a local sub-optimal decision, since the predictive model can gain insights on the performance of unexplored protocols in the current condition.

Randomized sampling. Our next experiment evaluates BFTBrain's adaptivity when there are *more variations* in the state space and when the variations are *more frequent*. Specifically, each dimension in the state space follows a certain normal distribution. We varied each dimension every 1s by randomly sampling from its distribution, and shifted the mean and variance of such distribution every 20 minutes. Appendix D.2 presents the detailed experiment setup and results. In a nutshell, during the entire 2-hour deployment, BFTBrain commits 44% more requests than ADAPT. This is a much larger improvement than the 14% improvement demonstrated in the cycle back experiment and can be explained as follows. Certain input factors in the cycle back experiment are correlated, e.g., a request size near zero is correlated with high proposal slowness. Thus, although ADAPT suffers from incomplete features, it indirectly learns the best-performing protocol under high proposal slowness using other features, as shown in Figure 2 between 90-150 minutes. However, randomized sampling breaks such correlations, and as a result, ADAPT does not have enough information to know how the surrounding environment changes which leads to poor performance.

7.4 Adaptivity to Changing Hardware

The next set of experiments evaluate BFTBrain under different hardware setups from our initial protocol evaluation, and compare with the adaptivity of ADAPT to these new setups. Specifically, we reran the settings of row 1 from Table 1 on a live WAN (RTT=38.7ms, bandwidth=559Mbps) instead of on a LAN, where we placed two nodes in Cloudlab's Utah data center on xl170 machines and the other two nodes in the Wisconsin data center on c220g5 machines. We compared BFTBrain with fixed protocols and with ADAPT in this setup.

The results are summarized under "Row 1 (WAN)" in Table 2; the detailed performance plot of BFTBrain and ADAPT is provided in Appendix D.3. Interestingly, compared to "Row 1 (LAN)" where Zyzzyva is the best protocol and leads by 15.6%, running the same workload and fault conditions on WAN with different instance types renders CheapBFT the best protocol, which outperforms Zyzzyva by 28.4%. The reason is that Zyzzyva commits via the fast path and requires votes from all nodes, resulting in high end-to-end latency, given the high RTT between some nodes. CheapBFT, on the other hand, requires a smaller quorum of voters, which could be co-located inside a single data center, thus achieving lower latency. With a moderate number of clients (i.e., 50 in our workload), this results in CheapBFT also achieving higher throughput than Zyzzyva due to finite parallelism. Under this unseen hardware setup, BFTBrain converges to the best-performing protocol within 1.58 minutes. We also observe that when running in a live WAN for hours, BFTBrain remains robust to noises in the network which occasionally cause significant drops in the transient throughput of certain epochs. On the contrary, ADAPT cannot transfer the knowledge it learned on different hardware setups. As a result, when ADAPT is pre-trained with complete data we collected on "Row 1 (LAN)", it remains stuck at the sub-optimal decision Zyzzyva.

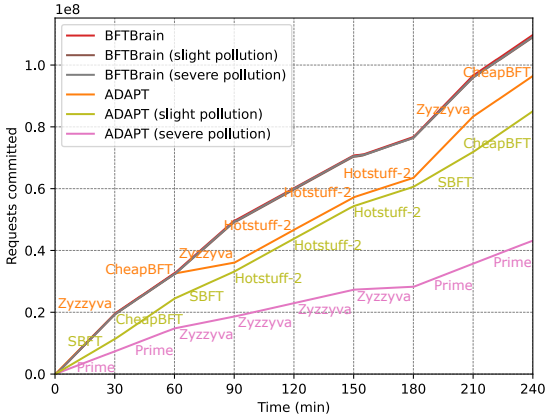


Figure 4: Robustness of BFTBrain against data pollution. The vertical dashed lines indicate when conditions change. Labels indicate the dominant protocol that ADAPT (before and after pollution) chooses under each setting.

This experiment demonstrates the operational benefits of BFTBrain when hardware configurations may differ across deployments. BFTBrain can rapidly converge to the best-performing protocol for new hardware settings without any human intervention. As discussed before, it may not be practical to recollect data and pre-train a supervised approach such as ADAPT on every new hardware setup.

7.5 Robustness of BFTBrain

In a Byzantine environment, malicious nodes might arbitrarily manipulate (i.e., pollute) their collected metrics in order to misguide the machine learning models. Our next experiment compares the resilience of BFTBrain and ADAPT under data pollution. We ran the same dynamic benchmark as the "cycle back conditions" experiment in Section 7.3, and compared BFTBrain with ADAPT's performance under two types of data pollution: slight and severe. In the slight pollution scenario, only the reward (i.e., throughput) of SBFT was increased to

2.5x of its true value. In the severe pollution, regardless of the protocol, for every single data point, both its state and reward were polluted by replacing the true values with random values. The random value was chosen uniformly between 0 and 5x the maximal true value for the polluted dimension. Note that when polluting BFTBrain, f learning agents were malicious and hence reported polluted values as their local measurements; when polluting ADAPT, the malicious centralized entity polluted training data and distributed the same data to all nodes in the system.

Figure 4 shows the number of committed requests with respect to time, under different types of pollution. Compared to non-polluted scenarios, under slight pollution, BFTBrain incurs only a 0.7% performance drop, while ADAPT suffers from a 12% drop. Thus, after such slight pollution where only less than 0.01% of the data population is polluted, BFTBrain outperforms ADAPT by 28%. Under severe pollution, although the f malicious learning agents cause a distribution shift in BFTBrain's global state and reward, BFTBrain only incurs a 0.5% performance drop. The reason is, as proved in Section C.2, the learning-coordination protocol guarantees that the global feature and reward always fall into the range between two honest local observations. On the contrary, when all features and rewards can be arbitrarily manipulated by the centralized entity in ADAPT, it performs no better than randomly choosing protocols. In the worst case, a smart pollution strategy misguides ADAPT to pick the worst protocol for each condition (as shown by the ADAPT (severe pollution) line in Figure 4), resulting in a 55% performance drop. In such a scenario, BFTBrain outperforms ADAPT by 154%.

7.6 Overhead of BFTBrain

Our last experiment evaluates the overhead incurred by BFTBrain's learning framework. We repeated the "cycle back conditions" experiment in Section 7.3, and plot BFTBrain's training and inference overhead in each epoch in Figure 15 (provided in Appendix D.4). Labels indicate the dominant protocol that BFTBrain chooses in each segment of the figure.

The training overhead increases quasi-linearly in each segment, but zigzags across different segments. The reason is that BFTBrain's training overhead increases with the number of data points in the experience bucket, and the bucket used for training (i.e., one out of K^2) depends on the previous and current protocol. Within one segment, since BFTBrain chooses the best-performing protocol dominantly after convergence, a certain bucket is selected dominantly and accumulates training data. The drops of overhead in the figure is caused by BFTBrain's exploration of other protocol, resulting in other smaller buckets being retrained. On the contrary, the inference overhead is always constant with the number of epochs.

We also measured the duration of epochs in this experiment: 0.88s in minimum and 1.31s in average. During the entire 4-hour deployment, compared to the duration of epochs, even the maximum training and inference overhead is negligible, due to the lightweight nature of random forests. More importantly, the learning agent and validator are two parallel processes. Thus, the overhead of learning does not adversely affect the node's throughput, as long as there are some spare CPU cycles devoted to the learning agent. When BFTBrain is deployed for a longer run, techniques such as periodic resampling and limiting the size of the experience bucket can be utilized [54] to control the overhead of learning.

8 Related Work

The BFT consensus problem has been studied extensively, and surveying it is beyond the scope of this work; readers may find introductory material in textbooks, various surveys, and measurement studies, e.g., [7, 17, 25, 26, 51, 56, 58, 61, 66].

BFTBrain employs a mechanism similar to *reconfiguration* which allows the system to switch the parameters of a BFT algorithm by agreement at certain positions within a sequence of decisions. The reconfiguration problem has been formalized in several works, both in the benign fault settings and the Byzantine setting, e.g., [8, 13, 16, 38, 42, 49, 50, 57].

To our knowledge, the first work to propose switching between a set of BFT protocols in real-time to adapt to dynamic conditions is Abstract [8]. It was followed by ADAPT [10], which further enhanced the approach by training a supervised learning model to govern switching. BFTBrain embraces the idea of a multi-protocol BFT engine while considerably enhancing its practicality via a decentralized reinforcement learning engine that provides significant operational benefits and by introducing deep performance metrics, capturing faults, and being hardware agnostic.

At the core of BFTBrain, learning in untrusted distributed settings is made resilient against intentional data pollution via robust aggregation and agreement. Similar techniques have been explored in robust distributed learning studies [3, 11, 32, 39, 44, 64]. More generally, harnessing learning to enhance performance has been done successfully in many systems under the umbrella of machine programming [36]: indexing [47], query optimization [54, 63, 68], database tuning [59], software analysis [41], scheduling [53], concurrency control [60], and transaction management in blockchains [62].

9 Conclusion

Existing BFT protocols lack flexibility and adaptability, leading to suboptimal performance in various scenarios. In this paper, we propose a practical reinforcement learning-based BFT system called BFTBrain, which dynamically selects the top-performing BFT protocols in real-time. Our extensive evaluation demonstrates that BFTBrain significantly outperforms existing solutions under various conditions, including dynamic environments and adversarial attacks.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback and suggestions. This work is funded by NSF grants CNS-2104882 and CNS-2107147.

References

[1] The diem team. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf, 2021.

[2] Shipra Agrawal and Navin Goyal. Further optimal regret bounds for thompson sampling. In *The International Conference on Artificial Intelligence and Statistics*, AISTATS '13.

[3] Youssef Allouah, Rachid Guerraoui, Nirupam Gupta, Rafael Pinot, and Geovani Rizk. Robust distributed learning: Tight error bounds and breakdown point under data heterogeneity. In *Conf. on Neural Information Processing Systems (NeurIPS)*, 2023.

[4] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[5] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The bedrock of byzantine fault tolerance: A unified platform for {BFT} protocols analysis, implementation, and experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 371–400, 2024.

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, and Yacov Manevich. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*, pages 30:1–30:15. ACM, 2018.

[7] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[8] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[9] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. Deploying intrusion-tolerant scada for the power grid. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 328–335. IEEE, 2019.

[10] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. Making bft protocols really adaptive. In *Int. Parallel and Distributed Processing Symposium*, pages 904–913. IEEE, 2015.

[11] Gilad Baruch, Moran Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[12] Omar Besbes, Yonatan Gur, and Assaf Zeevi. Stochastic multi-armed-bandit problem with non-stationary rewards. In *Advances in neural information processing systems*, NIPS '14, pages 199–207.

[13] Alysson Bessani, Joao Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 355–362. IEEE, 2014.

[14] Leo Breiman. Bagging predictors. In *Machine Learning*, Maching Learning '96.

[15] Leo Breiman. Random forests. 45(1):5–32.

[16] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350. USENIX Association, 2006.

[17] Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. In *Int. Symposium on Distributed Computing (DISC)*, pages 1–16, 2017.

[18] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX Association, 1999.

[19] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[20] Lujing Cen, Ryan Marcus, Hongzi Mao, Justin Gottschlich, Mohammad Alizadeh, and Tim Kraska. Learned garbage collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL @ PLDI '20. ACM.

[21] Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, NIPS'11.

[22] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Symposium on Operating Systems Principles (SOSP)*, pages 277–290. ACM, 2009.

[23] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 9, pages 153–168. USENIX Association, 2009.

[24] Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. Be aware of your leaders. In *Int. Conf. on Financial Cryptography and Data Security*, pages 279–295. Springer, 2022.

[25] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Verissimo. Byzantine consensus in asynchronous message-passing systems: a survey. *Int. Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

[26] Tobias Distler. Byzantine fault-tolerant state-machine replication from a systems perspective. *ACM Computing Surveys (CSUR)*, 54(1):1–38, 2021.

[27] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. Powerstore: Proofs of writing for efficient and robust storage. In *Conf. on Computer and communications security (CCS)*, pages 285–298. ACM, 2013.

[28] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *Annual Technical Conf. (ATC)*, pages 1–14. USENIX Association, 2019.

[29] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[30] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 215–229. ACM, 1985.

[31] Amr El Abbadi and Sam Toueg. Availability in partitioned replicated databases. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 240–251. ACM, 1986.

[32] Sadegh Farhadkhani, Rachid Guerraoui, Nirupam Gupta, Rafael Pinot, and John Stephan. Byzantine machine learning made easy by resilient averaging of momentums. In *Int. Conf. on Machine Learning (ICML)*, pages 6246–6283. PMLR, 2022.

[33] Miguel Garcia, Nuno Neves, and Alysson Bessani. An intrusion-tolerant firewall design for protecting siem systems. In *Conf. on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–7. IEEE, 2013.

[34] Miguel Garcia, Nuno Neves, and Alysson Bessani. Sieveq: A layered bft protection system for critical services. *IEEE Transactions on Dependable and Secure Computing*, 15(3):511–525, 2016.

[35] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 135–144. IEEE, 2004.

[36] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B. Tenenbaum, and Tim Mattson. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 69–80. Association for Computing Machinery.

[37] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. Diablo: A benchmark suite for blockchains. In *European Conf. on Computer Systems (EuroSys)*. ACM, 2023.

[38] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *European conf. on Computer systems (EuroSys)*, pages 363–376. ACM, 2010.

[39] Rachid Guerraoui, Sébastien Rouault, et al. The hidden vulnerability of distributed learning in byzantium. In *Int. Conf. on Machine Learning (ICML)*, pages 3521–3530. PMLR, 2018.

[40] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE/IFIP, 2019.

[41] Niranjan Hasabnis and Justin Gottschlich. ControlFlag: a self-supervised idiosyncratic pattern detection system for software control structures. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, MAPS '21, pages 32–42. Association for Computing Machinery.

[42] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *Annual Technical Conf. (ATC)*. USENIX Association, 2010.

[43] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European Conf. on Computer Systems (EuroSys)*, pages 295–308. ACM, 2012.

[44] Sai Praneeth Karimireddy, Lie He, and Martin Jaggi. Learning from history for byzantine robust optimization. In *Int. Conf. on Machine Learning (ICML)*, pages 5311–5319. PMLR, 2021.

[45] Jonathan Kirsch, Stuart Goose, Yair Amir, Dong Wei, and Paul Skare. Survivable scada via intrusion-tolerant replication. *IEEE Transactions on Smart Grid*, 5(1):60–70, 2013.

[46] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *Operating Systems Review (OSR)*, 41(6):45–58, 2007.

[47] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18. ACM.

[48] Jae Kwon. Tendermint: Consensus without mining. 2014.

[49] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[50] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *symposium on Principles of distributed computing (PODC)*, pages 312–313. ACM, 2009.

[51] Dahlia Malkhi. *Concurrency: the works of Leslie Lamport*. ACM, 2019.

[52] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[53] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters.

[54] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21. Award: 'best paper award'.

[55] Ian Osband and Benjamin Van Roy. Bootstrapped thompson sampling and deep exploration.

[56] Marco Platania, Daniel Obenshain, Thomas Tantillo, Yair Amir, and Neeraj Suri. On choosing server-or client-side solutions for bft. *ACM Computing Surveys (CSUR)*, 48(4):1–30, 2016.

[57] Peiyao Sheng, Chenyuan Wu, Dahlia Malkhi, Michael K. Reiter, Chrysoula Stathakopoulou, Michael Wei, and Maofan Yin. On orchestrating parallel broadcasts for distributed ledgers. Cryptology ePrint Archive, Paper 2024/923, 2024.

[58] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *Symposium on Networked Systems Design and Implementation (NSDI)*, volume 8, pages 189–204. USENIX Association, 2008.

[59] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024. ACM.

[60] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: {High-Performance} transactions via learned concurrency control. OSDI '21, pages 198–216.

[61] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. Bft in blockchains: From protocols to use cases. *ACM Computing Surveys (CSUR)*, 54(10s):1–37, 2022.

[62] Chenyuan Wu, Bhavana Mehta, Mohammad Javad Amiri, Ryan Marcus, and Boon Thau Loo. AdaChain: A learned adaptive blockchain. *Proc. of the VLDB Endowment*, 16(8):2033–2046, 2023.

[63] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 931–944. Association for Computing Machinery.

[64] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Int. Conf. on Machine Learning (ICML)*, pages 5650–5659. PMLR, 2018.

[65] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*, pages 347–356. ACM, 2019.

[66] Gengrui Zhang, Fei Pan, Michael Dang'ana, Yunhao Mao, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms. *arXiv preprint arXiv:2204.03181*, 2022.

[67] Li Zhou. A survey on contextual multi-armed bandits.

[68] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. Lero: A learning-to-rank query optimizer. 16(6):1466–1479.

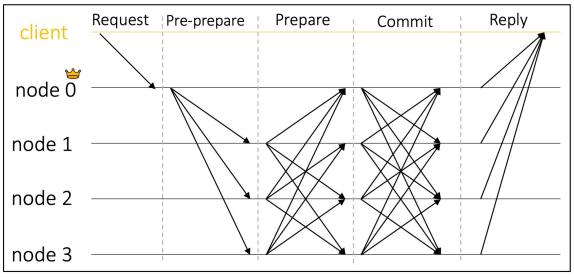


Figure 5: PBFT protocol

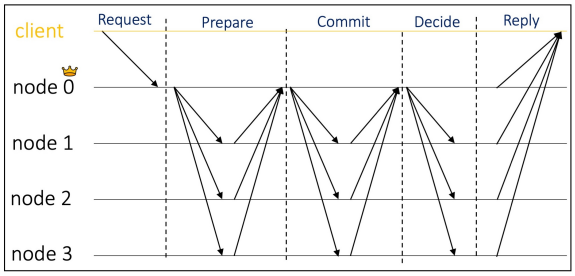


Figure 6: HotStuff-2 protocol

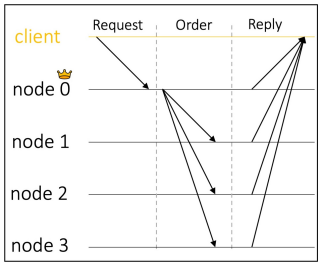


Figure 7: Zyzzyva protocol (fast path)

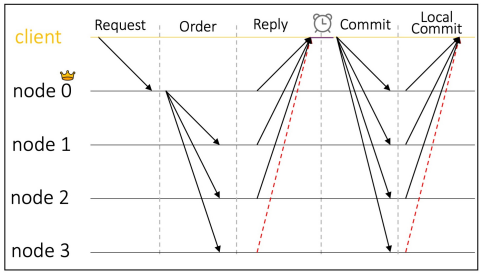


Figure 8: Zyzzyva protocol (slow path)

A Background on Selected BFT Protocols

In this section, we provide background on BFT protocols that are experimentally studied in Section 2.

PBFT [18, 19]. PBFT (Figure 5) is a leader-based protocol that operates in a succession of configurations called *views* [30,31]. Each view is coordinated by a *stable* leader (primary). PBFT consists of pre-prepare, prepare, and commit phases. The pre-prepare phase assigns an order to the request, the prepare phase guarantees the uniqueness of the assigned order, and the commit phase guarantees that the next leader after view-change can safely assign the order.

During a normal case execution of PBFT, clients send their signed request messages to the leader. In the pre-prepare phase, the leader assigns a sequence number to the request to determine the execution order of the request and multicasts a pre-prepare message to all backups. Upon receiving a valid pre-prepare message from the leader, each backup node multicasts a prepare message to all nodes and waits for prepare messages from $2f$ different nodes (including the node itself) that match the pre-prepare message. The goal of the prepare phase is to guarantee safety within the view, i.e., $2f$ nodes received matching pre-prepare messages from the leader node and agree with the order of the request.

Each node then multicasts a commit message to all nodes. Once a node receives $2f+1$ valid commit messages from different nodes, including itself, that match the pre-prepare message, it commits the request. The goal of the commit phase is to ensure safety across views, i.e., the request has been voted on a majority of non-faulty nodes and can be recovered after (leader) failures. The second and third phases of PBFT follow the *clique* topology, i.e., have $O(n^2)$ message complexity. If the node has executed all requests with lower sequence numbers, it executes the request and sends a reply to the client. The client waits for $f+1$ matching results from different nodes.

HotStuff-2 [52]. HotStuff-2 (Figure 6) HotStuff-2 is a leader-based BFT protocol with two main properties. First, it provides linear communication complexity (rather than quadratic as in PBFT). Specifically, each all-to-all communication phase of PBFT is replaced with two linear phases in HotStuff; one from the replicas to the leader and one from the leader to the replicas. Second, HotStuff-2 uses the leader rotation technique, where the leader is replaced after every single proposal in a predetermined manner (round-robin). This is in contrast to most existing protocols that rely on a stable leader, and the leader is changed only when it is suspected to be faulty. Compared to the original HotStuff [65], HotStuff-2 reduces one phase of communication, improving the overall performance of the protocol. Chaining (also proposed in [65]) is an optimization technique that is applicable to both HotStuff and HotStuff-2. The chaining optimization involves executing some phases which are deemed identical in a pipelined manner in order to reduce the latency of request processing.

Zyzzyva [46]. Zyzzyva (Figure 7) optimistically assumes that the leader and all backups are non-faulty. Hence, upon receiving an order message from the primary node which includes the request, nodes speculatively execute requests without running any agreement and send reply messages to the client. The client waits for $3f+1$ matching replies to accept the results. If the client timer is expired and the client has received matching replies from between $2f+1$ and $3f$ nodes, as presented in Figure 8, two more linear rounds of communication are needed to ensure that at least $2f+1$ nodes

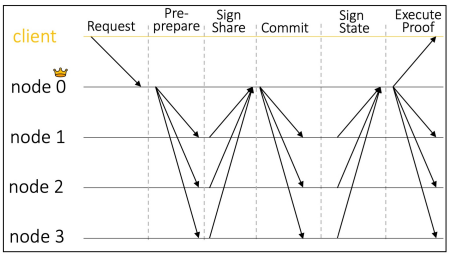


Figure 9: SBFT protocol (fast path)

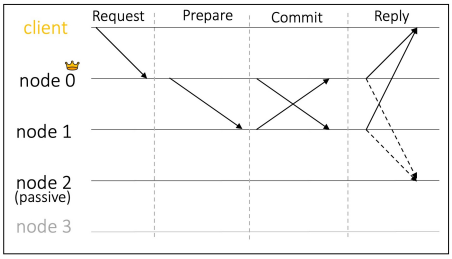


Figure 11: CheapBFT protocol

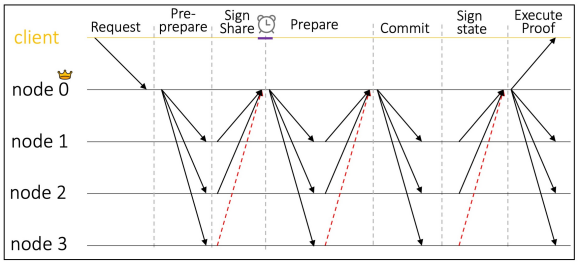


Figure 10: SBFT protocol (slow path)

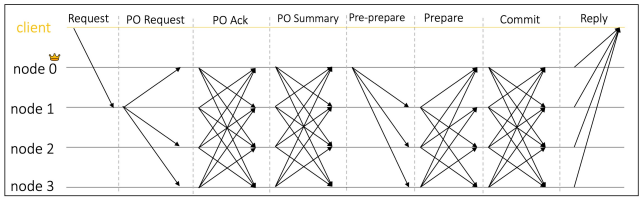


Figure 12: Prime protocol

have committed the request.

SBFT [40]. SBFT[4] presents a linear optimistic fast path (Figure 9), assuming all nodes are non-faulty. In SBFT, upon receiving a pre-prepare message from the primary node, all nodes send sign-share messages to the commit collector (i.e., the primary in our figure). If the commit collector is able to collect $3f+1$ sign-share messages, it puts them together to generate a commit message and broadcasts it to the backups. Otherwise (the commit collector does not receive messages from *all* backups in the sign-share phase and its timer is expired), SBFT switches to its slow path (Figure 10) and requires two more linear rounds of communication (prepare phase). The dual-path nature of SBFT requires nodes to sign each message with two schemes (i.e., $2f+1$ and $3f+1$). To send replies to the client, a single (execution collector) node receives replies from all nodes and sends a single (threshold) signed reply message.

CheapBFT [43]. CheapBFT (Figure 11) differs from PBFT in two major ways. First, it puts f nodes as passive nodes by optimistically assuming all nodes participating in the quorum (i.e., called *active* nodes) are honest. Second, it relies on trusted hardware to prevent equivocation. CheapBFT reduces the required quorum of active nodes to $f+1$ (the total number of nodes becomes $2f+1$) in its normal case execution, and reduces the number of phases from three to two as compared with PBFT (i.e., no pre-prepare phase). If an active node becomes faulty, it is replaced with a passive node.

Prime [4]. Prime (Figure 12) is a robust protocol by incorporating a pre-ordering stage. In the pre-ordering stage, nodes exchange the requests they receive from clients and periodically share a vector of all received requests, expecting the leader to globally order requests following those vectors. In this way, nodes can also monitor the leader to order requests in a fair manner. Prime also uses timers to periodically check performance of the protocol and replace the primary node if it does not provide acceptable performance (e.g., under slowness attack) corresponding to the current network condition.

[4] SBFT tolerates both crash and Byzantine failure ($n=3f+2c+1$ where c is the number of crashed nodes). Since the focus of this paper is on Byzantine failures, we consider a variation of SBFT where $c=0$.

B Switching BFT Protocols

In this section, we describe how BFTBrain performs the actual switching from BFT protocol t to $t+1$.

Similar to Abstract, BFTBrain operates in epochs. Each epoch in BFTBrain can be considered as a *Backup* instance in Abstract, which is marked by the completion of k requests, where k is a predefined constant hyper-parameter. For each epoch, an existing BFT protocol in our action space (i.e., PBFT, Zyzzyva, CheapBFT, Prime, SBFT, and HotStuff-2) is selected by the learning agent and remains unchanged throughout the epoch.

The *Backup* instance in Abstract is a thin wrapper that can be put around an existing BFT protocol. *Backup* works as follows: it is inactive until it receives a request containing a valid init history, which is an unforgeable history generated by the preceding BFT instance. At this point, *Backup* sets its state by executing the requests in the init history it received. Then, it simply executes the first k requests committed by the underlying BFT protocol, aborts all subsequent requests, and returns the signed sequence of executed requests as the init history for the next BFT instance. A client can switch from a *Backup* instance t to instance $t+1$ as soon as it receives $f+1$ signed messages from different replicas, containing an identical init history. The client uses the init history to invoke instance $t+1$. Once instance t aborts some requests and switches to $t+1$, it cannot commit any subsequently invoked requests.

Optimizations of Abstract. Abstract also implements the following optimization for the switching procedure above: align switching from a *Backup* instance with a checkpoint in order to minimize the size of init history, which is transferred through clients. In Abstract, BFT instances t and $t+1$ might be run on two different clusters, and not every instance is a *Backup* instance whose progress condition is k committed requests. For example, an instance could be *Quorum* whose progress condition is $3f+1$ matching history, or *Chain* whose progress condition is one correct reply, both of which require an honest client to validate the progress condition and initiate the switching. On the contrary, in BFTBrain, all BFT instances are run on the same cluster of machines, and every BFT instance is a *Backup* instance. Thus, the optimization can be taken one step further inside BFTBrain. First, instead of relying on the client, each replica has enough information to decide whether to switch or not. Once executing k requests, each replica multicasts an init history (i.e., checkpoint) to all other replicas. Second, an honest replica does not need to be blocked and wait for $f+1$ signed matching init history and execute it before starting the new epoch, since the init history is already reflected in its local service state. Once k requests are committed in epoch t and the BFT protocol for $t+1$ is derived by the learning agent, epoch $t+1$ is invoked. In other words, the switching can be performed asynchronously inside BFTBrain with low overhead.

Speculative Backup instances. *Backup* can be easily implemented over an existing BFT protocol, except for speculative protocols where clients are the commit collectors, e.g., Zyzzyva. This is because such protocols rely on clients to decide if a request has been committed, while the replicas might not have enough information to tell. Specifically, in Zyzzyva, if a request commits in the slow path where the client only receives $2f+1$ replies with matching history (instead of $3f+1$ replies in its fast path) and multicasts a commit certificate to all replicas, the replicas can deterministically commit the request as well. However, if a request commits in the fast path, the client directly completes the request without notifying the replicas. Thus, the replicas can not decide whether the current epoch is finished.

To address this issue, when running speculative protocols in epoch t, BFTBrain enforces the k-th request (i.e., the last request in t) to be a special NOOP request and to be committed in the slow path. On each replica, the view-change timer for this request is only removed when it can be considered as committed. Specifically, for the k-th request, the leader acts as a dummy client and multicasts a commit certificate even if $3f+1$ replies with matching history are received. Upon receiving the commit certificate or $f+1$ signed matching init history for epoch $t+1$, replicas can safely consider the k-th request of epoch t as completed. Note that a malicious dummy client might refuse to multicast the commit certificate. In this case, view-change is triggered and a new leader will be elected, which will act as a new dummy client for this request. Then, in the NEW-VIEW message, for this k-th request, the new leader always inserts a ORDER-REQ message where the original NOOP request is replaced by the NOOP request proposed by its own dummy client. For this special request, no safety issues exist since its content is not part of the service state. The mechanisms above guarantee that each honest replica will successfully commit this request (i.e., liveness), and thus concluding the current epoch.

Correctness of switching. Below, we discuss the safety and liveness of switching between epochs.

Safety. Abstract's idempotency theorem [8] specifies that if individual BFT instances are correct, irrespectively of each other, then the system composed through switching is also correct. Since each BFT instance in BFTBrain is an existing protocol whose safety has been previously proven (instead of our newly invented protocol), the composed system is safe.

Liveness. Abstract guarantees liveness if a request is not aborted by all instances, which can be made simple by reusing an existing BFT protocol as one of the instances. Apparently, BFTBrain satisfies this requirement too.

Moreover, to ensure liveness, Abstract exponentially increases the parameter k with every new instance of *Backup* by default. This is to prevent a corner case with very slow clients: k requests committed by a single *Backup* instance i might all be invoked by the same, fast client, while a slow client can then get its requests aborted by i. The same can happen with a subsequent *Backup* instance, and so forth. By exponentially increasing k, this liveness issue is resolved since no realistic load increases faster than exponentially. Alternatively, by having the replicas across different *Backup* instances share a client input buffer, such an issue can also be prevented without exponentially increasing k. BFTBrain takes the latter approach, since all epochs run on the same cluster, so the client input buffer is naturally shared across epochs.

C Learning Coordination

In this section, we present BFTBrain's learning coordination protocol and discuss the correctness of BFTBrain.

C.1 Detailed Description of The Protocol

Algorithm 1 presents the detailed learning coordination protocol, where VBC (i.e., any "blackbox" validated Byzantine consensus primitive) is implemented using PBFT. As shown in lines 1-7, after executing w requests (a hyper-parameter) in epoch t, each node i collects local performance indicators p_i^{t-1} measured during epoch $t-1$, featurizes the next state f_i^{t+1}, and broadcasts both metrics inside a report message. Each agent i collects the received valid report messages where p_j^{t-1} and f_j^{t+1} are non-null, into a local set $reportQC_i^t$. Upon collecting $f+1$ report messages from different agents, the agent triggers the coordination protocol for epoch t and starts a view-change timer $\tau_{c,1}$ to track the progress (lines 8-11).

Each time an agent l becomes the leader, as shown in lines 12-15 (the view-change leader is omitted), it starts an addi-

Table 3: The throughput of each studied protocol under different conditions. Each row characterizes a different condition, where the best throughput is highlighted in **blue**.

Condition Parameters					Throughput (tps)					
f	# of clients	# of absentees	request size	proposal slowness	PBFT	Zyzzyva	CheapBFT	Prime	SBFT	HotStuff-2
1	50	0	4KB	0ms	9133	**13664**	11822	4601	11067	6882
4	100	0	4KB	0ms	4316	**10699**	7966	4239	6414	7124
4	100	0	100KB	0ms	4261	6513	**7353**	4177	6518	6779
4	100	4	4KB	0ms	5386	1929	**10011**	4440	5347	8848
4	100	0	0KB	20ms	2435	2424	2433	4265	2432	**6201**
4	100	0	1KB	20ms	2435	2424	2432	4211	2433	**6099**
4	100	0	0KB	100ms	497	498	497	**4257**	497	3641
1	50	0	0KB	20ms	989	988	989	**4527**	989	2640

tional timer $\tau_{c,2}$ for collecting $2f+1$ reports. Once the size of $reportQC_l^t$ reaches $2f+1$ or $\tau_{c,2}$ expires, the leader multicasts $\langle \text{c-PROPOSE}, v_c, n_c, t, d \rangle_{\sigma_l}, reportQC_l^t \rangle$ message to all agents, where v_c is the view in the coordination protocol, n_c is the sequence number in the coordination protocol, t is the epoch id, and d is the digest of $reportQC_l^t$.

Once a coordinator agent i receives a valid c-propose with coordination sequence n_c that it has not voted for, it accepts that proposal and multicasts a c-prepare message if: (1) the size of corresponding $reportQC_l^t$ is at least $f+1$, (2) t is not committed and (3) n_c-1 is committed (lines 16-18).

Then, similar to PBFT, after the c-prepare and c-commit phases, the predicate c-committed$(v_c, n_c, t, reportQC_l^t)$ becomes true if $2f+1$ matching c-commit messages from different agents have been received. Once committed, agent i checks the size of $reportQC_l^t$. If smaller than $2f+1$, the learning algorithm (i.e., retraining and inference based on predictive models) is not invoked, and the agent utilizes the same protocol for epoch $t+1$ as in the current epoch t. It also initiates a view-change on the current BFT protocol used by its validator and multicasts a c-view-change message to all other agents. Here, s_c stands for the last stable checkpoint known to i in the coordination protocol, C_c is a set of c-checkpoint messages proving the correctness of s_c, and P_c is a set containing a set $P_{reportQC}$ for each $reportQC$ that prepared at i with a coordination sequence number higher than s_c. Each $P_{reportQC}$ contains the c-propose and c-prepare messages (lines 23-26). Otherwise, if the size is $2f+1$, for reward and each state dimension, the median value is taken as the global reward p^{t-1} and state f^{t+1}. The state-action-reward triplet for epoch $t-1$ is then added to the experience buffer, the predictive models are re-trained, and a promising BFT protocol is inferred for epoch $t+1$ based on state f^{t+1} (lines 27-31). Finally, regardless of the size of $reportQC_l^t$, a c-checkpoint message is multicast to all coordination agents.

Upon timer $\tau_{c,1}$ expires, each coordination agent multicasts a c-view-change message to all agents. Once receiving $2f+1$ c-view-change messages from different agents, the new leader l' begins the c-new-view phase similar to PBFT. At the end of this phase, l' checks all $P_{reportQC}$ of P_c components collected in the view-change quorum. If none of them includes a $reportQC$ for epoch t, l' will multicast a c-propose message to all agents which carries the $reportQC_{l'}^t$ it has collected.

We argue that the leader agent could also deliberately delay c-propose messages. However, unlike the active BFT instance in each epoch that has tens of thousands of requests to commit, this coordination protocol runs only once every epoch. Thus, the performance impact of such malicious behavior is negligible.

C.2 Proof Sketch

Below, we formalize the safety, liveness, and robustness guarantees of BFTBrain, followed by the proof sketch.

Safety. For each epoch t, each honest learning agent will agree on the same feature and the same reward if its learning algorithm is invoked.

Proof: We prove by contradiction that only one $reportQC$ can be committed for the same epoch t. First, by reduction to PBFT, c-propose, c-prepare, c-commit phases guarantee that for each coordination sequence n_c, only one $reportQC$ can be committed. Then, assume two different $reportQC$ for the same epoch t are committed at different n_c: $reportQC_j^t$ committed at $n_{c,1}$, and $reportQC_k^t$ at $n_{c,2}$, where $n_{c,1} < n_{c,2}$ and $D(reportQC_j^t) \neq D(reportQC_k^t)$. At least one honest agent i needs to accept the c-propose for both $reportQC$. Since the network is asynchronous, we consider two cases. If $n_{c,1}$ is committed before $n_{c,2}$ on i, i needs to accept c-propose with $reportQC_k^t$ when t is already committed. Otherwise if $n_{c,1}$ is committed at the same time with or after $n_{c,2}$, i needs to accept $reportQC_k^t$ whose coordination sequence is $n_{c,2}$, before $n_{c,2}-1$ is committed. Both cases contradict with line 17 of Algorithm 1. Thus, only one $reportQC$ can be committed for the same epoch t. Finally, since an honest learning agent takes the median value of the same committed $reportQC$, they agree on the same feature and reward if $reportQC$ includes $2f+1$ reports.

Following this safety property, through deterministic training (i.e., using the same random seed on each learning agent), each honest learning agent in BFTBrain derives the same action (i.e., BFT protocol) for the same epoch t.

Liveness. BFTBrain guarantees that every honest agent eventually invokes its learning algorithm in the same epoch.

Proof: If fewer than $2f+1$ reports are collected, the learning

Algorithm 1 Learning coordination

▷ On each agent i
Upon execution of w requests in epoch t
  if no state transfer happens during epoch $t-1$ **then**
    Record performance p_i^{t-1}
  if no state transfer happens during epoch t **then**
    Extract features f_i^{t+1} from executed requests in window w
  if $p_i^{t-1} \neq null \wedge f_i^{t+1} \neq null$ **then**
    Multicast $\langle \text{REPORT}, t, i, p_i^{t-1}, f_i^{t+1} \rangle_{\sigma_i}$ to all agents
Upon receiving valid REPORT message m
  Add m to $reportQC_i^t$
  if $reportQC_i^t.size = f+1$ **then**
    Start timer $\tau_{c,1}$
▷ On the leader agent l
Upon $reportQC_l^t.size$ reaching $f+1$
  Start timer $\tau_{c,2}$ ▷ $\tau_{c,2} < \tau_{c,1}$
Upon timer $\tau_{c,2}$ timeouts $\vee$ $reportQC_l^t.size = 2f+1$
  Multicast $\langle \text{C-PROPOSE}, v_c, n_c, t, d \rangle_{\sigma_l}, reportQC_l^t \rangle$ to all agents
▷ On each agent i
Upon receiving a valid C-PROPOSE from the leader
  if $reportQC_l^t.size \geq f+1 \wedge t$ is not committed $\wedge n_c - 1$ is committed **then**
    Multicast $\langle \text{C-PREPARE}, v_c, n_c, t, d, i \rangle_{\sigma_i}$ to all agents
Upon receiving valid matching C-PREPARE from $2f+1$ different agents
  Multicast $\langle \text{C-COMMIT}, v_c, n_c, t, d, i \rangle_{\sigma_i}$ to all agents
Upon receiving valid matching C-COMMIT from $2f+1$ different agents
  Commit $(v_c, n_c, t, reportQC_l^t)$
  if $reportQC_l^t.size < 2f+1$ **then**
    $protocol^{t+1} \leftarrow protocol^t$
    Initiate view change on validator i in $protocol^t$
    Multicast $\langle \text{C-VIEW-CHANGE}, v_c+1, s_c, C_c, P_c, i \rangle_{\sigma_i}$ to all agents
  else
    $p^{t-1} \leftarrow \text{median}\{m.p_j^{t-1} | m \in reportQC_l^t\}$
    $f^{t+1} \leftarrow \text{median}\{m.f_j^{t+1} | m \in reportQC_l^t\}$
    Add (f^{t-1}, $protocol^{t-1}$, p^{t-1}) to experience buffer
    $protocol^{t+1} \leftarrow \text{BESTPREDICTEDPROTOCOL}(f^{t+1})$
  Multicast $\langle \text{C-CHECKPOINT}, n_c, t, i, reportQC_l^n \rangle_{\sigma_i}$ to all agents
Upon timer $\tau_{c,1}$ timeouts
  Multicast $\langle \text{C-VIEW-CHANGE}, v_c+1, s_c, C_c, P_c, i \rangle_{\sigma_i}$ to all agents
▷ On the leader agent l' of view v_c+1
if no set in the P_c components has $reportQC_l^t$ **then**
  Multicast $\langle \text{C-PROPOSE}, v_c+1, n_c, t, d \rangle_{\sigma_{l'}}, reportQC_{l'}^t \rangle$ to all agents
▷ The remaing view change routine are omitted here

algorithm will be ineffective (i.e. not invoked) for one epoch, and the BFT protocol chosen for the next epoch will be the same as the current one. We first show agents are able to commit a $reportQC$ of $2f+1$ reports within f consecutive epochs (Part 1). We then show if such $reportQC$ is committed for epoch t, every honest agent will invoke its learning algorithm in epoch t (Part 2).

Part 1: In the active BFT protocol of any certain epoch, there can be at most f honest validators placed in-dark by the malicious leader, since otherwise, $f+1$ validators will trigger a view-change to replace the leader in the epoch. The agents on these in-dark validators will not send report messages according to our protocol. In addition, at most f malicious agents can deliberately refuse to send report messages. Thus, each agent is guaranteed to receive at least $f+1$ reports for epoch t, which starts a timer $\tau_{c,2}$ on leader agent l for collecting $2f+1$ reports in $reportQC_l^t$ and a timer $\tau_{c,1}$ on each agent for tracking the progress of coordination protocol. If $reportQC_l^t$ is committed, a view-change in the active BFT protocol used by validators will be initiated if it contains less than $2f+1$ reports. Since such coordination is run once every epoch, an honest leader validator will be found within at most f consecutive epochs. Otherwise, if $reportQC_l^t$ is not committed, the leader agent will be replaced while the current epoch waits for a report quorum to be committed.

Part 2: If a $reportQC$ of size $2f+1$ is committed on an agent, it is committed on at least $2f+1$ agents, otherwise c-view-change is triggered and it will be committed in view v_c+1 or above. Thus, at least $f+1$ honest agents will multicast a c-checkpoint message containing the $reportQC$ as the service state of the coordination protocol. Every honest agent, even if it is placed in dark by the malicious leader agent, will receive this stable checkpoint and thus invoke its learning algorithm.

Robustness. If honest feature/reward values form a range $[r_l, r_h]$, the global feature/reward taken by the learning agent always falls into this range.

Proof: According to our protocol, honest nodes never report invalid (e.g., zero or null) values to others, even when they are placed in-dark and recover their service state from others.

The learning agent only takes the median value of $reportQC$ of size $2f+1$, where at most f can be arbitrary values reported by malicious replicas, and the remaining $f+1$ is guaranteed to be honest. Now we prove by contradiction: w.l.o.g., assume the median value of $reportQC$ is r_m, where $r_m > r_h$. Since there are at least $f+1$ honest values smaller than r_m, to make r_m the median, at least $f+1$ dishonest values larger than r_m need to be reported. This contradicts the fact that, at most f can be arbitrary values in $reportQC$.

D Additional Experiment Results

D.1 Comprehensive Performance Comparison

Table 3 presents the comprehensive performance comparison results, which were previously summarized by Table 1 in Section 2. Under each different condition (i.e., row), we list the throughput of each protocol in terms of transactions per second. For a fair comparison, the common internal parameters of all six protocols are configured with the same value: we set the batch size to be 10 and the view-change timer to be 100ms. The protocol-specific internal parameters (e.g., leader rotation interval of HotStuff-2, the aggregation delay for global ordering in Prime, the timer that distinguishes fast path vs. slow path in Zyzzyva) are configured with a reasonable value such that the protocol has good performance on all conditions.

D.2 Randomized Sampling

Compared to the "cycle back conditions" experiment in Section 7.3, we created a benchmark that has more variations in the state space and introduced the variations more frequently. Specifically, each dimension in category State 1 and 2 (except $F1$) follows a certain normal distribution independently. We vary each dimension every 1s by randomly sampling from its distribution, and shift the mean and variance of such distribu-

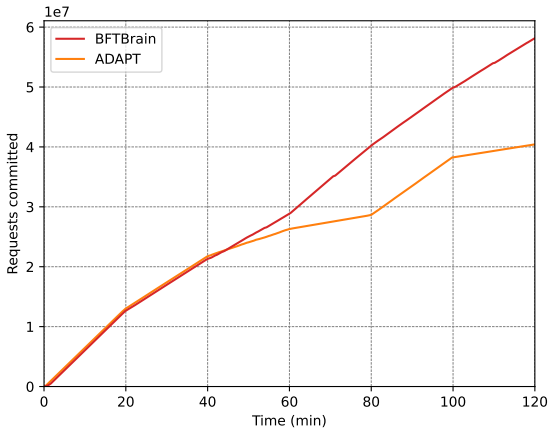


Figure 13: Adaptivity of BFTBrain compared to ADAPT under randomly sampled conditions.

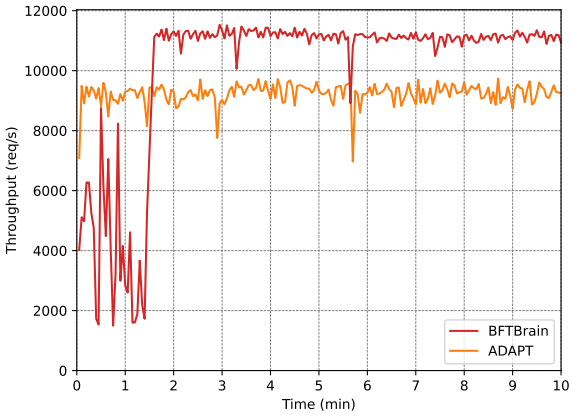


Figure 14: Adaptivity of BFTBrain compared to ADAPT under different hardware (WAN).

tion every 20 minutes. During the first hour of the experiment, all validators ($n = 13$) are responsive, whereas during the second hour, f validators are non-responsive. The entire experiment lasts 2 hours. Compared to other dimensions, we varied $F1$ at a lower frequency, since in real deployment scenarios, client workloads vary more frequently than faults in the system. The exact distribution we used and their shifting pattern can be found here[5].

We compare BFTBrain with state-of-the-art learning based approach ADAPT, which is pre-trained with complete data that we collected in this setup when running BFTBrain. Figure 13 shows the number of committed requests with respect to time. During the first 40 minutes, ADAPT performs even slightly better than BFTBrain. This is because BFTBrain incurs a few explorations (i.e., chooses sub-optimal protocols) when starting the system from scratch and the experience buckets are all empty, while ADAPT exploits its prior knowledge and thus avoids all such explorations. However, starting from 40

[5] https://github.com/JeffersonQin/BFTBrain/tree/master/exp/randomize

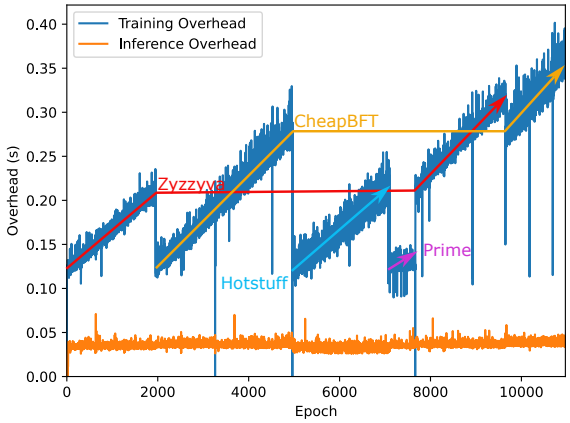


Figure 15: Learning overhead of BFTBrain in each epoch.

minutes, when the condition varies towards a more Byzantine setup (i.e., with persistent in-dark attacks and leaders that appear randomly slow), BFTBrain gradually outperforms ADAPT since the latter does not featurize faults in the system. Thus, although pre-trained with complete data, ADAPT consistently picks sub-optimal protocols since its predictive model does not have enough information to tell how the condition changed. As a result, during the entire 2-hour deployment, BFTBrain commits 44% more requests than ADAPT.

Compared to the 14% improvement in the "cycle back" experiment, BFTBrain improves ADAPT by a larger margin in this experiment. The reason is that some input factors in the cycle back experiment are correlated, e.g., a request size near zero is correlated with high proposal slowness. Thus, although ADAPT suffers from incomplete features, it indirectly learns the optimal protocol under high proposal slowness using other features, as shown in Figure 2 between 90-150 minutes. However, the randomized sampling in this experiment breaks such a correlation.

D.3 Adaptivity under WAN

Figure 14 plots BFTBrain and ADAPT's throughput with respect to time on the "Row 1 (WAN)" setup described in Section 7.4. Here, ADAPT is pre-trained with complete data we collected on "Row 1 (LAN)", while BFTBrain is started from scratch. BFTBrain converges to the optimal protocol CheapBFT after a few explorations. On the contrary, ADAPT does not perform any explorations and performs relatively well from the very beginning, but is stuck at the sub-optimal decision Zyzzyva. This is because the supervised model ADAPT trained is dependent on the hardware. Therefore, ADAPT cannot transfer what it learned on a LAN setup to a WAN setup, unless running a prolonged data re-collection process and re-training the supervised model before this new deployment.

D.4 Overhead of BFTBrain

Figure 15 plots BFTBrain's training and inference overhead in each epoch. Labels indicate the dominant protocol that BFTBrain chooses in each segment of the figure.